Experiences in Data-Parallel Simulation and Analysis of Complex Systems with Irregular Graph Structures

A thesis presented in partial fulfilment of the requirements
for the degree of

Doctor of Philosophy
in
Computer Science

at Massey University, Albany
New Zealand.

Arno Leist

2011

Abstract

The interactions between the components of many natural and artificial systems can be described using a graph. These graphs often have an irregular structure with non-trivial topological features. Complex system behaviour emerges on the macroscopic scale from a large number of relatively simple interactions on the microscopic scale. To better understand the observed behaviour of a complex system, the interactions among its basic elements are commonly described in a computational model. As long as the interactions are defined accurately and the number of elements is large enough for complex patterns to emerge, a simulation based on such a model is expected to produce the same behaviour as the system under investigation.

The difficulty is often to simulate the model on a large enough scale to obtain scientifically meaningful results. Powerful computer systems are required to calculate the effects caused by the interactions of large numbers of elements. Supercomputers that are constructed from hundreds of thousands of processing units can be used to update many components of the system in parallel and thus reduce the overall simulation time, but these systems are expensive to buy and maintain. As the processor architectures used in workstations and regular desktop computers are becoming more powerful, a small cluster constructed from these systems can be a more viable option. In recent years, the highly data parallel architecture of commodity graphics processing units (GPUs) has received a growing amount of attention due to their high peak compute throughput compared to central processing units (CPUs). New software development tools that turn the GPU hardware into a general purpose compute accelerator have become available.

This thesis describes how GPUs can be used to accelerate scientific simulations of complex systems that are based on irregular graph structures. New software development approaches and algorithms are needed to fully utilise the data parallel many-core architecture of today's GPUs. Irregular graph structures are particularly challenging, as the hardware is based on the single instruction, multiple data (SIMD) processor design, where a group of processing elements receives the same instructions. The architecture also imposes strict requirements on memory access patterns, making the optimisation of the memory layout for the irregular data structures and associated information particularly important. Performance suffers dramatically when the algorithm does not comply with these design restrictions.

The author proposes different software design strategies for a number of common graph problems and discusses the advantages and disadvantages of each approach. Two complex system models are used to demonstrate how the GPU can help to accelerate scientific simulations. The first model investigates how the phase transition from ordered to disordered system states in a computational ferromagnet is affected by distortions to the lattice substrate. The second model implements a large scale spiking neural network. The findings show that it is beneficial to utilise the GPU as accelerator to the CPU in almost all scenarios, as long as the project has a long enough run time to justify the more complex software development of the data parallel algorithms. When the model has some regularities in its structure or when some of the design decisions that influence the way memory is accessed can be made with the data parallel architecture in mind, then it is possible to achieve such high performance on the graphics device that it is best to leave the entire computational work to the GPU and use the CPU only to manage the execution of the program.

Acknowledgements

First and foremost I want to thank my parents, Helmut and Gisela, for their continuous support and encouragement. I also want to express my gratitude to my supervisors, Professor Ken Hawick and Associate Professor Chris Scogings, who provided guidance throughout this endeavour. Their experience and expertise have been invaluable to the success of this thesis.

I would like to acknowledge my peers, whose friendship has made the last few years so much more enjoyable and whose knowledge and advice have helped me many times: Dr Daniel Playne, Dr Anton Gerdelan, Dr Andrew Gilman, Dr Guy Kloss, Dr Andre Barczak, Dr Martin Johnson and all the other members of the Complex Systems and Simulations Group at Massey University. I also want to thank everyone who has helped to make me feel welcome and at home in New Zealand.

Finally, I would like to acknowledge the financial support from the Massey University Vice-Chancellor's Doctoral Scholarship and the New Zealand International Doctoral Research Scholarship.

Contents

List of Figures

List of Tables

Part I.

Introduction

Chapter 1

Overview and Aim of this Thesis

This thesis focuses on high performance simulations and analysis of complex systems, in which the structure of interactions between individual objects can be described using a graph. A complex system consists of many interconnected components that interact with each other in often non-linear and non-deterministic ways. The system as a whole exhibits properties that emerge from these interactions but are difficult to predict from the relatively simple rules governing the behaviour of individual parts. Small changes to one part of the system can have complex consequences throughout the whole system, giving rise to emergent behaviour.

Most systems of nature, as well as many artificial systems, are too complex for analytical solutions. Computer simulations can be used instead to analyse abstract models of such systems. Examples for simulations of both natural and artificial complex systems include weather forecast models [1], fluid dynamics [2–4], behaviour of ant colonies [5], stock price forecasting [6], and Animat models [7, 8]. Emergence of complex behaviour and patterns from a multitude of relatively simple interactions are central to complex systems.

To observe multi-length scale properties of complex systems – like the shift of the phase transition between ordered and unordered states in the rewired Ising model discussed in Chapter 7 or the power-law degree distribution in the connectivity of the World Wide Web [9] – it is often necessary to examine the system over several decades of scale. The simulation environment must be able to support system sizes large enough for these scaling effects to show up. Moreover, an increase in the system size or number of discrete simulation steps generally tends to improve the accuracy of the results and is therefore always desirable.

1.1. High Performance Computing

Powerful compute architectures are needed to achieve these goals and improve our understanding of complex systems. The obvious approach to increasing the compute capabilities of a system beyond what is achievable with a single processor is to add additional compute units. In the past, only specialised supercomputers were based on massively parallel system designs. Machines such as the Distributed Array Processor (DAP) [10], the Cray 1 [11], the Transputer [12], the Connection Machine [13] and their respective successors have pioneered many parallel processing techniques. Wide vector functional units were commonly used in addition to scalar or superscalar processing elements to increase the compute throughput for data parallel tasks, which are found in many scientific applications.

Vector processors are constructed from a large number of relatively simple processing elements. The control unit issues a single instruction to the vector processor and each processing element executes this instruction on a different data element in parallel. This class of parallel computers is accordingly referred to as single instruction, multiple data (SIMD). Scalar processor architectures, on the other hand, execute

one instruction on one data element at a time and are thus called single instruction, single data (SISD) in Flynn's taxonomy [14]. The next evolutionary step are superscalar hardware designs, which combine multiple execution units to achieve instruction level parallelism. This is one common form of a multiple instruction, multiple data (MIMD) architecture. For the sake of completeness, the final classification of machine organisation is the multiple instruction, single data (MISD) model, which is of no further relevance to this thesis.

While supercomputers have always been highly parallel, desktop systems have typically only had one central processing unit (CPU) with a single superscalar core. Although most CPU designs have integrated short vector extensions, mainly for multimedia processing, for over a decade, the vast majority of software was not written with parallelism in mind. This worked well as long as new processor generations became faster from higher clock rates, improved instruction level parallelism, pipelining, better memory caching strategies and other improvements that did not require any significant changes to the code to take advantage of the additional performance. But in recent years clock speeds have increased much more slowly, as chip manufacturers are finding it more and more difficult to deal with the thermal power produced at high processor frequencies. As a consequence, new CPUs now come with an increasing number of independent processing cores and wider vector units instead of higher clock rates. To really utilise these new architectures, software developers have to embrace both task and data-level parallel programming models [15, 16].

1.2. The GPU as Compute Accelerator

CPUs are not the only parallel processors found in most computers. To keep up with the demands from the multimedia industries, in particular the video gaming market, specialised graphics processing units (GPUs) have evolved into highly data parallel compute devices. The SIMD architecture is well suited for the rendering of detailed 3D scenes, where many pixels need to be updated as quickly as possible to create the final 2D image from the 3D projection. As GPUs became more powerful and flexible, the first attempts were made to utilise the processing resources for general computational tasks like real-time robot motion planning [17], artificial neural network modelling [18] or real-time cloud simulation and rendering [19]. The term general-purpose computation on graphics processing units (GPGPU) was coined [20]. As the advantages of low cost commodity GPU hardware for scientific simulations became more widely known, the first GPU clusters were built [21]. And according to the June 2011 supercomputer list [22], three of the top 10 fastest computers now utilise GPUs as general-purpose compute accelerators.

In the beginning, graphics hardware had to be programmed using low-level assembly language. As computer graphics became more complex, high-level shading languages like C for Graphics (Cg) were developed to keep up with the requirements and to make the hardware more accessible to software developers. But it was not until GPU manufacturers realised the potential of their hardware for general purpose computation and released their first software development kits (SDKs) dedicated to GPGPU, most importantly NVIDIA's compute unified device architecture (CUDA) [23] and AMD's Stream SDK [24], that a larger audience became aware of the capabilities of today's GPUs. Especially NVIDIA has been putting a lot of marketing and development efforts into their CUDA toolkit and hardware over the past years, trying to attract mass market consumer interest with GPU accelerated real-time physics for computer games [25] and the high performance computing community with features like large amounts of device memory as well as error detection and correction that are only found on their professional *Tesla* device series.

Optimal data organisation and movement has always been critical to the performance of supercomputers [26]. With the increasingly parallel nature of today's desktop, workstation and server machines, similar issues have to be dealt with to fully utilise these new hardware designs. Many-core accelerators like GPUs in particular are not all that dissimilar to highly data parallel supercomputers from a few decades ago put on a single chip. And the processors and accelerators used in many of today's fastest compute clusters [22] are

closely related to the CPUs and GPUs installed in consumer systems. Software designs are converging and scientific simulations need to be optimised for the multi and many-core architectures found in current and future hardware, no matter if they are running on a single workstation with a couple of graphics accelerators or on a compute cluster with hundreds of thousands or even millions of processors.

The recent introduction of the Graph500 list of supercomputers [27] emphasises the increasing importance of data intensive, graph-based algorithms in high performance computing. And while the cost to performance ratio of today's GPUs makes them a very tempting platform for all kinds of scientific simulations, the graphs used to describe the interactions of many complex systems are fundamentally irregular, with non-trivial topological features that do not easily map to a SIMD architecture and thus pose a challenge for general purpose computation on GPUs [28–30]. The major task of this research is therefore to investigate if the highly data-parallel architecture of modern GPUs is a suitable platform for scientific simulations based on complex networks with irregular data structures. The main contributions of this dissertation are as follows:

- Clearly identified that the GPU can provide competitive performance to the CPU for the majority of algorithms that operate on irregular graph structures, with the potential of significant speed-ups for algorithms that are particularly well suited for the GPU architecture.
- Demonstrated and discussed the strengths and weaknesses of the GPU architecture by use of a large number of relevant scenarios.
- Proposed a number of graph algorithms specifically optimised for GPUs.
- Proposed two implementations of the well-regarded Marsaglia lagged-Fibonacci random number generator [31] optimised for the GPU.
- Analysed the scaling of the critical temperature T_c of the Ising model [32] with respect to small-world perturbations to the lattice structure.
- Analysed the frequency domain effects of small-world rewired connections in the lattice structure of the Ising model.
- Demonstrated the use of the GPU for both compute tasks and graphics tasks and proposed interactive visualisation techniques that utilise this interoperability to give insights into the behaviour of 3-dimensional simulation models.

1.3. Thesis Structure

The thesis is structured into several major parts, each with a number of chapters that focus on different aspects of complex systems simulations on GPUs. In this introductory part, Chapter 2 introduces concepts of graph theory. Particular emphasis is put on small-world network structures, a type of complex graph that is of significance to many natural systems. Next, Chapter 3 describes the multi and many-core compute architectures relevant to this thesis, with emphasis on graphics accelerators that are based on the CUDA parallel computing architecture and the associated programming model.

The second part focuses on parallel graph algorithms. The first chapter in this part discusses parallel random number generation. Although not directly related to graphs, many of the following algorithms consume random deviates, making a fast and high quality random number generator on the same platform a necessity. Chapter 5 discusses the generation of synthetic graph structures and investigates different approaches to the management of dynamic memory on the graphics hardware. Following next, Chapter 6 discusses two graph analysis algorithms. The first of which is the well known problem of labelling the

connected components in a network structure. The author proposes a number of approaches to perform this task in parallel on the GPU. The second algorithm is a clustering metric that is particularly relevant to small-world graphs and poses some difficult algorithmic challenges on a SIMD architecture.

Part three introduces two complex system models and describes how the GPU can be used to accelerate the simulation and analysis algorithms to a point where scientifically relevant results can be obtained from individual computer systems and small cluster installations. Chapter 7 analyses the phase transition from ordered to disordered system states that is fundamental to the Ising model – a model of a computational ferromagnet – and how the critical point is shifted as the network structure is distorted. Chapter 8 introduces a neural network simulation that is based on a model of the cortex. The system is designed to study the effects of anaesthesia on the activity patterns of the neural network. In this document, the author concentrates on the algorithmic challenges encountered while implementing the model on the GPU and leaves major simulation work open for future projects.

In the last part, the author offers a discussion and conclusions based on the experiences gathered throughout the work that is presented here. The discussion in Chapter 9 looks at different parallel processor architectures and what can be expected from future developments. It also describes a number of parallel programming models and frameworks and how they relate to CUDA. Finally, Chapter 10 summarises the findings and concludes the dissertation.

Chapter 2

Graph Structures

Graphs model the pairwise relations between objects. They can be used to represent a wide variety of both natural and artificial structures, including communication networks, the power grid, business work-flows, food-chains or the neural networks in our brains to name but a few.

While the structures modelled by a graph can be very complex, the building blocks used to represent these structures are simple. A graph $G \equiv (V, E)$ consists of a set of vertices V, the nodes of the network, and a set of edges E, the connections between those vertices. The edges can either be undirected (symmetric), which means that no distinction is made between the two end points of the edge, or directed (asymmetric), in which case the vertex at one end points to the vertex at the other end. To avoid confusion, in this thesis the term *edge* is used for undirected links and E is the set of unordered pairs of vertices. Directed edges are referred to as *arcs* and A is the set of ordered pairs of vertices. Graphically, an edge $\{u, v\}$ is represented by a simple line between two vertices, whereas an arc (x, y) is considered to be directed from x to y and is represented by an arrow with the arrow head pointing towards vertex y.

Graph structures come in all possible forms, from simple d-dimensional lattices, where every vertex is connected to its $2d$ nearest neighbours, to random graphs, where any two vertices are connected with a certain probability, to more complex network structures like small-world and scale-free graphs. The following sections look at some structures that are relevant in the context of this thesis, with special emphasis on small-world graphs.

2.1. A Regular d-Dimensional Lattice

Many naturally occurring solid materials have a 3-dimensional crystalline structure. In its simplest variety – the simple cubic crystal system – an atom, molecule or ion sits on each corner of a cube and is shared equally by up to eight adjacent cubes [33]. This arrangement defines the basic structural pattern that is repeated in all three dimensions. Figure 2.1 shows a network model of this crystalline structure. Each vertex is connected to its six nearest neighbours, with the exception of vertices on one of the outer faces of the system. This crystalline structure, extended into any number of dimensions, represents a regular d-dimensional lattice.

In computer simulations it is sometimes useful to simulate a homogeneous system with periodic boundaries. This means that the "edges" of the regular lattice wrap around, connecting the vertices to their counterparts on the far end of the system in all dimensions, which effectively removes the boundaries. For example, the node in the top left corner of a 2D grid is connected to the nodes in the top right and bottom left corners in addition to the nodes directly to the right and below. Every node in this network has the same number of neighbours. While this removes boundary effects, the system is still characterised by the finite

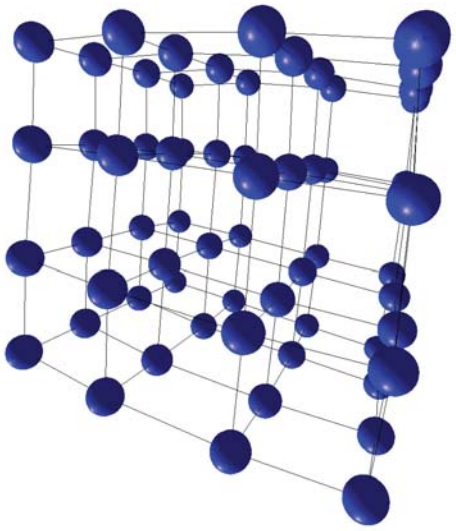


Figure 2.1.: The simple cubic crystal structure represented by a regular cubic lattice with $4 \times 4 \times 4$ nodes. Visualised using the force-based layout algorithm in GraViz3D[a].

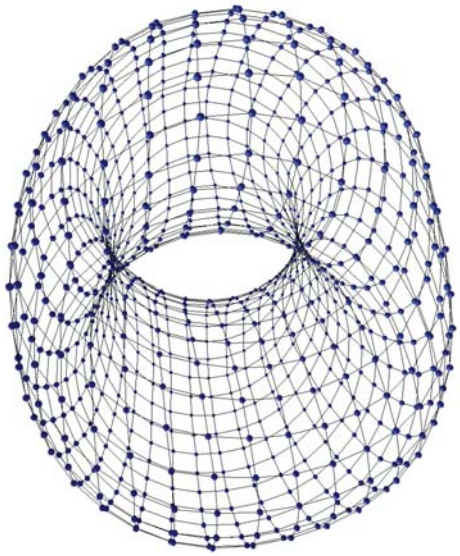


Figure 2.2.: A regular 2D lattice with 30×30 nodes and periodic boundaries can be represented as a torus. Visualised using the force-based layout algorithm in GraViz3D.

[a] GraViz3D is a high performance graph visualisation program developed by the author. It utilises the processing units of both the CPU and the GPU to speed-up the layout algorithms and rendering.

lattice length and the system properties differ to those of the infinite lattice [34]. A 2D regular lattice with periodic boundaries has the shape of a torus, as illustrated in Figure 2.2.

Because of their regularity, the graph structures for these lattices are very simple. But once the symmetry is broken, for example by rewiring a number of existing links, the graph structure becomes more complicated. And if the rewiring is done completely at random, then the regular lattice turns more and more into a random graph.

2.2. Random Graphs

According to the influential work by Erdős and Rényi [35], a simple undirected graph with n vertices and m edges is random if all of its $\binom{\binom{n}{2}}{m}$ possible instances have equal probabilities of being created. This is called the $G_{n,m}$ model. It is assumed that the presence or absence of an edge between two vertices has no influence on the presence or absence of any other edge. To generate such a graph, take n vertices and add edges between vertex pairs randomly chosen from the remaining pairs in the set $\binom{n}{2}$ until m edges have been added.

This is similar to the random graph model $G_{n,p}$ introduced by Solomonoff and Rapoport [36] and by Gilbert [37], in which every possible edge is added independently with probability p. There are $2^{n(n-1)/2}$ possible instances of the $G_{n,p}$ graph and it has $\left(\binom{n}{2}\right) p$ edges on average. The $G_{n,m}$ model is basically a snapshot of the $G_{n,p}$ model with exactly m edges. A slight variation of these models, in which self-edges – an edge that connects the vertex to itself – are allowed, is also commonly used.

The degree distributions of these random graphs are binomial. This means that the probability $P(k)$ of a vertex having k immediate neighbours is [38]:

$$P(k) = \binom{n-1}{k} p^k (1-p)^{n-1-k} \qquad k = 0, 1, 2, \ldots, n-1. \tag{2.1}$$

For large n, the binomial distribution is often approximated using the less cumbersome Poisson distribution:

$$P(k) = e^{-\lambda} \frac{\lambda^k}{k!} \qquad \lambda = (n-1)p \qquad k = 0, 1, 2, \ldots, n-1, \tag{2.2}$$

where λ is the mean degree. $G_{n,p}$ and $G_{n,m}$ random graphs are therefore often called "Poisson random graphs" [39].

Classical random graphs with their Poissonian distributions of connections are commonly referred to as "simple" networks, just like the regular lattices discussed in the previous section. In the strictest sense, every node in a "simple" network is indistinguishable from any other node regardless of the measured properties. A somewhat relaxed definition is to say that a graph is "simple" if all its nodes have similar property values for a given set of measurements [40]. Generally speaking, simple networks have no sophisticated topological features. Complex networks, on the other hand, have non-trivial topological features, like those found in networks of nature, society or technology [41]. As Newman puts it [39]: "Most of the interesting features of real-world networks [...] concern the ways in which networks are *not* like random graphs". The connections between nodes in a complex network are neither purely regular nor purely random. But what exactly defines a complex graph?

2.3. Graph Complexity

One definition of a complex graph is that its *excess* – defined as the number of edges m minus the number of vertices n – must be at least 1, which means that it has at least two cycles [42]. However, this definition seems too simplistic, as many random graphs and even the regular lattice based graphs would qualify.

Another way of looking at graph complexity is to analyse the amount of information needed to describe the graph structure. The more "choice" is involved in its formation, the less certain the outcome. The Shannon entropy [43] can be used to quantify this uncertainty. The Shannon entropy $H(X)$ of a set of possible outcomes X is defined as follows:

$$H(X) = -\sum_{i \in X} p_i \log p_i, \tag{2.3}$$

where p_i is the probability of event i occurring. The unit of measurement depends on the choice of the logarithmic base. If the logarithm is to the base two, then the information content is measured in bits. When a macroscopic system can be formed from a large class of microstates, then a high degree of "choice" or "freedom" is involved and it has a high entropy. If, on the other hand, only a small number of microstates satisfy the requirements of a particular macrostate, then there is not much "choice" and the entropy for this system is low [44]. Let X be the set of possible graph instances. In this case, the entropy is minimal (no uncertainty) for regular lattice graphs and maximal ($\forall i \in X, p_i = \frac{1}{|X|}$) for random graphs with Poissonian degree distributions. According to this definition, graphs are simple if their entropy is at either extreme and complex if it is somewhere in between the extremes.

The Shannon entropy is a general definition of the information content of an outcome [45]. Rashevsky [46], Trucco [47] and Mowshowitz [48] were among the first to apply entropy specifically to graphs. Here, X is a graph invariant (e.g. the number of vertices/edges or the vertex degree sequence) that is partitioned into k subsets using an equivalence relation α. The graph entropy $I(G, \alpha)$ can be defined as [49]:

$$I(G,\alpha) = -\sum_{i=1}^{k} \frac{|X_i|}{|X|} \log\left(\frac{|X_i|}{|X|}\right). \tag{2.4}$$

For example, Rashevsky [46] defines the topological information content of a graph G, where $|X_i|$ is the number of topologically equivalent vertices in the ith vertex orbit of G [49] and $|X| = |V|$ is the total number of vertices in G. Vertices are topologically equivalent if they are indistinguishable from one another. This is the case if they are of the same type, have the same number of incoming and outgoing connections, and their neighbours at a particular distance are indistinguishable from one another as well. More generally, two points, a and b, are equivalent ($a \sim b$) with respect to α if:

- $a \sim a$
- if $a \sim b$, then $b \sim a$
- if $a \sim b$ and $b \sim c$, then $a \sim c$.

Other graph entropy measures are based on metrical properties of graphs, like for example graph distances. Such measures have not only been defined to determine the global information content of a graph, but also the local entropy of subgraphs or individual vertices. The latter can be interpreted as a vertex complexity measure. The following definition represents the information distance of vertex $v_i \in V$ [49][1]:

$$I_D(v_i) = -\sum_{j=1}^{|V|} \frac{d(v_i,v_j)}{d(v_i)} \log\left(\frac{d(v_i,v_j)}{d(v_i)}\right), \tag{2.5}$$

where $d(v_i,v_j)$ is the distance from vertex v_i to vertex v_j and

$$d(v_i) = \sum_{j=1}^{|V|} d(v_i,v_j). \tag{2.6}$$

The entropy for graph G is defined as the sum of the local entropy measures:

$$I_D^{\star}(G) = \sum_{i=1}^{|V|} I_D(v_i). \tag{2.7}$$

The article by Dehmer and Mowshowitz [49] provides an extensive survey of the history of graph entropy measures. While they all provide some measure of the information content of a graph, the number of different structural features which contribute to the complexity of a graph is too large to capture in one unique definition. The choice of the best definition depends on the specific requirements for graph complexity.

Costa and Rodrigues [40] take a different approach to measuring the complexity of a graph. They look for the simple subgraphs of a given network using a set of local measurements that define homogeneity among vertices, namely the node degree, the clustering coefficient (see Equation 2.9 in Section 2.5), the average neighbouring degree (i.e. the average of the degrees of the immediate neighbours of a vertex i) and the locality index $loc(i) = N_{int}(i)/(N_{int}(i) + N_{ext}(i))$ of vertex i, where $N_{int}(i)$ is the number of links between the immediate neighbours of i including i itself and $N_{ext}(i)$ is the number of connections these nodes establish with the remainder of the network [50][2]. Thus, a vector with M measurements represents each vertex. The vectors for all vertices of graph G are projected into a two-dimensional space through principal component analysis (PCA) [52]. Then the probability density is estimated, which yields peaks for high concentrations of points in the 2D space. Finally, the nodes that are characterised by small dispersion of the measurements and that are also members of the same connected subgraph $G' \subseteq G$ are identified. Each

[1] Reported from the article (in Russian) by Konstantinova and Paleev "Sensitivity of topological indices of polycyclic graphs", Vychisl. Sistemy. 136 (1990).

[2] A modified version of [50] appeared later in [51].

of these clusters represents a class of statistically similar nodes. The number of peaks P in the probability density defines the number of different structures found in the network. The ratio of the cardinality of the largest cluster S to the network size N gives the *simplicity coefficient* in the range $(0,1]$ [40]. Networks with a high simplicity coefficient tend to be "simple" in terms of the measured properties and, conversely, networks with a low simplicity coefficient are more complex.

Nodes that are not part of a homogeneous cluster (i.e. the nodes with the smallest probability density values) are the most non-regular vertices in the graph. These nodes are called "singular" [51]. They have unique topological features and are therefore likely to play a special role in the network (e.g. hub nodes in a scale-free graph are singular in terms of node degree).

Many more graph complexity measures exist. For example, Kim and Wilhelm [53] define and compare various graph complexity measures based on the number of subgraphs found in a graph; "product measures", where the products of two quantities are zero in the extreme cases of a path or a fully connected graph and maximal for graphs with a medium number of links; as well as further entropy based complexity measures.

The number of existing graph complexity measures shows that there are many different ways to define complexity. As a general rule, very sparse graphs – like a path, ring or star – are not complex, because their topological features are trivial. And (nearly) fully connected graphs are not complex either, because all nodes have nearly the same degree and neighbourhood. Graphs with a medium number of links are complex if the connections form intricate topological structures [53]. What kinds of topological structures are analysed and considered as complex depends on the particular definition of graph complexity.

An example for a complex network is a graph that is entirely random except for the probability distribution of its vertex degrees [54]. And if this distribution follows a power-law, then the graph is called scale-free.

2.4. Scale-Free Graphs

When the probability of a graph property having a certain value varies as a power of that value then it is said to follow a power-law. For example, when the probability $P(k)$ of a node having k links follows $P(k) \sim k^{-\gamma}$, then the degree distribution of the graph follows a power-law. Such graphs are also called scale-free. The term comes from the scale invariance of power-laws, which is due to the fact that scaling k by a constant factor c only causes a proportionate scaling of the original function by $c^{-\gamma}$. A scale-free function therefore looks the same no matter on what scale one probes it [55]. The signature of a power-law distribution is the straight line on a log-log plot of k and $P(k)$.

It has been shown that for many real networks – including metabolic networks [56], scientific citation networks [57], actor collaboration networks [58], and the World Wide Web [9] – the exponent γ typically lies between 2 and 3. This phenomena of seemingly dissimilar systems exhibiting very similar critical exponents is called universality [41, 55]. The long tail of a power-law degree distribution rapidly ends at large degrees, which is called the cutoff point. The *natural cutoff* is highly model dependent [59].

Barabási and Albert [58] proposed a network model that incorporates two aspects they found to be fundamental to the development of real networks with power-law scaling. The first aspect is that real networks tend to grow and change throughout their lifetime. For example, the World Wide Web grows through the addition of new web pages, and neural (brain) networks grow through the addition of new neurons. The second aspect is that the connections in real networks do not form uniformly at random, but instead exhibit preferential attachment following certain rules. This leads to networks that follow the "rich get richer" principle, with few very highly connected nodes and many nodes with relatively low connectivity. This is also known as the Pareto principle or 80-20 rule [60].

The network model proposed in [58] starts with a small number of vertices n_0, the "core" of the network, which are connected in some fashion (at random, fully connected, etc.) using m_0 edges. At every time step t, a new vertex v_t is added to the network and connected to the existing vertices with $m \leq n_0$ edges. The neighbours of v_t are chosen so that the probability $P(k_i) = \frac{k_i}{\sum_{u \in V} k_u}$ of vertex v_t connecting to one of the existing vertices v_i depends on the connectivity k_i of v_i. The network gradually evolves into a scale-invariant state with $n_0 + t$ vertices and $m_0 + mt$ edges.

One of the likely reasons why complex networks with scale-free degree distributions are common in nature is that they are very robust against random failures of individual nodes or connections. The probability that a hub node – nodes with a degree much higher than the average degree – is affected by such an error is relatively low [61]. However, it has been shown [39, 41, 56, 60–63] that attacks that deliberately target hub nodes or bridges – nodes through which pass many shortest paths, also said to have a high "betweenness" [39] – have devastating effects on scale-free networks. The removal of a relatively small fraction of these highly connected nodes causes the network to fall apart into isolated clusters.

The preferential attachment does not always have to be by node degree alone. Many real networks possess localised structures, where two nodes are more likely to be connected if they have a neighbour in common. These kinds of networks are typically referred to as small-world networks.

2.5. The Small-World Phenomenon

"Given any two people in the world, person X and person Z, how many intermediate acquaintance links are needed before X and Z are connected?" [64] This question provoked the social psychologist Stanley Milgram to perform his famous small-world experiment (see Appendix A). He came to the unexpected conclusion that, on average, only 5 intermediaries are needed to deliver a message from one person to another, both living in the USA, even though every person in the chain is only allowed to forward the message to a personal acquaintance known on a first-name basis [3].

Although some doubts have been cast on the quantitative nature of Milgram's work, more recent work by Watts and Strogatz [66] and by Newman [67], among others, places the small-world concept on a firmly quantitative foundation. And while Milgram's results may not be absolutely accurate, they still show that a surprisingly small number of intermediaries is needed to connect two people considering the physical distance between the source and the target persons and the population of the USA.

The 5 links in the chain – or 6 steps in total – have given the widely known concept of the "Six Degrees of Separation" its name. This phrase, however, was not used until long after Milgram performed his studies. It is said [68] that it was popularised or even coined by John Guare in his 1990 play of the same name, where Ouisa, one of the characters, claims that "everybody on this planet is separated by only six other people. Six degrees of separation. Between us and everybody else on this planet" [69]. The play was adapted for the screen in the 1993 movie "Six Degrees of Separation".

Milgram was not the first one to ponder about the connectedness of people though. According to Barabási [68], the Hungarian poet and writer Frigyes Karinthy was the first one to propose that any two people on earth are connected through a very short chain of acquaintances in his 1929 short story "Láncszemek" ("Chains" or "Chain-Links"), which was published in the book "Minden masképpen van" ("Everything Is Different"). In "Láncszemek", Karinthy writes: "To demonstrate that people on Earth today are much closer than ever, a member of the group suggested a test. He offered a bet that we could name any person among earth's one and a half billion inhabitants and through at most five acquaintances, one of which he knew personally, he could link to the chosen one"[4].

[3] This section extends on work first published in [65] **A. Leist** and K. A. Hawick, "Circuits as a Classifier for Small-World Network Models," in *Proc. WORLDCOMP 2009 International Conference on Foundations of Computer Science (FSC 09)*, July 2009.

[4] See [70] for a translation of "Chain-Links".

Both Karinthy's claim and Milgram's studies are decades old. Since then many things have changed. Nowadays, due to relatively cheap air travel and new technologies like the World Wide Web, we are better connected than we were a few decades ago. This is why some authors claim that we could be much closer to 3 degrees of separation these days [68].

Milgram's experiment has inspired many scientists from different fields to analyse seemingly unrelated complex networks for properties characteristic to the network of human acquaintances. And they were successful. Biologists found them in metabolic networks [56, 71, 72] and neural networks [73]; sociologists in scientific collaboration networks [74, 75], the structures of criminal organisations [63] and many other types of social networks [76, 77]; and computer scientists found them in computer networks like the World Wide Web [9]. As diverse as these networks may appear to be, they can all be categorised as either small-world or scale-free networks, and some fit into both categories.

Even artificial networks that are specifically designed to mimic the structure of small-world networks have been developed following the interest in this type of network. For example, a peer-to-peer overlay network where a short route for messages sent between two nodes is determined based on local information only was suggested in [78] and experimentally tested in [79, 80]. The routing protocol in such a network functions very similar to the way messages were forwarded in Milgram's small-world experiment.

The attributes of scale-free networks have already been described in the previous section. But what does it take to be classified as a small-world network? Generally speaking, a small-world network combines the high clustering typical for a regular graph with the small mean geodesic distance of a random graph [66].

The first important property of small-world networks is therefore that the mean geodesic distance – also known as the mean shortest path between any two vertices – is small compared to the network size N. It must scale logarithmically or slower with N for fixed degree k [39]. The other property that clearly differentiates small-world networks from random graphs is the heightened clustering – also called network transitivity – which is caused by an increased number of short circuits, usually triangles, in the network. Triangles are sets of three distinct vertices, each of which is connected to the other two vertices. To meet both of these properties, small-world networks have localised, well connected communities as well as a relatively small number of long distance links or "shortcuts". These shortcuts are important both to keep the network from falling apart into many isolated clusters and to achieve the required scaling behaviour. The first such inter-community links have the biggest effect on the path length of the network [81, 82].

The clustering coefficient is a graph metric that is commonly used to measure the network transitivity. It was first introduced by Watts and Strogatz [66] and later redefined by Newman, Strogatz and Watts [54]. The former definition calculates the clustering coefficient C from the mean of the local clustering ratios computed for all vertices in a graph

$$C = \frac{1}{N}\sum_i C_i, \tag{2.8}$$

where C_i is defined as

$$C_i = \frac{\text{number of triangles connected to vertex } i}{\text{number of triples centered on vertex } i}. \tag{2.9}$$

Here a connected triple is a path of length two that connects three distinct vertices, or in other words a single vertex connected to an unordered pair of other vertices. The second definition of the clustering coefficient effectively calculates the fraction of triples that are part of a triangle such that

$$C = \frac{3 \times (\text{number of triangles in the network})}{\text{number of connected triples of vertices}}. \tag{2.10}$$

In this thesis, the second definition by Newman et. al. is used to compute the clustering coefficient unless specifically mentioned otherwise.

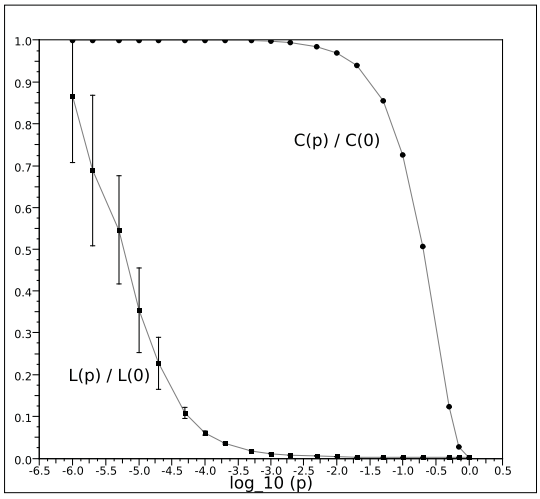


Figure 2.3.: The characteristic path length $L(p)$ and clustering coefficient $C(p)$ for Watts' small-world β-model with $n = 100,000$ vertices and degree $k = 20$. Each data point is averaged over 20 random realisations of the network model.

2.6. Small-World Network Models

The Watts-Strogatz small-world model [66], also referred to as the β-model [83], is a popular method for generating synthetic small-world graphs. The model starts of with a regular ring lattice of n vertices that are all connected to their k nearest neighbours (i.e. $k/2$ neighbours in either direction). Then, a vertex $i \in V$ is chosen and the edge $(v, v+1)$ connecting it to its nearest neighbour in a clockwise sense is rewired with probability p. When rewiring an edge, the old edge is deleted and a new edge (i, j) is created instead. The new neighbour j is picked uniformly at random from all vertices $j \in V, \forall j \notin A_i, i \neq j$, where A_i is the adjacency-list of vertex i (i.e. duplicate edges and self-connections are not allowed). This process is repeated for every vertex going around the ring in clockwise direction, until one lap has been completed. Then the whole process is repeated, considering the edge $(v, v+2)$ to the second-nearest neighbour of every vertex for rewiring, then the third-nearest and so on, until $k/2$ laps have been completed. At this point every edge has been considered for rewiring exactly once.

The β-model provides a simple way to create a graph structure that ranges anywhere from a "large-world" regular lattice with high clustering C and an average shortest path length L that scales linearly with the system size n, to a nearly random graph with small C and logarithmic length scaling by merely adjusting the rewiring parameter from $p = 0$ to $p = 1$. However, the most interesting graphs are generated for values of p somewhere in between regular lattice and random graph. It turns out that for a large range of p, the model generates small-world networks with $C(p) \gg C_{random}$ yet $L(p)$ almost as small as L_{random} [66]. The reason for this behaviour is that, as mentioned before, the first shortcuts have a highly nonlinear effect on L, drastically reducing the average path length for the entire graph. Larger values of p give diminishing returns for L. The clustering coefficient C, on the other hand, decreases much more slowly, because every rewired edge only has a local effect on the direct neighbourhood of the affected vertices. This behaviour is illustrated in Figure 2.3.

While increasingly large values of p eventually destroy the local community structure of the graph, moving it out of the region of disorder that classifies a small-world graph and turning it into a random graph, Barthélémy and Amaral [84] have shown that the small-world region can be extended to smaller and smaller values of p as long as the network size n is sufficiently large. The appearance of small-world

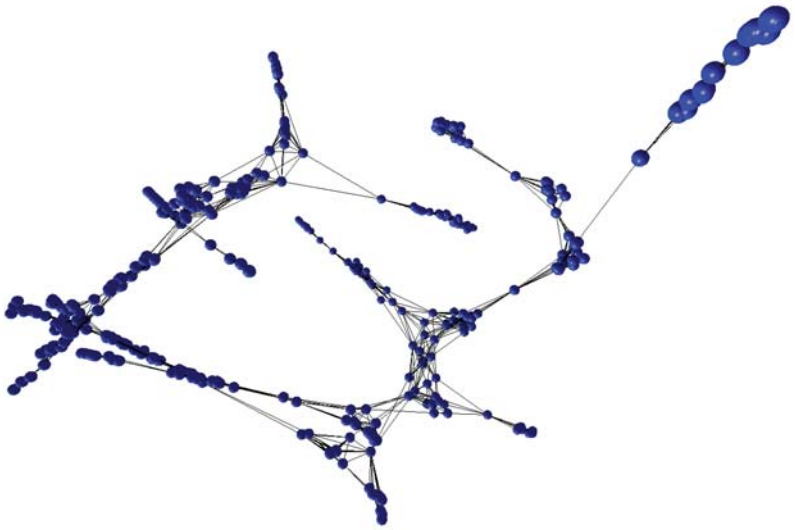


Figure 2.4.: Watts' small-world α-model with $n = 500$ vertices, degree $k = 10$ and $\alpha = 2.0$. The highly connected clusters of individual nodes are still clearly visible, but enough random shortcuts have been generated to combine all local communities into a fully connected graph. Visualised using the force-based layout algorithm in GraViz3D.

behaviour is a phase transition[5] that depends both on n and p. They propose the scaling law [84]:

$$L(n,p) \sim n^* F_k\left(\frac{n}{n^*}\right), \tag{2.11}$$

where F_k depends only on the degree k with $F_k(u \ll 1) \sim u$ and $F_k(u \gg 1) \sim \ln u$, n^* is a function of p such that $n^* = p^{-\tau}$ with $\tau = 1$ for $p \ll 1$ [82,85]. Hence, the crossover size from large-world to small-world behaviour is n^*. The mean shortest path length grows linearly for $n \ll \mathcal{O}(p^{-1})$ and logarithmically for large networks with $n \gg \mathcal{O}(p^{-1})$. This also holds for small-world networks built on lattices of dimension d greater than one [82].

The β-model can easily be extended to more dimensions d, iterating over all nearest neighbours in all dimensions before moving on to the second nearest neighbours and so on [83]. Because edges are rewired at random, it is possible for the graph to become disconnected, which is usually treated as infinite distance between two vertices in different disjoined components. This can be undesirable in some circumstances. To circumvent this problem, Newman and Watts [86] proposed a modified version of the model that does not rewire edges, but instead adds additional links between two randomly chosen vertices with probability p, one for each edge on the original lattice substrate. The number of shortcuts $npk/2$ thus remains the same on average, while the mean degree $k' = k + pk$ of the final graph increases with p. Furthermore, the modified model allows self-edges and multiple edges, so that the distribution of shortcuts is completely uniform.

Where there is a β-model, there must be an α-model as well. Watts' α-model [83] seems to have attracted somewhat less effort than the β-model, probably due to its higher computational demands and its more specialised focus on social networks. It was designed to construct a network in a fashion similar to how real social networks form, based on the currently existing network structure. Like the Barabási-Albert model [58] for scale-free graphs (see Section 2.4), the α-model thus generates small-world graphs using preferential attachment following a set of specific rules (see Appendix B). A single parameter α is used to

[5] See Newman and Watts [82] for a note on why this is a phase transition with varying rewiring probability p, contrary to what Barthélémy and Amaral [84] stated.

interpolate between a highly clustered but disconnected "caveman" world, where everybody is connected to everybody else in the same "cave" but to no one outside, and a random "Solaria"[6] world, where current friendships have almost no influence on the establishment of new friendships. An example for a graph generated with the α-model is given in Figure 2.4.

In the case of $\alpha = 0$ even a single shared neighbour drastically increases the propensity of two vertices to form a direct link between each other. Therefore, after the first edges have been created randomly, almost all following edges are created mainly based on the existing connections with very little chance of creating a random edge between two vertices that have no shared neighbours. The result is a disconnected "caveman" world. As α increases, existing links lose some of their influence on the network structure, allowing the "caves" to become connected by the increasing number of random shortcuts. When $\alpha \to \infty$ the network closely resembles a random graph, although it never becomes entirely random as the construction algorithm systematically loops over all vertices, which means that edges are not created in an entirely independent fashion as requested by the $G_{n,p}$ and $G_{n,m}$ random graph models described in Section 2.2. The model exhibits a phase change in its properties at a particular value of the α parameter. For example, the average shortest path length starts low for small α values and rises to a peak with increasing α, signalling the point where the network becomes fully connected. It then falls away to a flat fixed value at high α [83].

[6] Named after a planet in an Isaac Asimov novel (1957), where humans live in isolation and interact only via robots and computers [83].

Chapter 3

Parallel Processing Architectures

This chapter describes the compute architectures and parallel programming libraries used throughout this thesis. Features and restrictions that are specific to a particular processor design are explained.

3.1. Multi-Core CPU

While CPU manufacturers have traditionally increased the CPU frequencies from one generation to the next, this trend has slowed down dramatically over the last years, as the increasing power consumption becomes more and more difficult to manage. Manufacturers like Intel and AMD have instead started to incorporate more cores onto their chip designs, as illustrated in the CPU architecture diagram in Figure 3.1. Although the processing cores also integrate short vector units, typically 128-bit wide and more recently 256-bit with the new advanced vector extensions (AVX) in the latest generation of Intel processors, the CPU implementations described here do not take advantage of them beyond what the compiler can do automatically. These vector units only support floating point operations and are generally more limited than the SIMD units found in GPUs. The author concentrates on developing data parallel algorithms for the GPU and limits the CPU development efforts to task-level parallelism.

The consequence of the new processor designs for software developers and end users is that sequential software does not automatically run faster on newer CPUs. Programmers have to rethink and modify their software to use multiple threads so that parallel tasks can run concurrently on different CPU cores. Most programming languages either come with built-in support for multi-threading or allow external libraries

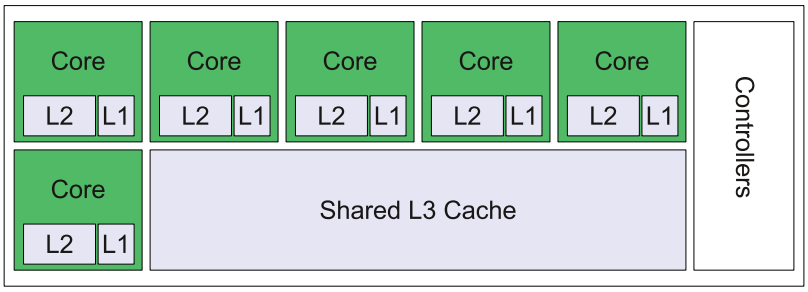


Figure 3.1.: A high-level view of a typical multi-core CPU architecture. Most modern desktop CPUs have four or six independent superscalar CPU cores with their own L1 and L2 caches and control logic. Some CPUs integrate a GPU on the chip at the cost of two main processor cores. A large, shared L3 cache provides low latency access to cached data.

to add such support. The POSIX Threads (PThreads) [87] multi-threading library is a low-level library commonly used in C/C++ programs. It enables the developer to create threads that are free to execute on different CPU cores. However, it is entirely up to the programmer to manage all concurrent threads and to ensure that they do not cause dead-locks or data race conditions. As it is relatively expensive to create new hardware threads, it should only be done if there is enough work available to amortise the overhead. Existing threads should be reused when possible for the same reason.

To increase programmer productivity and make multi-threaded programming more attractive to a larger group of developers, libraries have emerged that attempt to abstract some of the low-level tasks away. One of them is the Threading Building Blocks (TBB) template library [88] developed by Intel. It provides a set of library functions that hide most of the thread-level parallelism from the developer. Opportunities for concurrency are instead expressed in form of tasks that the TBB scheduler then automatically and efficiently maps onto available processor resources. Unless every thread does exactly the same amount of work and there are no other processes interfering, load balancing is an important task that the developer would have to do explicitly when using low-level threads. TBB offers many different constructs that are useful in particular situations, but for the algorithms discussed in this thesis, the author has found that a few of the most simple ones are often sufficient, namely the `tbb::parallel_for`, `tbb::parallel_for_each` and `tbb::parallel_reduce` functions.

The `tbb::parallel_for` function takes a range, the reference to a functor and an optional partitioner as parameters. Different range objects can be defined, for example for 1D, 2D or 3D value spaces. A simple 1D `tbb::blocked_range` object takes the lower and upper bounds of the iteration space and an optional grain size as its parameters. The grain size influences how the partitioner splits the iteration range, but its exact meaning depends on the implementation of the partitioner. The grain size is only a guide, it does not dictate the actual chunk size. The default `tbb::auto_partitioner` is generally a good choice, but Section 6.2.1 demonstrates a situation where a different partitioning strategy works better. TBB creates additional functor objects as it sees fit, assigns a sub-range to each object and calls it when processing resources are available. The `tbb::parallel_for_each` function offers a particularly simple way to iterate over a collection. It only takes two iterators, [first,last), and a functor to define the iteration range and tasks to be performed.

The `tbb::parallel_reduce` function computes a reduction over a given range. It works mostly like `tbb::parallel_for`, only that it expects an additional `join` method in the functor. This method takes another instance of the functor as its parameter and is expected to combine the results of the two objects in whatever way is suitable for the application. When the `tbb::parallel_reduce` function returns, the original functor object passed to it holds the answer of the parallel reduction.

Even though TBB adds an abstraction layer, the intelligent scheduling algorithms generally make up for the overhead. Section 6.2 compares PThreads and TBB implementations of the same algorithm and shows that both achieve comparable speed-ups over the sequential implementation.

3.2. Cell Broadband Engine

The Cell Broadband Engine (CellBE) is a multi-core CPU developed in a conjoint effort by IBM, Sony, Sony Computer Entertainment Inc. and Toshiba. It has been widely deployed as the processor in the PlayStation 3 (PS3) game console and in a number of server systems. The processor was first released in 2006 and represents a radical change to more traditional CPU architectures. Instead of a number of homogeneous processing elements, it consists of a single fully-featured 64-bit reduced instruction set computer (RISC) core, the PowerPC Processor Element (PPE), which can run an operating system. In addition to the PPE, the CellBE processor has eight Synergistic Processor Elements (SPEs), however one is disabled in the

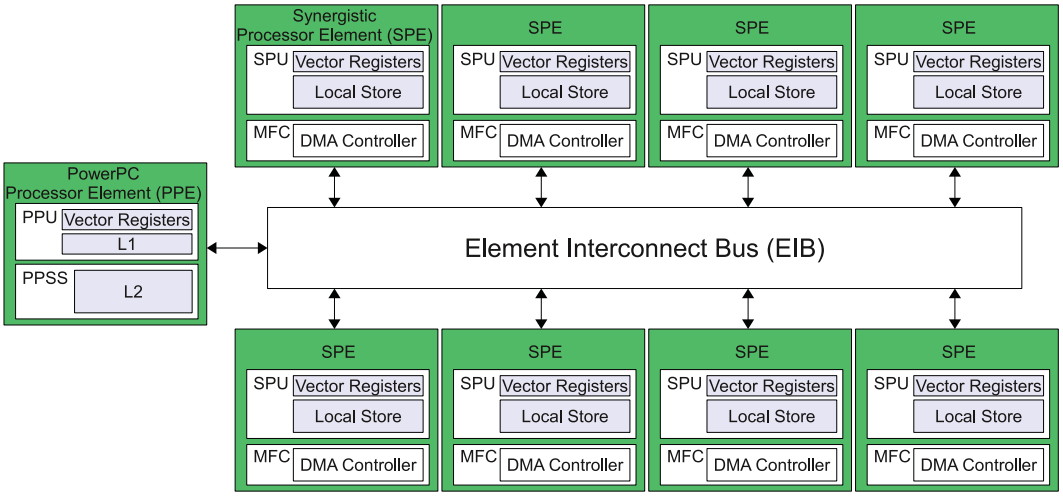


Figure 3.2.: A high-level view of the architecture of the CellBE. Although the processor in the PlayStation 3 contains eight SPEs, only six of them are available to the developer. The PPE is the main processor that runs the operating system and manages system resources. The SPEs are compute accelerators with a SIMD architecture and large register files. SPUs issue DMA commands to their MFC, which asynchronously transfers data between system memory and local stores.

processor that ships in the PS3 and another one is reserved for the operating system, which leaves six SPEs that are available to the application developer.

The SPEs are optimised for compute-intensive applications and come with extensive SIMD functionality. The Memory Flow Controller (MFC) contains a direct memory access (DMA) controller that is used to asynchronously transfer data between main memory and the private local store of the Synergistic Processor Unit (SPU), which is the compute unit of the SPE. All other load and store instructions issued by the SPU only access the local store. Several DMA memory transfers can be executed in parallel with computation, changing the focus from speculative memory fetches for low latency cache hits to explicit memory transfers with latency hiding. The MFC maintains and processes queues of up to 16 DMA requests. Requests to the MFC must be naturally aligned and the transfer size can be either 1, 2, 4, or 8 bytes or a multiple of 16 bytes up to a maximum of 16 KB. Up to to 2,048 DMA commands can be grouped into a DMA-list.

Figure 3.2 illustrates the architecture [89] of the CellBE. The PPE is divided into the PowerPC Processor Unit (PPU) and and the PowerPC Processor Storage Subsystem (PPSS), which contains the 512 KB unified L2 cache. The PPU comes with an integer arithmetic logic unit (ALU) and a floating point unit (FPU), 32 vector registers, each 128-bit wide, 32 KB L1 instruction and 32 KB L1 data caches and various control units. Every SPU contains 128 vector registers, each 128-bit wide, a unified 256 KB local store, two ALUs, one FPU as well as various controllers. The PPE communicates with the SPE through mailboxes, signal notification registers and direct memory. Mailboxes, which are the main communication mechanism used here, are queues that can be used to exchange 32-bit messages. Software development is supported in C/C++. The provided libraries offer a set of intrinsic functions for SIMD operations.

3.3. Graphics Processing Units

NVIDIA has put a lot of effort into promoting the use of GPUs as general purpose compute accelerators. Their CUDA toolkit introduces extensions to C++ that enable developers to tap into the processing power of the GPU while using well established programming tools and skills. The GPU code is written in the form of so called device *kernels* – routines that execute on the GPU – which are set up and managed by

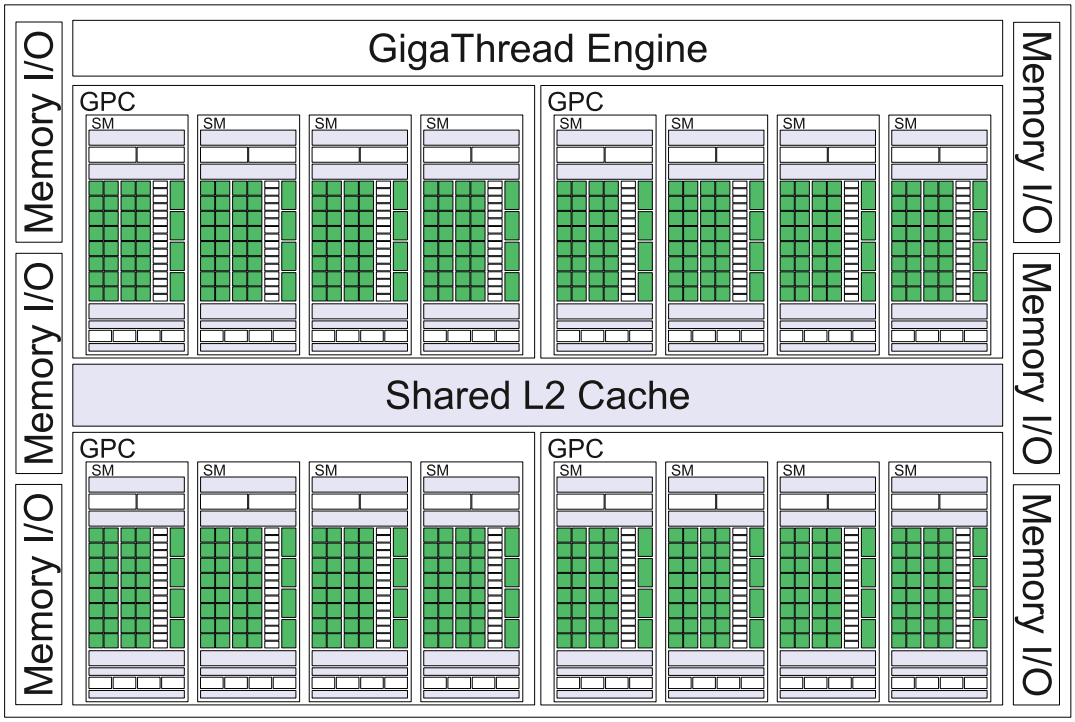


Figure 3.3.: A high-level view of the “Fermi” GPU architecture.

the host program that is running on the CPU. Device kernels run asynchronously, so that the CPU is free to perform other tasks while the GPU is busy. But before one can start writing applications that perform well on the graphics device, it is important to understand how the hardware works. Although this thesis focuses on CUDA hardware, graphics devices from other manufacturers support many of the same features. The general approaches to developing data parallel applications are thus transferable from CUDA to other GPGPU platforms.

3.3.1. The GPU Architecture

Figure 3.3 gives a high-level view of the “Fermi” GPU architecture that is used in the latest generation of NVIDIA devices. It extends the GT200-series of devices with additional features, but the fundamental design remains the same as far as software developers are concerned. The most significant design change is the addition of the L2 and L1 caches, which is further discussed below. However, to maximise performance on a particular device, a number of additional differences – like the precision of the ALU for multiply operations and related intrinsic functions – need to be considered that are beyond the scope of this discussion. See the CUDA programming guide [90] for detailed information. NVIDIA assigns a compute capability (CC) to each of their devices, which determines the core architecture and thus the supported hardware features. The devices used to test the algorithms that are proposed in this thesis are based on the GT200-series with CC 1.3 and the Fermi-architecture with CC 2.0. The CUDA toolkit version 3.2 is used [90]. This section gives a brief overview of the Fermi-architecture, specifically for devices with CC 2.0. More detailed information can be found in [90, 91].

The architecture diagram illustrates the large number of processor elements – 512 CUDA cores plus an additional 64 special function units (SFUs) in total – found on a fully-populated Fermi-device like the GeForce GTX580. The graphics chip is organised into Graphics Processor Clusters (GPCs) with four

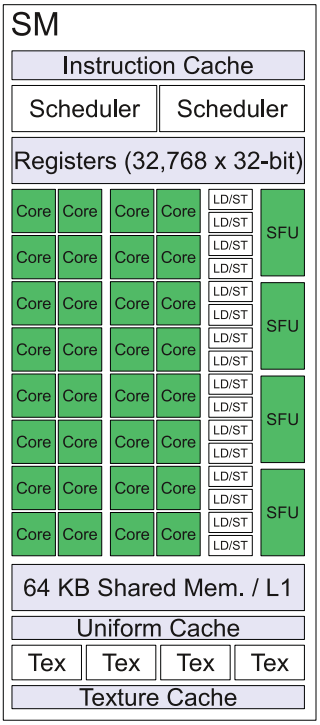


Figure 3.4.: A detailed view of the relevant features of a “Fermi” Streaming Multiprocessor.

Streaming Multiprocessors (SMs) each. A unified 768 KB L2 cache is used to service all device memory load and store requests. DRAM is accessed through a 384-bit memory interface with a maximum bandwidth of 192.4 GB/sec in the top of the line configuration. The professional Tesla line of devices even supports ECC memory, a feature that is particularly important for the acceptance of GPUs as accelerators for scientific applications. The GigaThread engine is the global scheduler that issues blocks of threads to individual SMs. Starting with CC 2.0, the GigaThread engine can process up to 16 concurrent kernels, whereas previous generations only support a single device kernel at any given time.

All SMs operate independently from each other. A single SM contains 32 CUDA cores – each with its own ALU and FPU – and four SFUs. Figure 3.4 provides a more detailed view of the SM architecture. Hardware features that are specific to graphics applications and not usable by GPGPU programs are omitted. The thread blocks issued to the SM are divided into groups of 32 sequential threads, which are called warps. When processing 32-bit instructions, each of the two schedulers and instruction dispatch units issues one instruction to a different warp per clock cycle. Half the threads from each warp are executed concurrently on the first or second of the two 16-element SIMD units respectively. When processing 64-bit instructions, then only one scheduler can issue an instruction and the 16 threads of a half-warp run on all 32 cores. The four SFUs execute single-precision transcendental instructions (sine, cosine, reciprocal, and square root) and can operate concurrently with the CUDA cores as long as no double-precision instructions are issued. Each SM has 16 load/store units that can calculate source and destination addresses for 16 threads per cycle.

Up to 48 warps from at most 8 thread blocks can be resident on each multiprocessor. The thread block size is defined by the developer when launching the kernel on the device, but it is limited to at most 1,024 threads per block. The actual number of resident warps is further limited by the register usage of the kernel, as the execution context for every resident warp is maintained on-chip. Switching from one execution context to another has no cost. Thread schedulers make use of this by issuing instructions to warps that are ready to proceed while other warps are waiting for data. This effectively hides memory latencies, which

is an important feature, because CUDA devices are optimised for high bandwidth and not for low latency when accessing device memory.

Device memory is generally referred to as global memory. All threads have shared access to global memory and it is persistent between kernel calls as long as the CUDA context exists. It is also the only memory that can be accessed from the host to copy data to or from the device. Devices with CC 2.x access global memory through L2 cache. A cache line is 128 bytes long and maps to a naturally aligned 128-byte segment in global memory. In addition to the shared L2 cache, each multiprocessor offers a number of fast on-chip memories to facilitate data reuse and information exchange among threads in the same thread block. On-chip memory should be used whenever possible to reduce global memory transactions. Each SM offers the following caches:

- 64 KB of memory that can be configured for either 48 KB of shared memory and 16 KB of L1 cache or vice versa. L1 cache has 128-byte cache lines that map to a 128-byte segment in L2 cache. L1 cache is also used for register spills. This is a significant improvement compared to pre-Fermi devices, which do not have the L1 and L2 cache and spill registers to device memory. Shared memory is an explicit cache that can be used to store data for repeated use or to exchange information between threads in the same thread block. Shared memory is organised into 32 memory modules, each with a bandwidth of 32 bits per two clock cycles.
- Although constant memory is located in device memory, every SM has 8 KB of constant cache. A device memory transaction is thus only performed on a cache miss, greatly improving access times when several threads access the same values. Constant memory is up to 64 KB large and read-only.
- Texture memory is another read-only access path to global memory that is optimised for 2D spatial locality. Each SM has 6-8 KB of texture cache. The texture units can perform certain filter operations and the user can define whether texture coordinates are normalised or not. While texture references can only be used to read data, the values can be updated through device memory pointers to the same data or via surface references if the texture is bound to a CUDA array. However, texture fetches are not kept coherent with respect to these writes and undefined data is returned when reading from an address that has already been written to in the same kernel call.

3.3.2. The CUDA Programming Model

When launching a CUDA kernel, the developer has to specify the execution configuration. This includes at least the grid size and the thread block size. Thread blocks are arranged in a one or two-dimensional grid with a maximum dimension length of 65,535. Each block arranges up to 1,024 threads (512 for devices with CC$<$ 2.0) in one, two or three-dimensions, with a maximum x- or y-dimension length of 1,024 and a maximum z-dimension length of 64. The block size should always be a multiple of 32 (the warp size). CUDA threads are managed in hardware and there is very little thread creation overhead. CUDA is optimised for an extremely fine-grained level of parallelism and can easily manage hundreds of thousands and even millions of threads, although not all threads are active at the same time. The number of active threads is limited by the resident threads per multiprocessor discussed in the previous section.

The first optional parameter to the execution configuration defines the amount of dynamically allocated shared memory per block. This is in addition to statically allocated shared memory. The second optional parameter specifies a CUDA stream. By default, all kernel calls and memory transfers initiated by the host are in stream zero. Kernel launches and asynchronous memory functions in different streams can overlap when the hardware supports this feature. The host can use library functions to synchronise different streams. A kernel function is defined as follows:

```
__global__ void MyKernel(int* d_in, int* d_out);
```

and the configuration is passed to the CUDA library when calling this kernel:

```
MyKernel<<<gridSize, blockSize, shrdMem, stream>>>(d_in, d_out);
```

Other device functions, which are annotated with `__device__`, can be called from the `__global__` function. The kernel is executed by every thread defined in the launch configuration. When comparing it to a sequential program, then the kernel is essentially the body of a loop, but instead of doing say $1,000,000$ loop iterations, $1,000,000$ different threads are used to perform one iteration each. Device code supports a subset of C++. Intrinsic functions are used to access hardware features. A particularly important function is `__syncthreads()`, which synchronises the threads in the same block. This is mostly used to coordinate access to shared memory. To synchronise globally, the kernel call has to return and a new kernel has to be started.

Code is expressed in terms of scalar threads that can take data-dependent conditional branches. But the scheduler always issues an instruction to a warp and not to individual threads. When all threads in a warp agree on an execution path, then only that branch is executed and the only overhead comes from evaluating the branch-condition. This overhead is small. However, if the threads in the same warp take different execution paths, then CUDA automatically serialises both branches and executes them in sequence. Threads are enabled and disabled as necessary so that they are only affected by the instructions from the branch that they are supposed to take. NVIDIA refers to this model as single instruction, multiple thread (SIMT) to emphasise the scalar behaviour of threads. This differs from regular vector SIMD units, where special instructions need to be used to mask unwanted results whenever there is a conditional statement. But nevertheless, warp divergence should be avoided whenever possible, especially when the conditional branches contain a large number of instructions. Divergence only occurs within a warp, different warps execute independently from each other.

Device memory is allocated by the host before the kernel is launched and then passed to the global function as illustrated above. Global memory instructions can access naturally aligned words of size equal to 1,2,4,8, or 16 bytes. Naturally aligned means that the memory is aligned to a multiple of the word size. Device memory transactions are 32 or 128 bytes wide and always naturally aligned, however, 32-byte transactions are only supported when L1 caching is disabled. When all the threads in a warp access memory addresses in the same segment of device memory, then the requests are coalesced into one, two, or four transactions, depending on the word size. For example, when all 32 threads in a warp access sequential 4-byte values in global memory, then a single 128-byte transaction can service all requests. However, if all threads in a warp access random memory addresses, then up to 32 independent transactions are executed, each 128 bytes wide if L1 caching is enabled, which significantly reduces the performance of the kernel. Arranging data in global memory in such a way that it can be accessed via coalesced transactions is among the most important optimisations for a CUDA program.

As mentioned in the previous section, shared memory is arranged into 32 memory modules, called banks, each with a bandwidth of 32 bits per two clock cycles. The banks are organised such that successive 32-bit words are assigned to successive banks. The threads in a warp can simultaneously access different memory banks, but if any two threads address different 32-bit words that are located in the same module, then there is a bank conflict and those requests have to be serialised and executed over multiple clock cycles. If multiple threads read bytes from the same 32-bit word, then the value is broadcast and no bank conflict occurs. When writing to the same word, then each byte is written by only one thread. It is important to minimise bank conflicts by using a suitable address stride.

More specific CUDA features then those discussed here are explained when needed in the main text.

3.4. Test Environment

Unless specifically mentioned otherwise, the following systems are used to run the performance experiments presented in this thesis:

CPU Both single-threaded and multi-threaded CPU implementations are tested on an Intel Core i7 970 with six physical cores – or 12 logical cores with Intel Hyper-Threading Technology – running at 3.2 GHz. Level 3 cache is 12 MB. The machine has 24 GB of DDR3 2000 MHz main memory. The operating system is Ubuntu Linux 10.10. C/C++ code is compiled with GCC 4.5 and TBB version 3.0 is used for multi-threaded TBB code.

GTX580 To represent the Fermi-architecture, the CUDA implementations are tested on NVIDIA GeForce GTX580 GPUs with 1536 MB of GDDR5 device memory and 512 CUDA cores running at 1.54-1.59 GHz. Three of these GPUs are installed in the same system, each using its own PCI Express (PCIe) $\times 16$ slot. The CPU is an Intel Core i7 970 like the one specified above. The machine has 24 GB of DDR3 1600 MHz main memory. The operating system is Ubuntu Linux 10.10 and CUDA code is compiled with the NVIDIA CUDA Compiler Driver (NVCC) release 3.2 and GCC 4.4.

GTX260 To represent the GT200-series of CUDA devices, a system with two NVIDIA GeForce GTX260 GPUs with 896 MB of device memory and 216 CUDA cores running at 1.4 GHz is used. Each GPU is installed in its own PCIe $\times 16$ slot. The CPU is an Intel Core 2 Quad Q8200 CPU at 2.33 GHz and 8 GB of main memory. The operating system is Ubuntu Linux 10.10 and CUDA code is compiled with NVCC 3.2 and GCC 4.4.

CellBE A PlayStation 3 is used for the CellBE experiments. The processor runs at 3.2 GHz and 6 SPEs are usable in addition to the PPE. The device has 256 MB of main memory. The operating system is Yellow Dog Linux 6.1. The Cell software development kit (SDK) version 3.1 is used for the code.

The Intel Core i7 970 and the NVIDIA GeForce GTX580 have been chosen for the performance comparisons as they were among the top processors available in their respective device categories and retailed for a comparable price at the time of purchase. The Core i7 970 CPUs can automatically overclock their cores to up to 3.46 GHz using the Intel Turbo Boost Technology. The maximum frequency depends on the number of active cores, the operating temperature and power consumption. This has to be taken into account when comparing the performance of a sequential CPU algorithm to that of a multi-threaded implementation, as it is likely that the sequential code is being executed at a slightly higher clock rate than the parallel algorithm.

Part II.

Parallel Graph Algorithms

Chapter 4

Parallel Random Number Generation

Random numbers are essential to many computer simulations[1]. Non-deterministic algorithms require a level of uncertainty or chance, for example to model external influences on the system. While quantum random number generators (RNGs) that create a "truly" random, irreproducible sequence of deviates exist [94], repeatability is important at least in the testing stages of a program or when it may be necessary to reproduce the conditions under which a certain event occurs. RNGs that are based on a physical phenomenon also require specialised hardware and tend to be slower [95] than pseudo-random number generators, which use a computational algorithm to generate a seemingly random sequence of deviates.

While a pseudo-random number generator is never truly random, a high quality algorithm produces a sequence of deviates that is sufficiently decorrelated for most problems. A related issue is the period length of the algorithm, that is the number of deviates it can generate before the sequence restarts from the beginning. The period must be long enough that even random number heavy simulations that take months on today's supercomputers do not exhaust it. A list of desirable properties that distinguish a good RNG from a bad one may include [31]:

- Randomness: The values in the generated sequence must be independent and random enough for all reasonable applications. A statistical test-suite like the "Diehard" battery of tests [96] or the more recent tests suggested in [97] are usually used to assess the quality of RNG algorithms.
- Long period: The generated sequence must be long enough that it does not repeat in a realistic time frame on any computer in the foreseeable future.
- Efficiency: Rapid execution with modest memory requirements.
- Repeatability: The generated sequence of random numbers is completely determined by the initial conditions (seed values).
- Homogeneity: All bits in the deviate must be equally random.
- Portability: Generates the same sequence of random numbers on a wide variety of computers given identical starting conditions.

RNGs typically produce a sequence of random uniform deviates, either b-bit integers in the range $[0, 2^b)$ or floating point numbers in the range $[0.0, 1.0)$. A number of transformation algorithms [98, 99] can take these uniform deviates as input and generate other statistically important distributions.

[1] This chapter extends on work first published in [92] K. A. Hawick, **A. Leist**, and D. P. Playne, "Mixing Multi-Core CPUs and GPUs for Scientific Simulation Software," *Research Letters in the Information and Mathematical Sciences*, vol. 14, no. ISSN 1175-2777, pp. 25–77, 2010 and in [93] K. A. Hawick, **A. Leist**, D. P. Playne, and M. J. Johnson, "Speed and Portability issues for Random Number Generation on Graphical Processing Units with CUDA and other Processing Accelerators," in *Proc. Australasian Computer Science Conference (ACSC 2011)*, 2011.

Algorithm 1 Initialisation of Marsaglia's Uniform Random Number Generator.

function initialise(*seed*)
  $r \leftarrow 97$ //the first lag value
  $s \leftarrow 33$ //the second lag value
  $u[r]$ //lag table of length r
  $r_{idx} \leftarrow r-1$
  $s_{idx} \leftarrow s-1$
  $c \leftarrow 362436/16777216$
  $d \leftarrow 7654321/16777216$
  $m \leftarrow 16777213/16777216$
  initialise($u, seed$) //initialise lag table u

Algorithm 2 Each call to the `uniform()` method generates a random number in the range $[0.0, 1.0)$.

function uniform($r_{idx}, s_{idx}, c, d, m, u$)
  $result \leftarrow u[r_{idx}] - u[s_{idx}]$
  if $result < 0$ **then** $result \leftarrow result + 1$ **end if**
  $u[r_{idx}] \leftarrow result$
  $r_{idx} \leftarrow r_{idx} - 1$
  if $r_{idx} < 0$ **then** $r_{idx} \leftarrow r-1$ **end if**
  $s_{idx} \leftarrow s_{idx} - 1$
  if $s_{idx} < 0$ **then** $s_{idx} \leftarrow r-1$ **end if**
  $c \leftarrow c - d$
  if $c < 0$ **then** $c \leftarrow c + m$ **end if**
  $result \leftarrow result - c$
  if $result < 0$ **then** $result \leftarrow result + 1$ **end if**
  return $result$

A number of different RNG algorithms are widely used [100], ranging from fast but low quality, such as linear congruential generators, to slower but high quality algorithms, like the 64-bit Mersenne-Twistor [101]. Generating good quality fast random numbers remains a challenge [102–104]. Many scientific simulations, in particular Monte-Carlo algorithms like the Ising model discussed in Chapter 7, rely heavily on reliable and high-performance RNGs. In fact, the Ising model and other Monte Carlo algorithms can be used themselves as demanding tests for the quality of random numbers, based on comparison with known results from analytical studies or extensive numerical analysis [105,106]. As the compute architectures used to tackle these kinds of problems become increasingly parallel, the need for random number generators that are optimised for such hardware increases as well [107–111].

This chapter describes two CUDA implementations of Marsaglia's lagged-Fibonacci generator [31], a 24-bit algorithm that fulfils all of the listed requirements and that has been tested in Monte Carlo work [106, 112]. The fact that it is a 24-bit algorithm is convenient as the types of devices used for most of the work described in this thesis, namely graphics processing units, do not always support 64-bit precision or use considerably more clock cycles to perform a 64-bit operation than to perform a 32-bit operation.

4.1. The Sequential CPU Implementation

A detailed description and a sequential implementation of the lagged-Fibonacci random number generator have been given by Marsaglia et. al. [31]. This section only gives a brief summary of the algorithm that is needed for the understanding of the following discussion on using the RNG for simulations on parallel architectures. The suggested RNG algorithm is actually a combination of a lagged-Fibonacci generator $F(r,s,\odot)$ and an arithmetic sequence for the prime modulus $2^{24}-3$. The pseudo-code in Algorithm 1 defines a number of variables and constants needed for the two generators. The procedure used to fill the initial lag table u with suitable deviates is not relevant for the discussion here, but is discussed in detail in [31].

Function `uniform()` in Algorithm 2 describes the process of generating a uniform random number in the range $[0,1)$. The first part shows the implementation of the lagged-Fibonacci generator. The lags r_{idx} and s_{idx} are the indices into the lag table[2], which is shown here of length 97 but can be any suitable prime. It provides a sequence of real deviates $x_1, x_2, x_3, \ldots$, with $x_n = x_{n-r} \odot x_{n-s}$, where the binary operation $x \odot y$ for reals x and y is defined as $x \odot y = \{\text{if } x \geq y \text{ then } x-y, \text{ else } x-y+1\}$. The period of the lagged-Fibonacci generator is $(2^r-1)2^{b-1}$ [113], where b is the precision in bits. Using the given values, the period of the generator is $(2^{97}-1)2^{23}$ or about 2^{120}.

[2]The lag table is indexed from $0, 1, 2, \ldots, r-1$.

Algorithm 3 Initialising the TBB implementation of Marsaglia's random number generator. The parameters to the function are the seed s_0 and the desired number of RNG tasks T.

function initialise-tbb(s_0, t)
  allocate V //vector used to store the RNG structures
  $r_0 \leftarrow$ new RngInstance //the root RNG
  initialise(r_0, s_0)
  for $i \leftarrow 1$ to T **do**
    $r_i \leftarrow$ new RngInstance
    $s_i \leftarrow r_0$.uniform() $*$ INT_MAX //generate the seed for r_i
    initialise(r_i, s_i)
    append r_i at the end of vector V
  end for
  return V

To further improve the quality of the generated random numbers, Marsaglia combines the lagged-Fibonacci generator with a simple arithmetic sequence for the prime modulus $2^{24} - 3$, which performs the binary operation $c \odot d = \{\text{if } c \geq d \text{ then } c - d, \text{ else } c - d + m\}$. The two generators are combined by means of the same binary operation used for the lagged-Fibonacci generator. The second generator has a period of 2^{24} [106], increasing the period of the combined generator to approximately 2^{144} for the given configuration [31].

4.2. The Multi-Threaded CPU Implementation

The straight forward approach to using Marsaglia's random number generator in a multi-threaded application is to use an independent RNG instance for each thread. A single root RNG can be used to generate the additional seed values needed to initialise the thread-specific RNG instances. If each thread executes code that consumes random numbers, then they can simply generate their own deviates independently from all other threads. If, on the other hand, the algorithm consuming the random numbers is sequential or at least does not fully utilise all available CPU cores and requires a large amount of random numbers, then the RNG instances can be used to quickly fill an array with deviates for later consumption by the main algorithm. It is important that the number of RNG instances and the region of this array filled by each one of those instances is kept consistent if the main algorithm is to be repeatable.

Here the TBB multi-tasking library is used to parallelise the lagged-Fibonacci generator. As shown in [92, 114], it can achieve comparable performance to low-level multi-threaded algorithms using POSIX threads. This is further discussed in Chapter 6.2. The RNG instances are not associated with a particular hardware thread, as TBB emphasises the use of light-weight tasks instead of full-blown threads. Therefore, each instance is contained in a structure that can also store additional, application specific information related to the RNG instance. For example, it may contain a pointer to an array that temporarily stores the generated deviates for later use, along with the array length. The structures are pushed into a vector after their RNG instances have been initialised. See Algorithm 3 for a description of this initialisation process.

The parallel random number generation using these RNG instances is invoked by passing the begin and end iterators of the vector to TBB's `parallel_for_each` function, along with a pointer to a function that takes the structure type as its only argument. TBB applies the given function to the results of dereferencing every iterator in the range [begin,end). This is the parallel variant of `std:for_each`. The invoked function can then use the RNG instance passed to it to fill the array range specified in the same structure or to immediately use the random numbers in the application specific context. The process remains repeatable even though the thread that executes the given task can change every time `parallel_for_each` is called.

TBB's task scheduler decides how many hardware threads are used and how they are mapped to the given tasks. This takes the burden of having to do load-balancing off the developer, who merely has to ensure

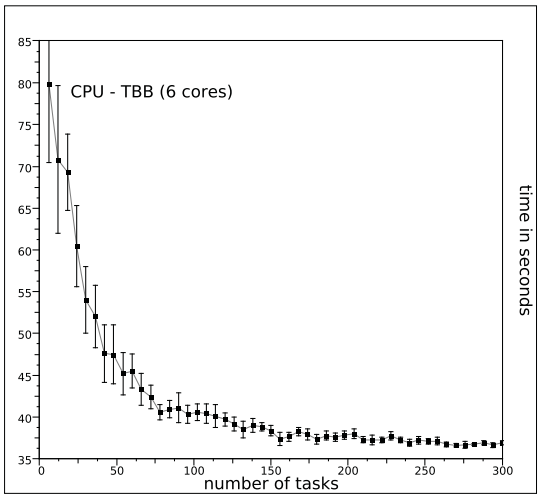


Figure 4.1.: This plot shows the time taken to generate a total of 30 billion random numbers and how it varies with the number of TBB tasks and associated RNG instances. The number of tasks is increased in multiples of 6, the number of physical cores in the processor. Each data point is averaged over 10 measurements.

that the number of RNG instances is large enough to let the task scheduler do its work. This is important as the number of tasks affects the performance considerably as illustrated in Figure 4.1. The processor used for these measurements is an Intel i7 970 with 6 physical and 12 logical cores. The results show that the performance suffers when the number of tasks is too small. This is different from an implementation using PThreads, where the performance does not change significantly when using more threads than logical cores [93] – assuming that no other processes steal cycles and thus interfere with the load-balancing – or even suffers from the overhead of creating more heavy-weight threads than necessary.

4.3. The CUDA GPU Implementation

The first CUDA implementation of the lagged-Fibonacci random number generator is based on generating a separate stream of random numbers with every CUDA thread. This approach, referred to as CUDA 1, is repeatable and fast as race conditions are avoided and no communication between threads is required. Algorithm 4 shows the host code used to prepare the CUDA RNG instances. A relatively small lag table should be used due to the memory requirements of this approach. The code example uses a table length of 97, which means 388-bytes just for the table and 400-bytes total including the other variables per thread. Larger lag tables can be used to further improve the period at the expense of memory usage. The input seed value is used to initialise a RNG instance on the host, which is then used to generate the seeds for the CUDA threads.

The CUDA implementations of the lag table initialisation and uniform RNG functions are essentially the same as on the CPU, only that ternary expressions, which can be optimised by the compiler, are used to avoid branches and array indexing is adapted so that global memory accesses can be coalesced as long as the threads of a half-warp always request a new random number at the same time. This is illustrated in Algorithm 5. It also demonstrates a CUDA kernel that generates a given amount of random numbers per thread and writes them to an array in global memory. This kernel is used by many algorithms discussed throughout this thesis. Some algorithms, however, produce their random numbers directly in the consuming kernel, thus avoiding the need to write them to array *rnd* in global memory first. These algorithms call

Algorithm 4 CUDA implementation of Marsaglia's RNG that produces T independent streams of random numbers, where T is the number of threads. This is the host code.

$L \leftarrow 97$ // lag table length
function RNG1(s)
  Input parameters: s is the RNG seed.
  initialise host RNG with seed s
  $X \leftarrow$ generate T random deviates on the host //seed values for the device RNGs
  allocate $X_d[T]$ in device memory
  copy $X_d \leftarrow X$
  allocate $U_d[TL]$ in device memory //T lag tables
  allocate $C_d[T], R_d[T], S_d[T]$ in device memory //the variables c, r, s needed by the RNG, for each thread
  do in parallel on the device using T threads: call INIT_KERNEL(X_d, U_d, C_d, R_d, S_d)
  //RNG instances are initialised, call RNG_KERNEL as needed
  do in parallel on the device using T threads: call RNG_KERNEL($count, U_d, C_d, R_d, S_d$)

Algorithm 5 The device code used to generate *count* deviates per thread and write them to array *rnd*. T is the thread count, $cd = 7654321/16777216$ and $cm = 16777213.0/16777216.0$.

```
__global__
void rng_kernel(Uint count, float* U, float* C, int* R, int* S, float* rnd) {
  Uint tid = (blockIdx.y * gridDim.x + blockIdx.x) * blockDim.x + threadIdx.x;
  float c = C[tid];
  int r = R[tid];
  int s = S[tid];
  for (Uint i = 0; i < count; ++i) {
    rnd[i*T+tid] = uniform(tid, U, c, r, s);
  }
  C[tid] = c;
  R[tid] = r;
  S[tid] = s;
}

__device__
float uniform(Uint tid, float* U, float& c, int& r, int& s) {
  float result = U[T * r + tid] - U[T * s + tid];
  result = (result < 0.0F) ? result+1.0F : result;
  u[T * r + tid] = result;
  r = (r == 0) ? TABLE_LENGTH-1 : r-1;
  s = (s == 0) ? TABLE_LENGTH-1 : s-1;
  c = c - cd;
  c = (c < 0.0F) ? c+cm : c;
  result = result - c;
  result = (result < 0.0F) ? result+1.0F : result;
  return result ;
}
```

the `uniform()` method directly. While the second approach is faster under certain circumstances, it is not always feasible because the kernel consuming the random numbers may be executed by so many threads that the memory cost of having a separate RNG instance per thread would be too high. The cost of having to temporarily store the random numbers to global memory may also be lower than the cost of having to load and save parameters c, r, s from and to global memory, which only needs to be done once per kernel call and thread regardless of the number of deviates generated. This is especially the case if the consuming kernel only generates a single random number per thread, but `rng_kernel` can be used to generate multiple deviates per thread. Another factor that needs to be considered is the additional number of registers needed for the RNG instances. If the consuming kernel already uses a large amount of registers, then those additional registers may further limit the number of active threads on the multiprocessor and consequently the amount of memory latency that the thread scheduler can hide by switching between threads. As can be seen

the optimal choice depends on a number of factors. If both options are viable, then it is usually best to test which one performs best in the given scenario.

If the random numbers are generated as part of another kernel, then the number of threads T usually depends on the algorithm that consumes the deviates. However if a dedicated RNG kernel is used to generate the random numbers and write them to global memory for later use, then the optimal number of threads depends on the graphics hardware. It should be large enough to keep the device busy and divide evenly by $(B \times M)$, where B is the thread block size and M the number of streaming multiprocessors available on the GPU. For example, a block size of 256 threads and a total thread count of $T = 16384$ or $T = 32768$ work well on the GeForce GTX580 (16 multiprocessors), whereas $T = 27648$ works well on the GeForce GTX260 (27 multiprocessors).

A different approach has to be taken if a single sequence of random numbers is required. This approach, referred to as CUDA 2, only makes sense if most of the CUDA threads require the same number of random deviates and if giving the control back to the host before the next random number is needed does not come at a high cost or has to be done by the algorithm which consumes the random numbers anyway. The latter is necessary because this is the only way to synchronise across all CUDA threads. Algorithms 6 and 7 show how Marsaglia's algorithm can be adapted to generate random numbers in parallel using a single, large lag table. This approach is based on the fact that the window between the table indices i and j is shifted by one every time a new random deviate is generated and that they start with an offset of $\frac{2}{3}$ of the table size L. This means that $\frac{1}{3}L+1$ random numbers can be generated before index j reaches the starting index of i. It takes 3 iterations with either $\frac{1}{3}L$ or $\frac{1}{3}L+1$ threads each to generate L random numbers, as the table length is a prime and therefore odd. The only value that changes every time a random number is generated is c, but this is not a problem as all future values can be calculated as shown in the code fragments. However, the values for c calculated in this way and thus the resulting random numbers are slightly different to those generated in the usual fashion due to floating point rounding errors. This means that in order to get the same results when running a simulation with the same seed multiple times, it is necessary to use the same RNG implementation every time and not use this CUDA implementation once and the CPU implementation the next time.

The host code initialises the lag table before it is copied to the device. It then calls the CUDA kernel three times with different offsets into the lag table, generating $L/3$ or $L/3+1$ deviates in each call for a total of L new random numbers. With a lag table of length 92153 and a thread block size of 64, 30720 CUDA threads are executed in each call, $2-3$ of which are unused.

Both CUDA implementations are mainly useful when the random numbers are consumed by other device functions, in which case they never have to be copied back to the host and often do not even have to be stored in global memory, but only exist in the local registers of the streaming multiprocessors. Lag table operations usually require global memory transactions, but if the conditions mentioned before are adhered, then all of these can be coalesced into 1 (approach 1) or $1-2$ (approach 2) transactions per half-warp.

4.4. Multi-Platform Lagged-Fibonacci Performance Results

The implementations of the lagged-Fibonacci generator for different architectures were tested by generating 30 billion random numbers and measuring the time taken. Each test was performed 10 times and the mean values of the performance measurements are presented in Table 4.1. The deviations from the mean values are insignificant. The random numbers have not been used for any purpose as the only intention was to measure the generation time. The deviates were merely summed up and finally written to global memory on the GPU to avoid having the compiler remove code that appears useless during its optimisation phase. This is obviously not useful in itself but it is assumed that any application generating random numbers such as these will make use of them on the same device as they were generated.

Algorithm 6 Host code for the CUDA implementation of Marsaglia's RNG that produces a single stream of random numbers using a large lag table.

$L = 92153$ //lag table length
$T = L/3+1$ //thread count
$D = 7654321.0/16777216.0$
$M = 16777213.0/16777216.0$
function RNG1(s)
  Input parameters: s is the initialisation seed.
  $U[L] \leftarrow$ initialise with seed s //the lag table in host memory
  allocate $U_d[L]$ in device memory //the lag table
  copy $U_d \leftarrow U$
  $c \leftarrow 362436.0/16777216.0$
  while more random numbers required **do**
    //every iteration generates L random deviates
    $o \leftarrow 0$ //offset into the lag table
    $l \leftarrow L/3+1$ //update l table elements
    do in parallel on the device using T threads: call KERNEL(l, o, U_d, c)
    $o \leftarrow o+l$
    $l \leftarrow$ **round**$(L/3)$
    do in parallel on the device using T threads: call KERNEL(l, o, U_d, c)
    $o \leftarrow o+l$
    $l \leftarrow L/3$
    do in parallel on the device using T threads: call KERNEL(l, o, U_d, c)
    $c \leftarrow c-LD$ //update c for the next iteration
    $c \leftarrow c+$ **ceil**(**fabs**$(c)/M)M$
  end while

Algorithm 7 Device code for the CUDA implementation of Marsaglia's RNG that produces a single stream of random numbers using a large lag table.

function KERNEL(l, o, U, c)
  $t \leftarrow$ thread ID queried from runtime
  if $t < l$ **then**
    $r_{idx} \leftarrow L-1-t-o$ //index r_{idx} into lag table
    $s_{idx} \leftarrow L/3-t-o$ //index s_{idx} into lag table
    if $s_{idx} < 0$ **then**
      $s_{idx} \leftarrow s_{idx}+L$
    end if
    $c \leftarrow c-(t+o+1)D$ //calculate c for thread t
    if $c < 0.0$ **then**
      $c \leftarrow c+$ **ceil**(**fabs**$(c)/M)M$ //until $0 \leq c < 1$
    end if
    $r \leftarrow U[r_{idx}]-U[s_{idx}]$ //new random deviate
    if $r < 0.0$ **then**
      $r \leftarrow r+1.0$
    end if
    $U[r_{idx}] \leftarrow r$
    $r \leftarrow r-c$
    if $r < 0.0$ **then**
      $r \leftarrow r+1.0$
    end if
    do something with random number r
  end if

The results show that the concurrent implementations all perform well compared to the single-core CPU implementation. This comes as no surprise, as all threads execute independently from one another, using different lag tables and generating multiple streams of random numbers (with the exception of CUDA 2). The lag table size generally does not have a large effect on the performance, only the GTX260 shows a somewhat more significant drop in performance between sizes 1021 and 4093. The decision for the lag table size thus mostly comes down to the amount of memory available, unless extremely long periods are

Table 4.1.: Comparison of the time taken to generate 30 billion random numbers using the lagged-Fibonacci generator on different hardware architectures. The CPU used for these measurements is an Intel Core i7 970. Three different lag table sizes are used to test the effect on the performance, except for CUDA 2 which uses a fixed size lag table of length 92153. The speed-up is reported relative to the single-core CPU implementation using the smallest lag table.

Device	**Time in seconds using lag table size:**				**Speed-up**
	97	**1021**	**4093**	**92153**	
CPU (1 core)	229.0	232.7	232.7	–	1.0x
CPU -TBB (6 cores)	36.8	37.3	37.5	–	6.2x
GTX580 - CUDA 1	2.3	2.5	2.5	–	99.6
GTX260 - CUDA 1	4.2	4.4	5.1	–	54.5
GTX580 - CUDA 2	–	–	–	9.3	24.6
GTX260 - CUDA 2	–	–	–	17.3	13.2

required. Throughout this thesis a lag table of 97 deviates, the value proposed by Marsaglia [31], is used.

The TBB implementation benefits slightly from Intel's hyper-threading technology, which allows each physical processor core to appear as two logical cores and to work on two tasks at the same time, improving the utilisation of the physical core and increasing throughput. This makes a speed-up of more than 6 possible. The implementation uses 300 tasks to ensure that TBB's thread scheduler does not starve some of the CPU cores (see Figure 4.1).

The independence between threads is the perfect situation for the GPUs and the speed-up values for CUDA 1 clearly reflect this. In implementation CUDA 2, on the other hand, all threads work together to produce a single stream of random numbers using a large lag table. The performance is significantly lower than what is achieved using CUDA 1, but it is by no means unusable if there is a good reason to prefer a single sequence of deviates. The initial set-up time is insignificant compared to the time taken to generate 30 billion random numbers and can be ignored in most use cases.

4.5. GPU Performance Results

This section focuses on the GPU and compares the performance of the CUDA implementation "CUDA 1" of Marsaglia's RNG to a number of different pseudo-RNG algorithms proposed in Numerical Recipes third edition [115]: Ran, Ranq2, Ranq1, Ranlim32 and Ranhash. The CUDA implementations of these RNG algorithms are straight forward and basically the same as the sequential CPU implementations. Each CUDA thread uses its own RNG instance and thus generates an independent stream of random numbers just like algorithm CUDA 1.

All algorithms are used to generate uniform deviates on $[0,1)$, as this is the distribution mostly used throughout the algorithms discussed in this thesis. Marsaglia's lagged-Fibonacci generator and Ranlim32 are the only algorithms out of the bunch that use 32-bit arithmetic, the other algorithms all require 64-bit double precision. Ran is the highest quality generator recommended by Press et. al. in Numerical Recipes, with a period of $\approx 3.138 \times 10^{57}$. Ranq2 (period $\approx 8.5 \times 10^{37}$) and Ranq1 (period $\approx 1.8 \times 10^{19}$) are deemed good enough for "everyday" use, whereas Ranlim32 is only recommended as fallback when 64-bit arithmetic is not available. Ranhash is not a RNG in the usual sense, as it maintains no state and generates a random hash taking a 64-bit integer as input for every deviate it produces. It is purely out of curiosity that it was tested alongside the others. Once again, each algorithm is used to generate 30 billion random numbers. 10 measurements are performed and the respective mean values are reported in Table 4.2. The measurement errors are insignificant. Algorithm 8 describes the test setup used for the performance measurements.

Algorithm 8 This pseudo-code describes how the performance of the different RNG algorithms was measured. `rng_params` is a place-holder for all algorithm specific parameters. Every thread sums the random numbers that it generates and finally stores the value to global memory to avoid code from being removed during the compiler's optimisation phase. T is the total number of threads.

```
tid ← thread ID queried from CUDA runtime
if tid < T then
    params ← load rng_params for this thread from global memory
    initialise(tid, seeds[tid], params) //initialise the RNG stream with its individual seed
    x ← 0.0
    for i ← 1 to ceil(3 × 10^10 / T) do
        x ← x+ generate_uniform(params)
    end for
    rng_params ← save params to global memory
    results[tid] ← x //store x to global memory
end if
```

Table 4.2.: Comparison of the time taken to generate 30 billion random numbers on the GPU using different RNG algorithms.

Performance results	Ran	Ranq2	Ranq1	Ranhash	Ranlim32	Marsaglia (CUDA 1)
GTX580	2.0	1.2	1.1	1.5	1.5	2.3
GTX260	10.5	6.0	6.3	8.0	6.5	4.2

The double precision performance improvements made to the Fermi architecture based GTX580 compared to the previous generation GTX260 are clearly reflected in the results. The latter device suffers much more from the jump to 64-bit than the former, which performs over five times faster on the Ran algorithm but only just under two times better on Marsaglia's generator. The performance difference for Ranlim32 is with over four times also relatively large, which is most likely due to the fact that the GTX260 needs to execute multiple instructions to perform 32-bit integer multiplications, as it only supports 24-bit integer arithmetic natively. The GTX580, on the other hand, does support 32-bit integer arithmetic natively. Ranlim32 uses considerably more instructions than Ranq1 or Ranq2, enough to make it slower than those algorithms on both devices. There is, therefore, no reason to consider using this algorithm on these GPUs.

The Ran algorithm is the most interesting algorithm out of those suggested in Numerical Recipes, as it produces the highest quality random numbers, and the Monte Carlo algorithms discussed later on in this thesis rely heavily on the quality of the deviates used. The real match is therefore between Ran and the lagged-Fibonacci generator. The GTX580 deals well with the 64-bit Ran algorithm, it even slightly outperforms the CUDA implementation of Marsaglia's RNG. Ran also requires considerably less memory, as it does not have to maintain a lag table. It would therefore be the algorithm of choice for this thesis if the performance of the GTX260 did not suffer so much from the 64-bit calculations. As it is, Marsaglia's algorithm is selected as the best choice overall, considering its performance on both devices and the established quality [31, 106] of the deviates generated.

Chapter 5

Parallel Graph Generation

Complex networks arising from generator algorithms [1] like the Watts α small-world model [83] or Barabási-Albert scale-free model [58] have non-trivial topological features similar to those found in many real-world networks such as social networks [64, 74], metabolic networks [56, 71, 72], or neural networks [73]. Many of these real networks possess the properties of high clustering [54] and short mean vertex-vertex path lengths [117, 118] characteristic to small-world networks [66], or exhibit power-law degree distributions found in scale-free networks. Other metrics that can give helpful insights into the structure of graphs include betweenness measures [39,59], which play an important role in the attack vulnerability of a network [62,63]; circuit and loop structures [65, 119]; and reachability [120].

Being able to generate synthetic graph or network data with the same properties as real networks is very useful, as it helps to systematically investigate the statistical properties of such networks on various sample network realisations. This is a powerful approach, as it is often difficult or costly to obtain a large number of independent data sets from experiments and real physical systems. But it is important to be able to analyse a large enough data set to detect and eliminate any noise and inaccuracies that may be present in individual instances. It is generally also easier to make comparisons with theoretical predictions when comparing with synthetic data, as it is often possible to vary certain parameters of the model and systematically study the effects on the properties of the generated graphs.

Many of the interesting phenomena of complex networks are only revealed over multiple length scales. It is therefore often necessary to generate graphs with a large number of nodes and edges, which becomes a computational challenge for the models discussed here. This chapter describes parallel computing techniques for multi-core CPUs and GPUs to reduce the wall clock times of graph generators for the Watts small-world α-model and Barabási-Albert scale-free network model.

The CUDA implementations focus on a relatively new feature introduced with CUDA Toolkit 3.2 [90], namely, the ability to dynamically allocate and free global device memory from device code. Previously, it was necessary to use library calls in the host program running on the CPU to do all global device memory management. The new `malloc()` and `free()` device functions enable the developer to write applications that require dynamic memory management – such as graph generators – without having to return control to the host program whenever more memory needs to be allocated. The developer has to specify the heap size required for all dynamic memory allocations in device code before the first call to the respective device function. CUDA simply fails with an undescriptive error message if it runs out of heap memory. It is therefore important to allocate a large enough heap and to take into account that the pointers returned by calls to `malloc()` are aligned to 16 byte boundaries, which means that some memory does not get used

[1] This chapter extends on work first published in [116] **A. Leist** and K. A. Hawick, "Graph Generation on GPUs using Dynamic Memory Allocation," in *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'11)*, no. PDP3939, Las Vegas, USA, July 2011.

Figure 5.1.: A disconnected "caveman" network generated using the Watts α-model with $n = 500$ vertices, degree $k = 10$ and $\alpha = 1.0$. Individual nodes are well connected within their local "cave" (community), but there are no connections between "caves".

unless the allocation size is always a multiple of 16 bytes. The same goes for memory fragmentation, where it is possible that even though enough heap memory is available, none of the remaining continuous memory regions is large enough to satisfy the request. The following sections will show the use and analyse the performance of these new memory management functions.

5.1. Small-World Graphs

As described in Section 2.6, the parameter α is used to interpolate between the *caveman* model and the "Solaria" model. Nodes in the "caveman" model are highly interconnected within their own "cave" or cluster, where they share many neighbours with their peers, but have hardly any connections to nodes from other communities. In the "Solaria" world, on the other hand, most connections are formed randomly and localised community structures do not form. Figure 5.1 shows an example of a disconnected "caveman" world. The algorithm for the α-model is given in Appendix B.

As can be seen from the algorithm, there are some compute-intensive stages involved in generating α-model synthetic networks. The $\mathcal{O}(n^2k)$ complexity of the algorithm arises from the need to compute a property over all vertices for each edge of each vertex. It is therefore attractive to find a way of accelerating or parallelising the algorithm to make it more feasible to simulate networks with large n and k.

The α-model has been implemented for: a single CPU core, a multi-core CPU utilising all available cores, and a graphics device used as compute accelerator. Implementation details and performance results are given in the following sections.

5.1.1. The Sequential CPU Implementation

The sequential CPU implementation is used as reference to measure the scaling behaviour of the multi-threaded implementation and to explain the basic steps of the algorithm. Algorithm 9 uses pseudo-code to describe the steps that are executed when generating a graph using the α-model [83].

The propensity $R_{i,j}$ of vertex v_i to connect to vertex $v_j \in V, j \neq i$ is calculated according to Equation B.1. The baseline, random probability of an edge existing is $p_{base} = \binom{n}{2}^{-1}$. To account for the $p \ll p_{base}$ in the

Algorithm 9 Pseudo-code for the sequential CPU implementation of Watts' α-model network generator.

```
//generate M ← kn/2 edges
for e ← 1 to M do
  //R is the set of vertices not yet chosen in this round
  if R = {} then
    R ← init. remaining vertices to set of all vertices V
  end if
  v_i ← randomly chosen vertex from R
  remove v_i from R
  for all v_j ∈ V, j ≠ i do
    p ← p_base× uniform random number
    determine if edge e_{i,j} exists
    count the neighbours shared by v_i and v_j
    P[j] ← compute v_i's propensity to connect to v_j
  end for
  normalise the results in P
  //select v_i's new neighbour v_j
  r ← uniform random number
  p_sum ← 0.0
  for all v_j ∈ V, j ≠ i ; p_sum < r do
    p_sum ← p_sum + P[j]
  end for
  insert v_j into adjacency-list A_i of v_i
  insert v_i into adjacency-list A_j of v_j
end for
```

equation, the probability p is calculated for every possible edge $e_{i,j}$ by multiplying the baseline probability with a uniform random number.

All implementations of the generator described here insert a new neighbour into an adjacency-list so that the list is sorted in ascending order. This significantly improves the performance when counting the neighbours shared by two vertices, as it eliminates the need to compare every element in the adjacency-list of one vertex to every element in the adjacency-list of the other vertex.

5.1.2. The Multi-Threaded CPU Implementation

TBB is used to parallelise the CPU implementation of the α-model. To speed-up the graph generator algorithm, the inner-loops need to be parallelised where possible. The following loops from Algorithm 9 can be executed in parallel:

- The set R of vertices not yet chosen in the current round is initialised using `tbb::parallel_for`.
- The random numbers needed for $p \leftarrow p_{base} \times$ *uniform random number* are generated using T instances of `tbb:tbb_thread`, where T is the number of logical CPU cores available. Each thread generates n/T random numbers using its own random number generator (RNG) instance to ensure repeatability of the algorithm. The random numbers are stored in an array for consumption in the next step. Repeatability is also the reason why the random numbers are not generated as part of the following `tbb:parallel_reduce` step. The TBB scheduler assigns the tasks used during the reduction operation dynamically to hardware threads and does not expose this information to the developer, making it difficult to generate the same sequence of random numbers over multiple runs initialised with the same seed.
- The most time consuming loop by far is the computation of v_i's propensity to connect to all $v_j \in V$. It is however fairly straight forward to parallelise, as there are no dependencies between the iterations except for the calculation of the total propensity sum, which is later on needed to calculate the uniform

Algorithm 10 Propensity computation using TBB's `tbb::parallel_reduce`. This code uses the lambda expressions introduced in the upcoming C++0x standard. Lambda expressions let the compiler do the work of creating the function objects needed by TBB, which makes the code easier to read. Each task iterates over a range of values, computing the respective propensity values.

```
double totalPropensity = tbb::parallel_reduce(
  tbb::blocked_range<int>(0, nVertices), 0.0,
  [=](const tbb::blocked_range<int>& range, double init)->double {
    //this lambda function computes the propensities for a range of vertices
    double sum = init; // sum over given range
    const int end = range.end();
    for (int i = range.begin(); i < end; ++i) {
      // compute propensity[i] here
      sum += propensity[i]; // add to sum
    }
    return sum;
  },
  [](double x, double y)->double {
    //this lambda function combines two results
    return x+y;
  }
);
```

propensity values. This can be implemented using the `tbb::parallel_reduce` operation as shown in Algorithm 10.

The last remaining inner-loop iterates over the propensity values, normalises these values and computes the inclusive prefix-sum until a random number exceeds the sum, at which point it has found the new neighbour v_j and can stop running. This loop is computationally cheap, runs for only $n/2$ iterations on average, would require a parallel scan over all n elements and another parallel operation to determine v_j and is therefore not worth parallelising on the CPU.

The two function calls needed to create the new edge by inserting the two vertices into each others adjacency-lists (v_j into A_i and v_i into A_j) can be executed in parallel using `tbb::parallel_invoke`. However, this decreased performance slightly when tested and is thus not done in parallel.

5.1.3. The CUDA GPU Implementation

The GPU has a highly data-parallel architecture with many processing units, making it very powerful when executing the same instructions on large arrays of data, but making it difficult to achieve good performance when running algorithms that are more serial in nature, have many conditional branches or are bandwidth-limited. The task of generating a graph like the α-model is particularly challenging, as it is necessary to repeatedly iterate over the neighbours structure, which puts high demands on the memory bandwidth and does not perform many compute instructions per data element. But the recent interest in the Graph500 [27] and the newly gained ability to dynamically allocate and free memory in device code makes it an interesting challenge. Algorithm 11 describes the host code that coordinates the device kernel execution.

Marsaglia's random number generator [31] is used to generate both the host and device random numbers. For large networks it would not be feasible to have a separate RNG instance for every vertex, as each instance requires 400 bytes of global memory. Therefore, a specialised RNG kernel is used, which is executed by T CUDA threads, to generate the random numbers. Every thread generates x random numbers per kernel call, where $x = ceil(n/T)$. See Section 4.3 for a discussion of the value of T.

Algorithm 12 shows how the propensity values are computed on the device. A_i can be loaded to shared memory using coalesced memory transactions as all threads in the block cooperate to load it from global

Algorithm 11 Pseudo-code for the CUDA implementation of Watts' α-model network generator. This is the host code that manages the CUDA execution.

```
allocate device memory incl. sufficient heap memory
generate seeds for the device RNGs and copy to device
do in parallel on the device using T threads: initialise device RNGs
V_deg ← 0 //init. the vertex degree device array
create CUDPP plan for parallel scan on the device
//generate M ← kn/2 edges
for e ← 1 to M do
   //R is the set of vertices not yet chosen in this round
   if R = {} then
      R ← init. remaining vertices to set of all vertices V
   end if
   v_i ← randomly chosen vertex from R
   remove v_i from R
   do in parallel on the device using T threads: generate n random numbers
   do in parallel on the device using n threads: call compute_propensity_kernel
   do in parallel on the device using n threads: call normalise_propensity_kernel
   do in parallel on the device: call cudppScan (inclusive prefix-sum)
   do in parallel on the device using n threads: call select_nbr_kernel
   do in parallel on the device using 2 thread blocks: call add_arc_kernel
end for
destroy CUDPP plan and free device memory
```

Algorithm 12 The device code for `compute_propensity_kernel`.

```
Input parameters: v_i, p_base, alpha, k
v_j ← global thread ID queried from CUDA runtime
sum_s ← 0.0 //init. block local propensity sum array in shared memory
p ← p_base× (random uniform number)
k_i ← load degree of vertex v_i
A_i(ptr) ← load the pointer to adjacency-list A_i
A_i ← load adjacency-list at A_i(ptr) into shared memory
synchronise thread block
determine if edge e_{i,j} exists (v_j ∈ A_i)
if edge does not exist then
   k_j ← load degree of vertex v_j
   A_j(ptr) ← load the pointer to adjacency-list A_j
   count the neighbours shared by v_i and v_j
   p ← compute v_i's propensity to connect to v_j
end if
sum_s[btid] ← p //btid is the thread idx within the block
P[v_j] ← p //write propensity to global memory
synchronise thread block
sum_s ← perform reduction operation
//sum_s[0] now contains the block local sum
if btid = 0 then
   atomically add the local sum sum_s[0] to the global sum
end if
```

memory. The data in A_i is used by all threads. A_j is different for every vertex and the global memory transactions are not coalesced, but the automatic caching on Fermi devices helps to keep the throughput relatively high. Algorithm 13 shows the device function that is called to count the shared neighbours. It has to loop over both adjacency-lists and is therefore highly critical to the performance.

After the propensity values and propensity sum have been computed, they need to be normalised. This is all that `normalise_propensity_kernel` needs to do. The array of normalised propensity values is then passed to `cudppScan`, a function of the CUDA Data Parallel Primitives Library [121] (CUDPP), which performs an inclusive parallel prefix-sum operation on the input data.

The result of the scan operation is the array $P_{i,j}, \forall j \neq i$ of subintervals in the range $[0,1)$. This is passed to

Algorithm 13 Device function `countSharedNbrs` counts the number of neighbours shared by two vertices. It assumes that the adjacency-lists are sorted in ascending order. The performance of this function is critical to the overall performance. A_i is stored in shared memory.

```
__device__ __forceinline__
Uint countSharedNbrs(int v_i, int* A_i, int v_j, int* A_j) {
 Uint count = 0;
 for (int idx1=0, idx2=0, tmp; idx1<v_i && idx2<v_j;) {
  count = A_i[idx1]==A_j[idx2] ?count+1 :count;
  tmp   = A_i[idx1]<=A_j[idx2] ? idx1+1 : idx1;
  idx2  = A_j[idx2]<=A_i[idx1] ? idx2+1 : idx2;
  idx1  = tmp;
 }
 return count;
}
```

Algorithm 14 The device code for `select_nbr_kernel`.

Input parameters: random number *rnd*
$v_j \leftarrow$ global thread ID queried from CUDA runtime
$P_s[btid+1] \leftarrow$ load scan results from $P[v_j]$ into shared mem.
if $btid = 0$ **then**
  if $v_j > 0$ **then**
    $P_s[0] \leftarrow$ load last value from previous block $P[v_j - 1]$
  else
    $P_s[0] \leftarrow 0$ //first subinterval, no lower value
  end if
end if
//determine if *rnd* falls into the subinterval $P_{i,j}$
if $P_s[btid] \leq rnd$ AND $P_s[btid+1] > rnd$ **then**
  write v_j to mapped host memory
end if

`select_nbr_kernel`, along with a uniform random number. The kernel, described in Algorithm 14, then determines for every v_j if the random number falls into the respective subinterval. To do this, each thread checks whether the random number is larger than or equal to the upper end of the propensity range for the previous vertex $P[v_j - 1]$ and smaller than the upper end of the propensity range $P[v_j]$. This is the case for exactly one vertex v_j, which is selected as the new neighbour.

Finally, the edge $e_{i,j}$ can be created. This is done in Algorithm 15 `add_arc_kernel`. This kernel is only executed by 2 thread blocks, one for each end of the new edge, which it inserts into the respective adjacency-list. This only utilises a fraction of the processing units on the device (16 multiprocessors with a total of 512 processing units on the GTX580). To improve device utilisation, two CUDA streams are used to concurrently run the `add_arc_kernel` and generate the random numbers needed for the next iteration. These two kernels have no dependencies on each other and can therefore safely run in parallel.

The `add_arc_kernel` does not allocate new memory every time it adds a new vertex. Instead, it increases the allocated length by X elements when needed. The choice of X is critical to the performance, as the malloc operations are expensive. The minimum increment should be 4 for 32-bit arrays, as the pointers returned by malloc are aligned to 16 bytes. However, larger values of X mean less frequent memory re-allocations. Also, every time a longer adjacency-list is allocated, the contents of the old list have to be copied to the new memory region. On the other hand, values of X that are too large mean memory is being wasted. Good values for X are determined in the following section.

This CUDA implementation is referred to as CUDA 1. A second version of the algorithm, which is referred to as CUDA 2 respectively, is identical to the first implementation except for one key criterion: It replaces the `add_arc_kernel` with a kernel that merely determines the indices into the adjacency-lists at which the new neighbours have to be inserted to keep them sorted. The indices are written to mapped host

Algorithm 15 The device code for `add_arc_kernel`.

```
Input parameters: v_i, v_j
v ← v_i for block 0 and v_j for block 1
nbr ← v_j for block 0 and v_i for block 1
k_v ← load degree of vertex v
if btid = 0 AND k_v > 0 then
   A_v_old(ptr) ← load the pointer to A_v into shared mem.
end if
if btid = 0 AND need to allocate new A_v then
   A_v(ptr) ← allocate memory for k_v + X integers
else if btid = 0 then
   A_v(ptr) ← A_v_old(ptr) //keep using the existing array
end if
synchronise thread block
A_v ← load adjacency-list at A_v_old(ptr) into shared mem.
copy elements from A_v to A_v(ptr), adding one to the index if the current value is larger than nbr
if btid = 0 then
   insert nbr into A_v(ptr) so that A_v(ptr) is sorted (asc.)
   write k_v + 1 to global memory
   if A_v(ptr) ≠ A_v_old(ptr) then
      write the pointer A_v(ptr) to global memory
      if k_v > 0 then
         free memory pointed to by A_v_old(ptr)
      end if
   end if
end if
```

memory. The actual memory allocation, deallocation and copying is initiated in the traditional way by the host thread with calls to `cudaMalloc, cudaFree` and `cudaMemcpy(Async)`. This second implementation is used to compare the performance of the new in-kernel functions for dynamic memory management to the library calls initiated by the host thread.

The CUDA implementations of the graph generation algorithm require features only available on the latest generation of Fermi-based GPUs. In particular the device code memory management functions are not supported on older architectures.

5.1.4. Performance Results

To compare the performance of the different implementations of the α-model, the execution time for each of the algorithms is measured. As the plot in Figure 5.2 shows, the value of the α parameter has no significant effect on the performance of any of the tested architectures and is therefore set to the fixed value of $\alpha = 1.0$ for all further measurements. All performance results presented in this section are averaged over five independent runs. Error bars depicting the standard deviations from the mean values are plotted but smaller than the symbol size for most results.

As mentioned before, the allocation size multiple value X plays a critical role in the performance, in particular for implementation CUDA 1. It is therefore necessary to determine which values work best for a given network degree, which is most likely to affect the value as it directly determines the average adjacency-list length of the final graph instance. Compared to scale-free graphs, the individual node degrees of graphs generated with the α-model do not vary much from the mean degree. This makes it easier to pick a value for X that is large enough for most adjacency-lists without having to re-allocate them at all during the graph generation, while not being overly wasteful in terms of unused memory either. Figures 5.3 and 5.4 show how allocation size multiples from 2^2 up to 2^8 affect the performance for three different combinations of network size n and degree k for all implementations of the algorithm.

The performance impact of the chosen allocation size multiples on CUDA 1 is very pronounced. Inter-

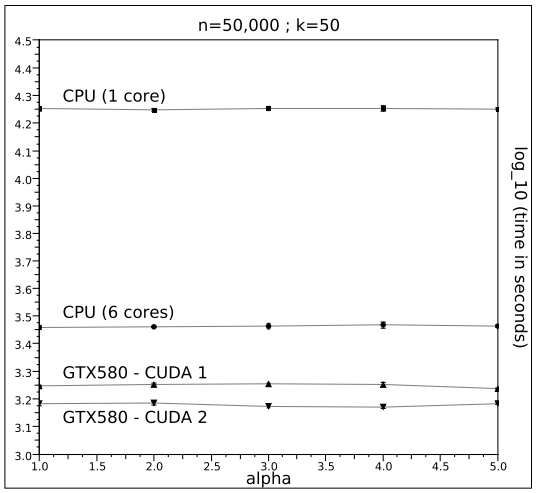


Figure 5.2.: This plot shows that the value of the α parameter has little influence on the execution time of the algorithms. The network size $n = 50,000$ and the mean degree $k = 50$.

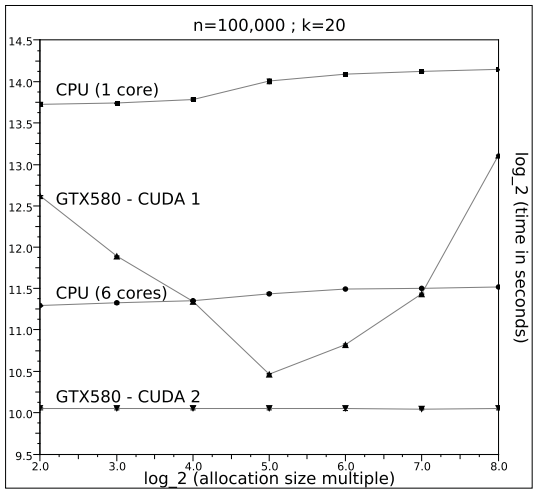
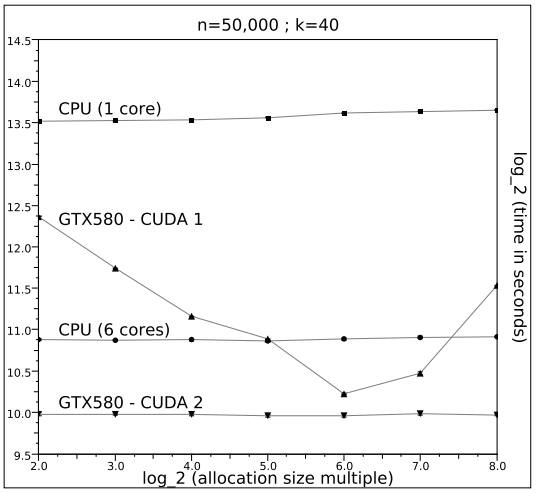


Figure 5.3.: The effect on the performance when scaling the allocation size multiple for graphs with size $n = 100,000$ and degree $k = 20$.



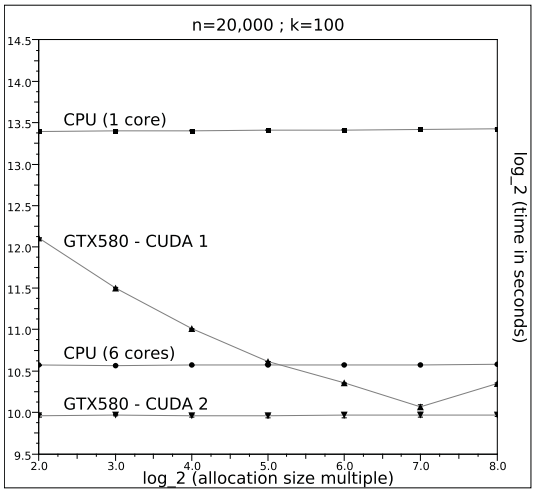


Figure 5.4.: The effect on the performance when scaling the allocation size multiple for graphs with size $n = 50,000$, degree $k = 40$ on the left and $n = 20,000$, $k = 100$ on the right.

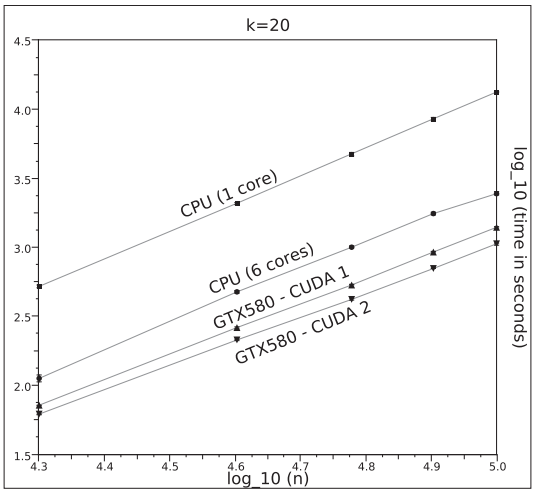


Figure 5.5.: This plot shows how the algorithms scale with the network size $n = 20,000$ to $100,000$. The mean degree is set to $k = 20$ and $\alpha = 1.0$.

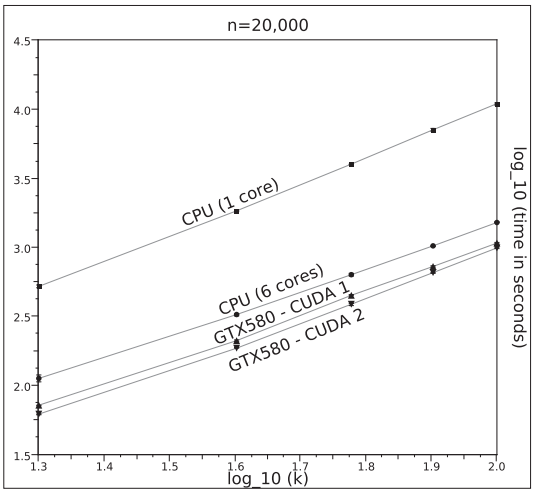


Figure 5.6.: This plot shows how the algorithms scale with the mean degree $k = 20$ to 100. The network size is set to $n = 20,000$ and $\alpha = 1.0$.

estingly, values that are too large are almost as bad as values that are too small. This shows that it is not only costly to call the device code `malloc()` function, and afterwards copy the contents of the old adjacency-list to the new memory region, but that the amount of memory allocated greatly matters too. In all three cases, the best performance is achieved when X is chosen to be the value just larger than the average degree. Specifically, the best value of those tested for degree $k = 20$ is $X = 2^5 = 32$, for $k = 40$ it is $X = 2^6 = 64$ and for $k = 100$ it is $X = 2^7 = 128$. This makes sense as a value slightly larger than the mean degree means that most adjacency-lists are only allocated once, but the overhead of allocating too much memory is kept small. All further performance measurements using CUDA 1 automatically choose a value for X that is the smallest power of two that satisfies $X \geq 1.2k$.

The host code library calls to `cudaMalloc()` are clearly less expensive and the chosen value for X does not significantly affect the performance of algorithm CUDA 2. Values smaller than 64 are not recommended, however, as the pointers returned by this library function are aligned to at least 256-bytes (see the CUDA Guide [90] chapter 5.3.2.1.1). For all further performance measurements, the value of X in CUDA 2 is chosen to be at least 64 and to scale in the same way as it does in CUDA 1 to minimise the overhead caused by memory copies when the network degree is very large.

The results for the CPU implementations show that a value of X much larger than the degree decreases the performance slightly. A small value like $X = 4$, however, does not have a negative impact on the performance even for degree $k = 100$. The CPU implementations thus use $X = 4$ for all further measurements.

Figure 5.5 compares the performance with increasing network size and fixed degree $k = 20$. For the largest measured network size $n = 100,000$, the multi-core TBB implementation runs 5.42 times faster than the sequential algorithm and thus scales well with the six physical cores available on the Intel Core i7 970 CPU. On a GeForce GTX580, the GPU implementation CUDA 1 runs 1.76 times faster than the TBB implementation and 9.56 times faster than the sequential CPU code. Implementation CUDA 2 is 2.32 times and 12.61 times faster than the sequential and TBB CPU implementations respectively. The slopes of the least square linear fits to the data sets on a log-log scale are: 2.02 for the sequential and 1.89 for the TBB CPU implementations, 1.84 for CUDA 1 and 1.76 for CUDA 2.

Figure 5.6 compares the performance with increasing degree and fixed network size $n = 20,000$. For the largest degree $k = 100$, the multi-core TBB implementation runs 7.16 times faster than the sequential

implementation. The speed-up value shows that Intel's hyper-threading technology once again pays-off, allowing the algorithm to achieve performance values higher than possible by the increase of physical cores alone. Implementation CUDA 1 running on the graphics device is 1.41 times and 10.13 times faster than the TBB and sequential CPU implementations respectively. CUDA 2 is 1.55 times faster than the TBB implementation and 11.08 times faster than the sequential CPU implementation. The slopes of the least square linear fits to the data sets on a log-log scale are: 1.90 for the sequential and 1.74 for the TBB CPU implementations, 1.68 for CUDA 1 and 1.70 for CUDA 2.

These speed-ups are not as impressive as those seen for more regular or compute heavy algorithms, but the results show that the GPU can still consistently outperform the CPU even for a graph generator problem like the one described here. The CUDA implementations also scale better to larger network sizes or degrees. The new device code `malloc()` and `free()` functions, while arguably more elegant, have been shown to be more expensive and to require more care when choosing the allocation size increments than the proven method of handing device memory management tasks to the host. Memory management on the GPU is in general an expensive operation and the next section shows how the CUDA implementations of a scale-free graph generator, with its much higher deviations from the mean adjacency-list length, hold up to the CPU implementations.

5.2. Scale-Free Graphs

As described in Section 2.4, the Barabási-Albert network model [58] generates graphs that evolve into a scale-invariant state with a power-law degree distribution. After t time steps, the size of the generated network is $n_0 + t$ vertices and $m_0 + mt$ edges, where n_0 and m_0 are the number of vertices and edges in the core network and m is the number of edges added at every time step. The size of the core network should be kept small, as it is typically a simple random or fully-connected graph, but it must fulfil the condition $m \leq n_0$ as it would otherwise be impossible to generate m unique edges at time step $t = 0$. The algorithms explained in the following sections always generate a fully connected core network with $m_0 = n_0(n_0 - 1)/2$ edges to give every vertex in the core a uniform chance of being chosen as a neighbour of the vertex added at time step $t = 0$.

5.2.1. The Sequential CPU Implementation

The sequential implementation is once again the reference implementation used to measure the scaling behaviour of the parallel implementations and to explain the basic steps of the algorithm. The pseudo-code in Algorithm 16 describes the steps that are executed when generating a graph using the Barabási-Albert network model.

A new vertex v_t is added at every time step t and connected to m different vertices v_i from the current vertex set V. The neighbours are chosen at random with probability $\Pi(v_i)$, which scales linearly with degree $k_i = |A_i|$. Once a vertex has been chosen, it is inserted into v_t's adjacency-list A_t such that the list is sorted in ascending order. For the first set of random numbers (i.e. iteration 1 of the loop starting in line 9), the selected vertices v_i can simply be stored in A_t in the order that they are found in (line 19). Because the random numbers are already sorted and the adjacency-list is initially empty, the vertices in A_t will also be sorted at the end of this iteration. This speeds the algorithm up considerably, as it is not necessary to determine the correct position to insert the neighbour nor to move the existing neighbours with a higher ID to make space for the new vertex. And it becomes less and less likely that a second iteration of the loop is needed as the network size becomes larger, because the chance that two random numbers fall into the same sub-interval in Π decreases, which means that most of the time a second set of random numbers is not needed at all. If a second iteration has to be done, however, then it is necessary to ensure that no duplicate

Algorithm 16 Pseudo-code for the sequential CPU implementation of the scale-free network generator.

```
 1: create the fully-connected core network
 2: for t ← n_0 + 1 to n do
 3:    for v_i ∈ V ≡ {1,2,3,...,t−1} do
 4:       //for each existing vertex, compute its propensity Π(v_i) to connect to v_t such that Π(v_i) = |A_i|/∑_j |A_j|
 5:       Π[i] ← |A_i|/(2|E|) //E ≡ {A_1,A_2,A_3,...,A_{t−1}}
 6:    end for
 7:    c ← 0 //edge counter
 8:    iter ← 0 //iteration counter
 9:    repeat
10:       iter ← iter + 1
11:       R ← generate m − c random numbers on [0,1)
12:       sort R in ascending order
13:       R_idx ← 0 //index into array R
14:       p_sum ← 0.0
15:       for v_i ∈ V ; R_idx < (m − c) do
16:          p_sum ← p_sum + Π[i]
17:          if p_sum > R[R_idx] then
18:             if iter = 1 then
19:                append v_i to the end of v_t's adjacency-list A_t
20:                c ← c + 1
21:             else if edge {v_t, v_i} does not exist then
22:                insert v_i into adjacency-list A_t, such that A_t remains sorted in ascending order
23:                c ← c + 1
24:             end if
25:             increment R_idx until p_sum <= R[R_idx]
26:          end if
27:       end for
28:    until c = m
29:    for v_i ∈ A_t do
30:       append v_t to the end of v_i's adjacency-list A_i //note that v_t is always larger than any existing vertex in A_i
31:    end for
32: end for
```

edges are created and that adjacency-list A_t remains sorted (line 22). It is important to remember that no more than $(m-c)$ random numbers can be created per iteration, as this would bias the algorithm towards choosing neighbours from the beginning of the vertex set after the random numbers are sorted. Adjacency-lists are extended in multiples of 4 elements to reduce the overhead of allocating new memory and copying the existing elements from the old list to the new list.

5.2.2. The Multi-Threaded CPU Implementation

This algorithm does not lend itself well to parallelisation, because none of the steps is particularly compute intensive. The first inner-loop, which calculates the probability array Π (line 3), is the only good point to accelerate the computation. There are no dependencies between iterations and it is thus straight forward to parallelise with `tbb::parallel_for`.

The random numbers could be generated in parallel in the same fashion as described for the α-model, but tests have shown that the overhead of parallelising this step far outweighs the benefits for the system sizes tested here. The amount of random numbers needed is simply not large enough for values of m (usually in the tens or hundreds), unlike in the α-model, where $(n-1)$ random numbers are needed for every edge.

Parallelising the entire repeat-until-loop (line 9) would introduce a considerable overhead too, as it would be more costly to ensure that the same vertex v_i is not selected twice by different threads. Additionally, inserting the new neighbours into the adjacency-list A_t would require extra checks and caution and the speed-up achieved in the first iteration of this loop, as explained in the previous section, could not be realised in the same way.

Algorithm 17 Pseudo-code for the CUDA implementation of the Barabási-Albert scale-free network generator. This is the host code that manages the CUDA execution.

```
allocate device memory including sufficient heap memory
create CUDPP plan for parallel scan on the device
create CUDPP plan for parallel sort on the device
do in parallel on the device using n_0 thread blocks: create the core network
for t ← n_0 + 1 to n do
  F_d ← 0 //initialise F_d, used to flag vertices already selected as neighbours
  do in parallel on the device using t threads: call compute_pi_kernel(Π_d)
  R_d ← generate m random numbers on host and copy to device
  M_d ← initialise index mapping array on host and copy to device
  do in parallel on the device: call cudppScan(Π_d) //inclusive prefix-sum
  repeat
    //fill the preliminary adjacency-list A_tmp(d) for v_t by fitting random numbers R_d to intervals Π_d
    do in parallel on the device using t threads: call select_nbr_kernel(A_tmp(d), R_d, M_d, Π_d, F_d)
    determine if m vertices have been selected, if not then re-initialise R_d and M_d for the next iteration
  until m neighbours have been selected
  do in parallel on the device: call cudppSort(A_tmp(d))
  do in parallel on the device using 128 threads: call create_adj_list_kernel(A_tmp(d))
  do in parallel on the device using m threads: call append_kernel(A_tmp(d))
  P_h ← P_d //copy pointers to the old and new adjacency-lists to host memory
  for i ∈ 1 to m do
    if P_h[i] ≠ NULL then
      copy all elements in P_h[i] to P_h[i + m] //copy old adjacency-list to new adjacency-list
    end if
  end for
end for
destroy CUDPP plans and free device memory
```

Even the last loop (line 29) proved to be counter-productive to parallelise with a `tbb::parallel_for` in the performance experiments. The number of instructions in the loop body is far too small to improve performance when only a few hundred iterations or so are performed.

5.2.3. The CUDA GPU Implementation

After the multi-core CPU implementation already provided so few opportunities for effective parallelisation, the CUDA implementation has a tough time trying to fare any better. Nevertheless, this section describes how to generate scale-free graphs using the Barabási-Albert model on the GPU.

Algorithm 17 shows the main steps executed by the host thread. The initialisation of arrays F_d, R_d and M_d can overlap with the execution of `compute_pi_kernel`. As the name suggests, the kernel computes the probabilities Π to connect to the new vertex v_t for all vertices v_i in the current vertex set V. This is the straight forward CUDA version of the loop in line 3 of the sequential CPU implementation, running one thread for every vertex in V. The results are written to Π_d. Once completed, `cudppScan` is used to compute the prefix sum of this array on the device. It now contains the half-open intervals $0.0 \leq \Pi(v_1) \leq \Pi(v_2), \Pi(v_3), \ldots \Pi(v_{t-1}) \approx 1.0$ ($\Pi(v_{t-1}) = 1.0$ if there are no rounding errors).

The next step is to select the neighbours for vertex v_t using `select_nbr_kernel` (see Algorithm 18). If all random numbers in R_d fall into different intervals in Π_d, then the kernel only needs to be called once. While this becomes more and more likely with increasing network size, sometimes two random numbers do fall into the same interval, especially when the network is still small. But because duplicate edges are not allowed, the respective vertex can only be used to create one edge and not two. Therefore, a new random number has to be generated for all such cases and the kernel needs to be called again to fill the remaining slots in the temporary adjacency-list $A_{tmp(d)}$.

The mapping array M_d described in the pseudo-code is not needed for the first call to `select_nbr_kernel` for a particular vertex v_t, but it is needed for all subsequent calls if any are necessary. This is because the

Algorithm 18 The device code for `select_nbr_kernel`.

Input parameters: $A_{tmp(d)}, R_d, M_d, \Pi_d, F_d, S_d$
$v_i \leftarrow$ query the thread ID from the CUDA runtime
$\Pi_s[btid+1] \leftarrow \Pi_d[v_i]$ //btid is the thread block local thread ID; load Π values for this block into shared mem.
if $btid = 0$ **then**
  if $v_i > 1$ **then**
    $\Pi_s[0] \leftarrow \Pi_d[v_{i-1}]$ //the first thread in the block loads the last value from the previous block
  else
    $\Pi_s[0] \leftarrow 0$ //this is the first thread block, no previous value
  end if
end if
synchronise thread block
for $r \in R_d$ **do**
  if $\Pi_s[btid+1] > r$ AND $\Pi_s[btid] \leq r$ **then**
    //this is true for exactly one vertex v_i per random number r
    isSuccess $\leftarrow$ flip the bit-flag stored in $F_d[v_i]$ //this avoids choosing the same vertex twice as neighbour for v_t
    $S_d[M_d[r_{idx}]] \leftarrow$ isSuccess //r_{idx} is the current index into R_d; M_d maps this to the correct index into the adjacency-list $A_{tmp(d)}$ and S_d marks the respective slot as filled
    if isSuccess = true **then**
      $F_d[v_i] \leftarrow 1$ //mark this vertex as 'selected' in F_d
      $A_{tmp(d)}[M_d[r_{idx}]] \leftarrow v_i$ //write v_i to the preliminary adjacency-list
    end if
  end if
end for

Algorithm 19 The device code for `create_adj_list_kernel`.

if first thread in thread block **then**
  $A_t \leftarrow$ allocate adjacency-list for v_t in dynamic device memory
  $k_t \leftarrow$ write degree m for vertex v_t to global memory
end if
synchronise thread block
use all threads in this block to copy $A_{tmp(d)}$ to A_t

adjacency-list $A_{tmp(d)}$ is already partially filled – the slots are marked accordingly in S_d – and only the remaining open slots need to be filled in the following calls. Only the required amount of random numbers are generated and written to R_d and now the indices into R_d do not map 1:1 to the open slots in $A_{tmp(d)}$ anymore. M_d takes care of this new mapping.

Once all neighbours for v_t have been chosen, the temporary adjacency-list $A_{tmp(d)}$ is sorted on the GPU using `cudppSort`. Then `create_adj_list_kernel` is called with a single thread block to allocate adjacency-list A_t of vertex v_t and fill it with the values from $A_{tmp(d)}$ (see Algorithm 19). As there are no dependencies between `create_adj_list_kernel` and `append_kernel` and because the former only utilises a single multiprocessor, the two kernels are executed concurrently on devices that support this feature (i.e. compute capability 2.0 and above, which includes Fermi-architecture devices).

Kernel `append_kernel` (see Algorithm 20) first checks if there is any memory from old adjacency-lists, as recorded in $P_d[0,1,2,\ldots,m-1]$ during the previous execution of this kernel, that can be released and calls `free()` for each pointer that it finds. It then checks the degree of all vertices in $A_{tmp(d)}$ and allocates new memory for adjacency-lists that are currently full. Memory allocation is done in multiples of X, just like it is done for the small-world graph generator described before. The choice of a good value for X is discussed in the performance results section below. Finally, `append_kernel` appends the new neighbour v_t to the end of the adjacency-lists, increments the degrees of all vertices $v_i \in A_{tmp(d)}$ and, for those threads that allocated a new adjacency-list, writes the old and new adjacency-list pointers to array P_d. The last step is done so that the host can copy these pointers to host memory and use them to quickly copy the content of the old adjacency-lists to the respective new arrays (Algorithm 17). Letting the host initiate these copies instead of doing it in the kernel solves the problem of copying adjacency-lists of different lengths and for

Algorithm 20 The device code for `append_kernel`.

Input parameters: memory is allocated in multiples of X; P_d is an array of pointers to pending memory
$tid \leftarrow$ thread ID queried from CUDA runtime
if $P_d[tid] \neq$ NULL **then**
  free memory pointed to by $P_d[tid]$
end if
$v_i \leftarrow A_{tmp(d)}[tid]$ //this thread processes vertex v_i
$k_i \leftarrow$ load v_i's degree from global memory
if k_i is a multiple of X **then**
  $A_{i(old)} \leftarrow A_i$
  $A_i \leftarrow$ allocate a new adjacency-list of length $k_i + X$
else
  $A_i \leftarrow$ load the pointer to the existing adjacency-list
end if
$A_i[k_i] \leftarrow v_t$ //write the new neighbour to the end of A_i
$k_i \leftarrow k_i + 1$ //increment the degree
if new adjacency-list has been allocated **then**
  $P_d[tid] \leftarrow A_{i(old)}$ //save the pointer to the old adjacency-list to global memory
  $P_d[tid + m] \leftarrow A_i$ //save the pointer to the new adjacency-list to global memory
end if

only some of the vertices, which would cause significant warp divergence and thus poor performance.

The CUDA implementation described so far shall again be referred to as CUDA 1. And just like it is done for the GPU implementation of the small-world graph generator, a second version, referred to as CUDA 2, has been implemented. CUDA 2 is mostly identical to CUDA 1, except that it does not invoke the device code memory management functions, but instead utilises the host thread to perform all memory management tasks. The tasks of `create_adj_list_kernel` – allocating the initial adjacency-list for v_t and copying $A_{tmp(d)}$ to A_t – are entirely performed by the host using library calls to `cudaMalloc` and `cudaMemcpyAsync`. Because the neighbours for v_t are chosen on the device, the host thread has to copy $A_{tmp(d)}$ to host memory before it can determine if some of the vertices in that list need a larger adjacency-list in order for v_t to be appended to it. It then allocates, frees and copies memory as necessary. The host also keeps a record of the vertex degrees, so that they do not have to be copied from device memory. While `append_kernel` still exists in CUDA 2, it's functionality is reduced to appending vertex v_t to the adjacency-lists of all vertices in $A_{tmp(d)}$ and to increment their degrees respectively.

5.2.4. Performance Results

The performance of the different implementations of the Barabási-Albert scale-free network generator is compared. All performance results presented in this section are averaged over five independent runs. Error bars depicting the standard deviations from the mean values are plotted but smaller than the symbol size for most results.

The first task is to determine good values for the allocation size multiple X described before. Figure 5.7 compares the performance using three different combinations of network size and degree for values of X ranging from 2^2 to 2^8. Similar to the results presented for the small-world graph generator, the CPU implementations are not significantly affected by the allocation size and a value of $X = 4$ is used for all further performance measurements.

While it is not surprising that the results for algorithm CUDA 1 are again heavily influenced by the choice of X, the curves look significantly different from those discussed before. Where the results for the α-model suggest that a value just higher than the mean degree is the best choice, here any value smaller than the mean degree performs well, whereas values larger than the mean degree decrease performance significantly. It is important to remember that in this model a large fraction of the nodes have a degree that is equal to or only slightly larger than $m \approx k/2$ (the number of edges used to connect each vertex to the network). Only

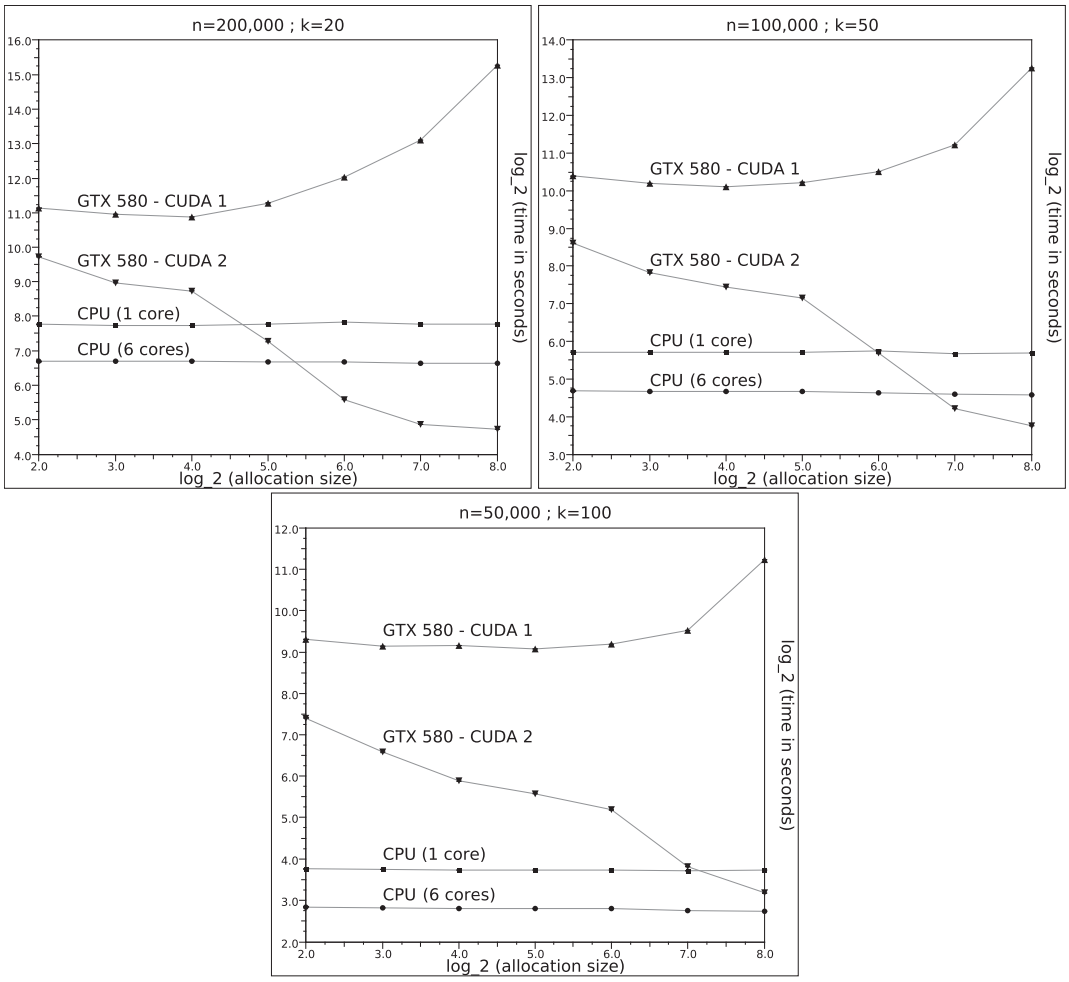


Figure 5.7.: The effect on the performance when scaling the allocation size multiple X from 2^2 to 2^8 for graphs with size $n = 200,000$, degree $k = 20$ (top left); $n = 100,000$, $k = 50$ (top right); and $n = 50,000$, $k = 100$ (bottom).

a small fraction of nodes have a much larger degree. It appears that for CUDA 1 the cost of allocating too much memory for most vertices outweighs the cost of having to re-allocate the adjacency-lists of the highly connected nodes more frequently. $X = 16$ works well for all tested degrees and is therefore used for all further performance measurements independent of the degree.

The results for CUDA 2 also differ significantly from those described for the α-model, where the value of X does not affect the performance much when using the host library functions for memory management. For the scale-free network model, the performance increases significantly as soon as the allocation size multiple is larger than the mean degree and it continues to improve up to the largest value tested. This shows that the amount of memory allocated during a single call to `cudaMalloc()` does not matter much compared to the overhead of having to repeatedly expand the adjacency-list of highly connected nodes. And while only a small fraction of nodes are highly connected, these hub nodes have a high propensity Π to connect to new nodes, causing their degree to grow very quickly with m (the rich-get-richer phenomenon). The best performance is clearly achieved with a high value of X, but a value that is much larger than m also wastes lots of memory and therefore a balance needs to be found. As described before, a value smaller than 64 does not make sense for 32-bit integers, as the pointers returned by the memory allocation function are

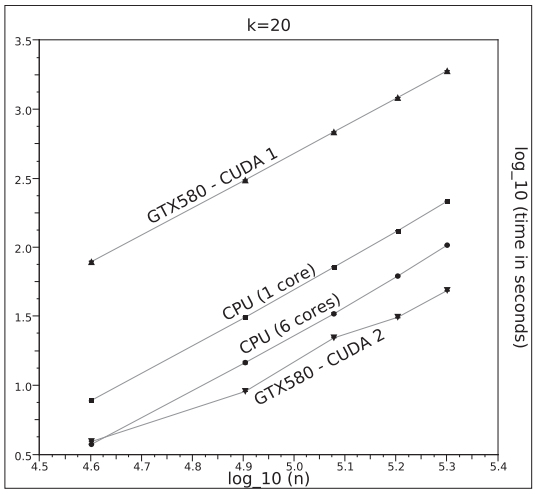


Figure 5.8.: This plot shows how the algorithms scale with the network size $n = 40,000$ to $200,000$. The average degree is set to $k = 20$.

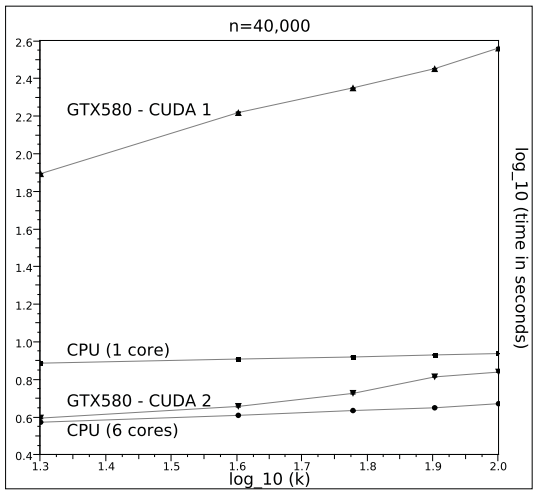


Figure 5.9.: This plot shows how the algorithms scale with the average degree $k = 20$ to 100. The network size is set to $n = 40,000$.

aligned to 256-bytes. This minimum value for X is automatically increased to the smallest power of two that satisfies $X \geq 2k$. While this still means that a significant amount of memory goes unused, the frequency in which the adjacency-lists of highly connected nodes have to be extended warrants a value on the higher end of the range to reduce the execution time. If memory usage is an issue for the algorithm that uses the graph after it has been generated, then it is always possible to re-allocate the entire neighbours structure once after the generator algorithm is finished and the individual adjacency-list lengths are known for all vertices.

Now that the values for the allocation size multiple are established for all implementations of the scale-free network generator, their performance can be compared with each other. Figure 5.8 shows how the execution time changes when the network size n is scaled from $40,000$ to $200,000$. $m = 10$ edges are used to connect every new vertex to the network, giving a mean degree $k \approx 20$. Implementation CUDA 1 clearly struggles with this algorithm, it is about an order of magnitude, 8.76 times on the GeForce GTX580 for the largest measured network size, slower than even the sequential CPU implementation on a Core i7 970. The relatively slow performance of the device code memory management functions is highlighted even more strongly here than it is for the small-world graph generator, because the scale-free network algorithm has less work to do when selecting the neighbours for a new vertex. Furthermore, the large deviations from the mean degree make it necessary to expand adjacency-lists much more frequently. The memory management therefore makes up a significantly larger part of the overall execution time.

Implementation CUDA 2 outperforms the multi-threaded TBB implementation by 2.16 times for the largest tested network. It is 4.51 times faster than the sequential implementation and 39.49 times faster than CUDA 1. While this is less than the performance difference between CUDA 2 and the CPU for the α-model, the fact that the multi-threaded TBB implementation is only 2.09 times faster than the sequential algorithm shows that this graph generator provides much less parallelisation potential. The slopes of the least square linear fits to the data sets on a log-log scale are: 2.06 for both the sequential and TBB CPU implementations, and 1.99 and 1.59 for CUDA 1 and CUDA 2 respectively.

Figure 5.9 compares the performance of the algorithms when the network size is fixed to $n = 40,000$ and the degree is scaled from 20 to 100. As a higher degree results in increasingly connected hub nodes, but does not significantly increase the computation time of the algorithm, the performance depends more

Algorithm 21 The memory allocation kernel used to allocate a total of 64 MB of memory. It is executed by 16384 threads, each allocating `count`$= 2^{26}/(16384 \times$ `ALLOC_SIZE`$)$ arrays of `ALLOC_SIZE` bytes each.

```
__global__ void malloc_kernel(Uint count, char** mem, int* d_errorFlag) {
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  for (Uint i = 0; i < count; ++i) {
    char* tmp = (char*) malloc(ALLOC_SIZE);
    if (tmp == NULL) {
      *d_errorFlag = 1;
      break;
    }
    mem[tid+i*THREAD_COUNT] = tmp;
  }
}
```

and more on the speed of the memory management functions. Algorithm CUDA 2 is merely 1.26 times faster than the sequential CPU implementation for the largest tested degree, but it takes 1.48 times longer to complete than the TBB implementation. CUDA 1 copes even worse and takes 42.23 times longer than the sequential CPU algorithm. The slopes of the least square linear fits to the data sets on a log-log scale are: 0.07 and 0.14 for the sequential and multi-threaded CPU implementations respectively, clearly showing that a change in degree does not affect them much. CUDA 1 has a slope of 0.94 and CUDA 2 has a slope of 0.37.

5.3. Memory Allocation Experiment

The CUDA host library memory management functions are consistently faster than the device code functions in the implementations of the graph generators. However, the results presented so far are also affected by memory copies performed either by calls to `cudaMemcpy(Async)` or by threads in a device kernel, as well as other small differences between the respective CUDA 1 and CUDA 2 implementations. It is therefore worth looking at the allocation and deallocation performance of the host library and device code functions separately to get a feeling for their actual costs when allocating many relatively small blocks of memory. To do this, a total of 64 MB of global memory are allocated using either the host library function `cudaMalloc()` or the device code `malloc()`. After all memory has been allocated, it is released using the appropriate deallocation function `cudaFree()` or `free()`.

While only 64 MB of memory are allocated in total, fragmentation makes it necessary to allocate a significantly larger amount of heap memory to ensure that the tests are successful. A heap size of 640 MB is allocated for the experiments using the device memory management functions. Fragmentation is even more of an issue when allocating very small arrays using the host library functions, as the pointers returned by `cudaMalloc()` are aligned to at least 256 bytes. In fact, the experiment with the smallest allocation size (16 bytes) does not complete successfully using the library functions on the GeForce GTX580 with its 1536 MB of device memory, because it runs out of memory.

The experiment is performed on a GTX580 using 16384 threads for the device code measurements. The CUDA memory allocation kernel executed by these threads is given in Algorithm 21. The deallocation kernel works the same way, only that it loads the address from global memory instead of storing it and that it does not need to check if the operation is successful. The code used to test the host library functions is similar to the device kernels, only that there is a single host thread calling the equivalent library functions in sequence instead of many CUDA threads doing this in parallel on the device.

Figure 5.10 presents the results of the experiment. The host library function `cudaMalloc()` takes about 65% less time and `cudaFree()` takes about 60% less time to complete for every doubling of the allocation

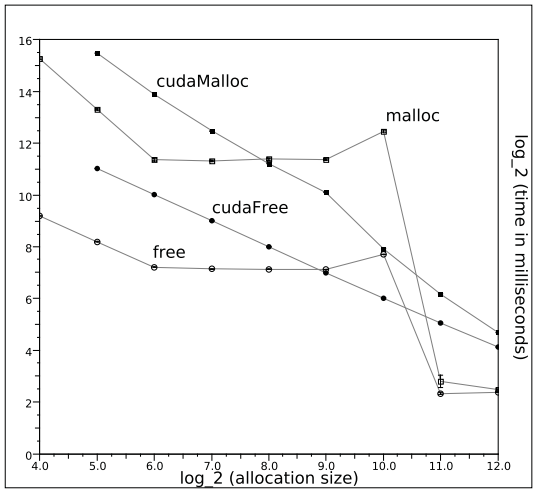


Figure 5.10.: Memory allocation and deallocation performance using either the host library `cudaMalloc` and `cudaFree` calls or the device code `malloc` and `free` calls. A total of 64 MB of memory are allocated using allocation junk sizes from 2^4 to 2^{12} bytes.

size. While very consistent for the deallocation routine, the allocation results have a slight deviation from this at 256 and 512 bytes allocation size, where the performance improves more slowly.

The results for the device code functions are not as straight forward. Every thread has to allocate from at most 256 arrays for allocation size 16 down to a single array for allocation size 4096. While the execution time drops quickly when the allocation size is increased from 16 to 64 bytes, it levels off at this value and does not change much at all up to an allocation size of 512 bytes. Interestingly, it takes longer to allocate four arrays of 1024 bytes per thread than it does to allocate eight arrays of 512 bytes. And then the performance increases drastically for an allocation size of 2048 bytes. It is not clear what exactly is happening "behind the scenes" that affects the performance so profoundly, but it is important to be aware of this sensitivity to the chosen allocation size when using the device code memory management functions.

5.4. Discussion

The addition of new features like dynamic memory allocation in device code makes CUDA an increasingly powerful environment that enables programmers to tap into the parallel processing power of today's graphics processing units. Many tasks in scientific applications or computer games have to perform computations on irregular graphs and dynamically allocate and free memory as needed.

The ability to generate synthetic graph or network data directly on the graphics accelerator is particularly important when this data is further processed on the same device. This saves time because now the graph data does not have to be copied over the PCI express bus before it can be used. A simulation of a complex system running on the GPU can directly modify the structure of an underlying graph without incurring the overhead of copying it back and forth between host and device memory.

Not only synthetic graph generators, but also dynamic network analysis algorithms that process data from constantly evolving real-world networks [122] can benefit from the ability to dynamically change graph structures. Many systems need to evaluate temporally or geospatially changing data. Systems that continuously monitor social behaviour on the World Wide Web can use this data to offer targeted suggestions to individuals based on their own behaviour or that of groups of individuals with similar interests. Other systems monitor the constantly changing and evolving data available on the Web to predict future

Table 5.1.: Summary of the performance results. The speed-up values are relative to the respective sequential CPU implementations and the quoted slopes are for the least squares linear fits to the data sets on a log-log scale.

Compute Device	**Small-world α-model**		**Scale-free network model**	
	Speed-up	**Slope**	**Speed-up**	**Slope**
	$n = 100,000$; $k = 20$; $\alpha = 1.0$		$n = 200,000$; $k = n_0 = 20$	
Core i7 970 (1 core)	1.00	2.02	1.00	2.06
Core i7 970 (6 cores)	5.42	1.89	2.09	2.06
GTX 580 - CUDA 1	9.56	1.84	-8.76	1.99
GTX 580 - CUDA 2	12.61	1.76	4.51	1.59
	$n = 20,000$; $k = 100$; $\alpha = 1.0$		$n = 40,000$; $k = n_0 = 100$	
Core i7 970 (1 core)	1.00	1.90	1.00	0.07
Core i7 970 (6 cores)	7.16	1.74	1.85	0.14
GTX 580 - CUDA 1	10.13	1.68	-42.23	0.94
GTX 580 - CUDA 2	11.08	1.70	1.26	0.37

events or to quantify reactions to past events. Network analysis is even employed in counterterrorism measures, where "dynamic metanetwork analysis" is able to deal with change in terrorist networks over time, answering questions regarding the "who, when, what, where and why" [123].

The performance results presented in this chapter are summarised in Table 5.1. They show – reflected by the results of the CUDA 1 algorithms for both generators – that the use of the dynamic memory heap in global memory and the associated device functions `malloc()` and `free()` can be computationally very expensive. However, the results presented in Figure 5.10 indicate that this depends on the exact scenario and that in a different situation – one where all threads allocate memory in parallel and the allocation size can be controlled more tightly – the device code functions might even perform better than the host library functions.

It may also be justified to use the device code functions, which are arguably the more elegant solution compared to having to hand control back to the host just to allocate some memory, when they are only called rarely to allocate small amounts of memory. But for algorithms like the graph generators discussed here, the "old fashioned" way of using `cudaMalloc()` and `cudaFree()` is most likely the better choice.

Compared to the CPU implementations running on a high-end six core Intel CPU of the same vintage, the GPU algorithms labelled "CUDA 2" for both the small-world and scale-free graph generators perform well on an NVIDIA GTX580 but do not deliver the same speed-up values achieved for some regular geometric problems. It is, however, interesting to observe that even for highly irregular and unbalanced problems, GPUs still provide some performance utility as accelerators to the CPU. While the performance of the generators alone may not justify the additional time needed to implement the much more complex CUDA algorithms, this can easily change when the algorithms are just one piece of a complex simulation running on the GPU. In this case the time saved by not having to copy a dynamically changing graph structure between host and device memory can quickly add up.

Chapter 6

Parallel Graph Analysis

Numerical graph analysis algorithms aim to determine certain properties of a given graph. The process can be based solely on structural information or it can include the values of particular variables on the nodes or connections of a graph. Graph properties can be global for the entire graph or local based on individual nodes or connections and their neighbourhood. Common questions may be whether a graph is simple, random or complex according to one of the metrics discussed in Section 2. Does the graph have the properties of a small-world or a scale-free network? What is its diameter, mean degree or average geodesic distance? Is it fully connected and if not what is its component size distribution? How does the existence of a particular vertex affect the properties of other vertices or the entire graph? What is its betweenness and how many hops does it take to reach any other vertex from the given source node? These are merely a few examples out of a vast number of metrics that can be computed to analyse a graph structure.

In this chapter only two of these graph metrics are discussed and a number of approaches to implementing them efficiently on parallel hardware for fast execution are proposed. These algorithms are of particular interest to this thesis. The following section focuses on the well known component labelling problem, which is used in the following chapters both as an integral part of a larger simulation algorithm and to analyse how the component size distribution of a system changes over the course of the simulation. The second algorithm is discussed in Section 6.2. It computes the clustering coefficient as described in Section 2.5, which can be used as part of the classification process for small-world graphs.

6.1. Component Labelling

Graph component labelling is an important algorithmic problem in many application areas. Common use cases include object recognition in image analysis applications or the detection of disconnected groups in a population that is modelled by a structured graph. How the connectivity of nodes in the system is determined is application specific. It can simply depend on the existence of an edge that models some kind of relationship between the vertices, or it may require that any number of properties on the nodes have the same or related values.

Some scientific simulation algorithms, like the Wolff cluster update method discussed in Section 7.2, have to repeatedly determine the nodes that belong to a particular subset of the system or label all the components found in the entire population to figure out the number of disconnected clusters and to find the giant component if one exists. The cluster size distribution by itself is an interesting metric that can give some insights into a particular system, as demonstrated in Section 7.4. The labelling phase can be responsible for a substantial fraction of the simulation or analysis time and a fast implementation is therefore of much value.

Component labelling or the more specific case of graph colouring – which gets its name from the process of assigning "colours" to nodes such that all the nodes in a connected cluster receive the same colour value – is not a new idea. A number of component labelling algorithms for a variety of systems, including parallel architectures and distributed memory computers, have been proposed [124,125] before. Application specific algorithms, for example for lattice physics simulation models [126, 127], have also been described in the applications literature.

This section[1] describes a number of different approaches to labelling the components of an arbitrary graph on the GPU using CUDA and discusses the strengths and weaknesses of each approach. This is particularly valuable if the simulation or analysis code that makes use of the labelling algorithm is executed on the GPU as well. Sequential and parallel CPU implementations are used as reference for the performance of the CUDA kernels. The graphs are arbitrary in the sense that their structure is not known beforehand. Examples are random graphs and complex networks as discussed in Sections 2.2 and 2.3. These graphs require a data structure that makes no assumptions about the vertices and how they are connected to each other. In particular, the data structure needs to support arbitrarily long – and potentially widely differing – adjacency-list lengths for each node.

6.1.1. Data Structures

An arbitrary graph can be represented in different ways in memory. The data structure can have a critical impact on the performance of the labelling algorithm. Different hardware architectures with diverse memory hierarchies and processing models often require different data structures to maximise performance. The component labelling implementations described here are restricted to undirected graphs, but the data structures and algorithms can easily be adapted to work with directed graphs as well. Figure 6.1 illustrates the data layout used to represent an arbitrary graph in the main memory of the host system or the device memory of the graphics card.

The graph representations in device memory are designed so that the CUDA kernels can maximise the usage of the available memory bandwidth. One important way to do this is to reduce the number of non-coalesced global memory transfers. This is a challenge for arbitrary graphs on CUDA devices, as the structure of the graph is not known beforehand.

6.1.2. CPU Implementations - Sequential & TBB

This section discusses the single- and multi-threaded CPU implementations. The sequential code is used as the reference to which the performance and scaling of the parallel algorithms is compared. The pseudo-code in Algorithm 22 describes the execution steps.

The algorithm first assigns a unique colour to all vertices. It then iterates over the vertex set V and starts the labelling procedure for all vertices $v_i \in V$ that have not been labelled yet as indicated by the frontier array F. The labelling procedure iterates over the arc set $A_i \subseteq A$ of vertex v_i, comparing its colour value c_i to that of its neighbours. If it finds that the colour value c_j of a neighbour v_j is greater than c_i, then it sets c_j to the value of c_i and recursively calls the labelling procedure for v_j, assuming that the maximum recursion depth R_{max} has not been reached. If it has been reached, however, then vertex v_j is written to list P, thus indicating that, later on, the process has to be picked up again beginning with this vertex. This is done to ensure that the application does not fail because it runs out of stack space, which could otherwise happen easily when processing a large component in this fashion.

If, on the other hand, the colour value c_j is lower than c_i, then the algorithm sets the colour c_i to the value c_j and goes back to the beginning of adjacency-list A_i to iterate over it once again. When the labelling

[1] This section extends on work first published in [128] K. A. Hawick, **A. Leist**, and D. P. Playne, "Parallel Graph Component Labelling with GPUs and CUDA," *Parallel Computing*, vol. 36, no. 12, pp. 655–678, December 2010.

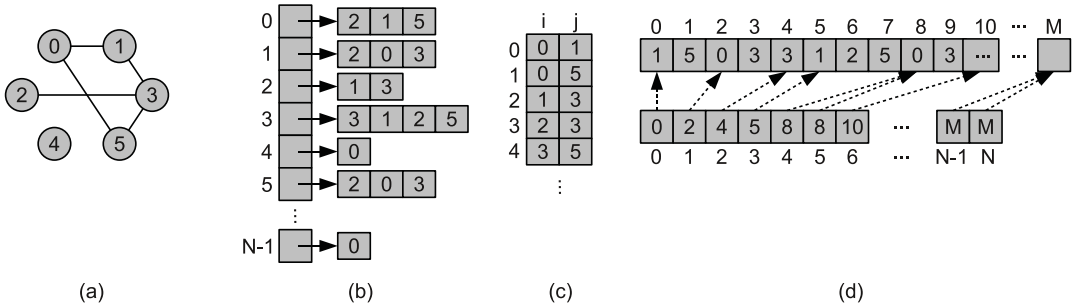


Figure 6.1.: The data structures used to represent a graph by the CPU and GPU implementations. All examples illustrate the same undirected graph shown by **(a)**. Figure **(b)** shows the adjacency-list representation used by the CPU algorithms. Every vertex $v_i \in V$ stores a list of its neighbours at index i of the vertex array. The first element of each adjacency-list is its length $|A_i|$, also called its out-degree k_{i_out} or simply degree k_i. The following elements are the indices of the adjacent vertices, that is the arc set $A_i \subseteq A$. To represent an undirected edge, each one of its end points needs to store the respective neighbour. **(c)** illustrates the edge set E used by Kernel 1. Every edge $e_{\{i,j\}} \in E$ is represented by a 2-tuple, which stores the IDs of the respective end points. **(d)** shows the vertex set V (bottom) and the arc set A (top) used by Kernels 2 and 3. Every vertex $v_i \in V$ stores the start index of its adjacency-list A_i at index i of the vertex array. The adjacency-list length $|A_i|$ can be calculated by looking at the adjacency-list start index of v_{i+1} ($V[i+1]-V[i]$). The vertex array contains $|V|+1$ elements so that this works for the last vertex too.

Algorithm 22 The single-threaded CPU implementation of the component labelling algorithm. Although the data structure used by this algorithm uses lists of directed arcs to represent the links between vertices, it assumes that if there is an arc (i, j) in the adjacency-list $A_i \subseteq A$ of vertex $v_i \in V$, then there is also an arc (j,i) in the adjacency-list $A_j \subseteq A$ of vertex $v_j \in V$. Thus, the algorithm essentially works on edges.

function LABEL_COMPONENTS_CPU(G)

Input parameters: The graph $G := (V,A)$ is an array of adjacency-lists, one for every vertex $v \in V$. The arc set A_i of a vertex v_i is stored in position $V[i]$. $|V|$ is the number of vertices in G and $|A_i|$ is the number of neighbours of v_i.
allocate $C[|V|]$ //$C[i]$ stores the colour assigned to vertex v_i
allocate $F[|V|]$ //flags used to indicate which vertices still need to be processed
allocate P //list of vertices that need to be visited again, initially empty
for all $v_i \in V$ **do**
  $C[i] \leftarrow i$ //initialise the colour array
  $F[i] \leftarrow$ true
end for
for all $v_i \in V$ **do**
  if $F[i] =$ true **then**
    call PROCESS_CPU($G,C,F,P,v_i,0$) //see Algorithm 23.
  end if
end for
while $P \neq \{\}$ **do**
  $v_i \leftarrow$ pop the next element from P
  if $F[i] =$ true **then**
    call PROCESS_CPU($G,C,F,P,v_i,0$) //process vertices in P
  end if
end while
return C //the colours array now contains the component ID for every vertex $v \in V$.

procedure reaches the end of this adjacency-list, all vertices that are reachable from v_i have been labelled with the same colour value. Once all vertices have been visited, the vertices added to list P are processed again in the same way as their colour may have been changed after their turn in the previous step. Once P is empty, the colour array C assigns one of N_C values in the range $[1,N_C]$ to each node, where N_C is the number of components in the graph.

Algorithm 23 Function PROCESS_CPU. R_{max} is the maximum recursion depth.

function PROCESS_CPU(G, C, F, P, v_i, r)
  $F[i] \leftarrow$ false
  for all $(i, j) \in A_i$ **do**
    if $C[i] > C[j]$ **then**
      $C[i] \leftarrow C[j]$
      re-run the for-loop from the beginning of adjacency-list A_i
    else if $C[i] < C[j]$ **then**
      $C[j] \leftarrow C[i]$
      if $r < R_{max}$ **then**
        call PROCESS_CPU($G, C, F, v_j, r+1$) //process v_j now
      else
        append v_j to P //maximum recursion level reached, process v_j later
      end if
    end if
  end for

Algorithm 24 The parallel CPU implementation of the component labelling algorithm uses TBB.

function LABEL_COMPONENTS_TBB(G)
  allocate $C[|V|]$ of type tbb::atomic<int>
  allocate $F[|V|]$ //flags used to indicate which vertices still need to be processed
  for all $v_i \in V$ **do**
    $C[i] \leftarrow i$ //initialise the colour array
    $F[i] \leftarrow$ true
  end for
  tbb::blocked_range $br(0, |V|)$ //initialise the iteration range for the parallel tasks
  repeat
    $m \leftarrow$ false
    do in parallel: call tbb::parallel_reduce(br, PROCESS_TBB(G, C, F, v_i)) //see Algorithm 25.
  until $m =$ false
  return C //the colours array now contains the component ID for every vertex $v \in V$.

The parallel CPU implementation, given in Algorithms 24 and 25, uses `tbb::parallel_reduce` to iterate over the vertex set V. One of the main differences between the sequential and parallel implementations is that the colours array C is now of type `tbb::atomic<int>` to support atomic `compare_and_swap` operations, which only update the colour with the value given in the first parameter if the current colour is equal to the value given in the second parameter. The function returns the old value, which is used to check if the update was successful. The TBB implementation does not use recursive calls and therefore has no need for list P. A parallel implementation with recursion has not achieved the same performance in tests as the algorithm described here. Access to the frontier array is not guarded in any way, which means that two threads may update it concurrently. While this may cause a vertex to be set as active in the frontier while it is being processed by another thread, it can not cause any inconsistencies, because a vertex v_i marking another vertex v_j as active in F always means that m is set to true as well and thus the entire procedure gets repeated, which ensures that the updated colour gets propagated as necessary.

6.1.3. The CUDA Implementations

Three different approaches to solving the component labelling problem with CUDA are discussed. As described in Figure 6.1, Kernel 1 uses a different data structure from Kernels 2 and 3. While Kernel 1 works on the edge set, executing one thread for every edge for a total of $|E|$ threads, Kernels 2 and 3 operate on the vertex set, executing one thread for every vertex for a total of $|V|$ threads.

Algorithm 25 Function `PROCESS_TBB`. This function is executed in parallel.

```
function PROCESS_TBB(G, C, F, v_i)
  for all v_i in range [br.begin(), br.end()) do
    if F[i] = true then
      F[i] ← false
      c_i ← C[i]
      for all (i, j) ∈ A_i do
        c_j ← C[j]
        while c_i > c_j do
          c_tmp ← C[i].compare_and_swap(c_j, c_i)
          if c_tmp = c_i then
            c_i ← c_j //successfully swapped the value in C
            re-run the for-loop from the beginning of adjacency-list A_i
          else
            c_i ← c_tmp //another thread changed v_i, try again with the updated value
          end if
        end while
        while c_i < c_j do
          c_tmp ← C[j].compare_and_swap(c_i, c_j)
          if c_tmp = c_j then
            c_j ← c_i //swap successful
            F[j] ← true
            m ← true //flag this run as modified
          else
            c_j ← c_tmp //another thread changed v_j, try again with the updated value
            if c_i > c_j then
              process the current neighbour v_j again
            end if
          end if
        end while
      end for
    end if
  end for
```

Edge-Based Kernel 1

In this approach, the colours array is initialised to a unique integer value for every vertex $v \in V$. Every thread loads one edge $e := \{i, j\} \in E$ from the edge set and compares the colours c_i and c_j of the two vertices v_i and v_j. If one of the colour values is smaller than the other, then the colour of the vertex with the higher colour value is updated. An `atomicMin` operation is used for this to ensure that simultaneous updates by different threads do not interfere with each other and do not rely on a particular ordering. In addition to updating the colour, the thread also sets a flag that indicates that there are still changes being made to the vertex colours. This last operation can also cause simultaneous writes to the same address by multiple threads. However, CUDA guarantees that at least one of them will succeed, which is all that is needed in this case. The kernel is called until no more changes are being made. The pseudo-code describing this process is given in Algorithms 26 and 27.

Performance considerations: The performance of the algorithm depends strongly on the graph diameter d – the longest of the shortest paths connecting any two vertices within the same component of the graph – since the colour needs d steps to propagate between all vertices. However, this does not mean that the kernel has to be called d times, as the $|E|$ threads are not actually all running at the same time. Changes to the colour of a vertex v_i made while processing an edge $\{i, j\}$ can already be visible when processing an edge $\{i,k\}$ at a later time during the same kernel call. The diameter is thus the upper bound for the number of kernel calls. If the graph consists of more than one component, then the diameter of the component with the largest value for d is relevant.

CUDA specific optimisations: The kernel uses a boolean flag in shared memory to indicate if any changes

Algorithm 26 Kernel 1: The first approach to labelling the components of an arbitrary graph in parallel on a CUDA-enabled device. This kernel operates on the edge set E, running one thread for every edge for a total of $|E|$ threads per kernel call.

function LABEL_COMPONENTS_KERNEL1_HOST(E,N)
  Input parameters: E is the list of all edges in a graph $G := (V,E)$. $N = |V|$ is the number of vertices.
  allocate $E_d[|E|]$ in device memory //the edge list
  allocate $C_d[N]$ in device memory //the colours array
  copy $E_d \leftarrow E$ //copy the edge list to the device
  do in parallel on the device using N threads: initialise $C_d[1 \ldots N]$ such that $C_d[i] \leftarrow i$
  allocate m_d in device memory //indicates if the labels are still changing
  repeat
    copy $m_d \leftarrow$ false
    do in parallel on the device using $|E|$ threads: call LABEL_COMPONENTS_KERNEL1(E_d, C_d, m_d)
    copy $m \leftarrow m_d$
  until $m =$ false //repeat until the colours are not changing any more
  copy $C \leftarrow C_d$ //get the colours from device memory
  return C //C now contains the component ID for every vertex $v \in V$.

Algorithm 27 The CUDA kernel called by Algorithm 26.

function LABEL_COMPONENTS_KERNEL1(E_d, C_d, m_d)
  $t \leftarrow$ thread ID queried from the CUDA runtime
  $e_t \leftarrow E_d[t]$ //read the edge $e_t := \{i, j\} \in E_d$ from global memory
  $c_i \leftarrow C_d[i]$ //read the colour c_i of vertex v_i from global memory
  $c_j \leftarrow C_d[j]$ //read the colour c_j of vertex v_j from global memory
  if $c_i < c_j$ **then**
    call atomicMin($C_d[j], c_i$)
    $m_d \leftarrow$ true
  else if $c_j < c_i$ **then**
    call atomicMin($C_d[i], c_j$)
    $m_d \leftarrow$ true
  end if

have been made by one or more threads in the same thread block. If this flag is set to true once all threads in the block have completed their work, then the first thread in the block sets the respective flag in global memory. This optimisation reduces the number of global memory writes and replaces them with much faster writes to shared memory.

Vertex-Based Kernel 2

Kernel 2, just like Kernel 1, initialises the colours array to a unique integer value for every vertex $v \in V$. Every thread t_i then loads the adjacency-list start index of vertex v_i into shared memory. This allows all but the last thread in the block to calculate the adjacency-list length $|A_i|$ of their vertex without another read from global memory. Every thread then checks if its vertex is flagged as active in the first frontier array $F1_d$, which means that this is either the first kernel iteration (all vertices are flagged) or that its colour has changed in the previous kernel call and therefore needs to be compared to the colour values of its neighbours. The next step for all threads that are flagged is to load the current colour c_i of their vertex. Now they can iterate through their adjacency-lists, comparing c_i with the colour c_j of every one of their neighbours v_j and updating either one of the colours if its value is higher than that of the other vertex. The kernel is called until no more changes are being made. Algorithms 28 and 29 describe this process.

Performance considerations: The performance of this algorithm depends strongly on the graph diameter d, as the colour needs d steps to propagate between all vertices. Again, this does not mean that d iterations are needed for the colour to spread through the connected components.

CUDA specific optimisations: In addition to the shared memory flag already used in Kernel 1, this kernel uses an array in shared memory to store the adjacency-list start indices of the threads within the same

thread block as described before. Furthermore, it uses a one-dimensional texture reference to iterate over the arc set A. Texture fetches are cached and can potentially exhibit higher bandwidth as compared to non-coalesced global memory reads if there is locality in the texture fetches. And since a thread t_i iterates over $|A_i|$ consecutive values and thread t_{i+1} iterates over the directly following $|A_{i+1}|$ consecutive values and so on, this locality is given.

The implementation uses one texture reference for the arc set. As the maximum width for a texture reference bound to linear memory is 2^{27} for all currently available CUDA devices (compute capability ≤ 2.1 [90]), it can only process graphs with up to 2^{27} arcs or half as many edges (each edge is represented by two arcs). The maximum size of the arc set is thus 512 MB (4 bytes per arc). As most modern graphics cards provide more than 512 MB of device memory, they could be used to process larger graphs if it was not for this limitation. In that case, two or more texture references could be used to access the arc set when necessary. It is important to ensure that the size of the arc set does not exceed this limit, as the CUDA libraries do not give any warning or failure message when trying to bind more elements than supported to the texture reference. The kernel appears to complete successfully and only the incorrect results, if spotted, show that something went wrong. This is a commonly encountered problem when using CUDA and requires particular care from the programmer.

Every thread in Kernel 2 iterates over the adjacency-list of one vertex, which causes warp divergence if the degrees of the vertices that are being processed by the threads of the same warp are not the same. Due to the SIMD data model – or SIMT in CUDA terminology – all threads in a warp have to do k_{max} iterations, where k_{max} is the highest degree processed by this warp. This problem can be mostly eliminated by sorting the vertices by their degree. However, the time needed to sort the vertices on the CPU and the overhead in the CUDA kernel, which comes from the fact that the vertex ID is no longer equal to the thread ID and therefore has to be looked up from device memory, are relatively large relative to the overall execution time. Tests have shown that this technique can slightly improve the kernel execution times when processing scale-free graphs with their "fat tail" degree distributions, but the overall execution time, which includes the time required to sort the vertices, increases. These findings are in line with previous results reported in [30]. Consequently, the implementation of Kernel 2 reported here does not sort the vertices. Nevertheless, other graph algorithms may benefit from this technique, for example when they execute the same kernel multiple times without changes to the graph structure or if the body of the diverging loop contains more instructions than it does in this case.

Breadth-First Search Kernel 3

This approach is based on the CUDA algorithm for a breadth-first search proposed by Harish and Narayanan [29]. Instead of calculating the cost to reach all vertices in the same component as a given source vertex, however, the implementation described here assigns the same colour label to all vertices in the same connected component for all components in the graph. Like Kernel 2, the implementation uses two frontier arrays, one for the current iteration and one for the following iteration, which are swapped between kernel calls to avoid concurrent updates that can lead to race-conditions.

Kernel 3 initialises the colours array to 0, which indicates that the respective vertex has not been coloured yet. The breadth-first search starts from a source vertex $v_{src} \in V$ with a new colour value c_{src} and spreads this colour in the component $C_i \subseteq V$, where $v_{src} \in C_i$, until all reachable vertices have been labelled. During iteration $s = \{1,2,3,\dots\}$ all vertices that are connected to v_{src} through a shortest path of $s-1$ edges are coloured with value c_{src} and the vertices at distance s are flagged as active in the frontier array. Only vertices that are flagged as active are processed during an iteration of the kernel. A new breadth-first search is started for every vertex that has not been coloured during one of the previous searches and is thus in a different, disjoint component.

Algorithm 28 Kernel 2: The second approach to labelling the components of an arbitrary graph in parallel on a CUDA-enabled device. This kernel operates on the vertex set V, running one thread for every vertex for a total of $|V|$ threads per kernel call. Like the CPU implementation, the kernel operates on a set of arcs but assumes that for every arc (i, j) an arc (j, i) exists as well.

function LABEL_COMPONENTS_KERNEL2_HOST(V,A)

Input parameters: The vertex set V and the arc set A describe the structure of a graph $G := (V,A)$. Every vertex $v_i \in V$ stores the index into the arc set at which its adjacency-list A_i begins in $V[i]$. The adjacency-list length $|A_i|$ is calculated from the start index of vertex v_{i+1} ($V[i+1]-V[i]$). To make this work for the last vertex $v_N \in V$, the vertex array contains one additional element $V[N+1]$. This additional element is not included in the number of vertices $|V|$. $|A|$ is the number of arcs in G.
allocate $V_d[|V|+1], A_d[|A|], C_d[|V|]$ in device memory
allocate $F1_d[|V|], F2_d[|V|]$ in device memory //$F1_d[i]$ indicates if a vertex v_i is to be processed in the current iteration //and $F2_d[i]$ indicates if it is to be processed in the next iteration
copy $V_d \leftarrow V, A_d \leftarrow A$ //copy the vertex and arc sets to the device
do in parallel on the device using $|V|$ threads: init. $C_d[1 \dots |V|]$, $F1_d[1 \dots |V|]$, $F2_d[1 \dots |V|]$ such that $C_d[i] \leftarrow i$, $F1_d[i] \leftarrow$ true and $F2_d[i] \leftarrow$ false
allocate m_d in device memory //indicates if the labels are still changing
repeat
  copy $m_d \leftarrow$ false
  do in parallel on the device using $|V|$ threads: call LABEL_COMPONENTS_KERNEL2($V_d, A_d, C_d, F1_d, F2_d, m_d$)
  $F1_d \leftrightarrow F2_d$ //swap the frontier arrays for the next iteration
  copy $m \leftarrow m_d$
until $m =$ false //repeat until the colours are not changing any more
copy $C \leftarrow C_d$ //get the colours from device memory
return C //C now contains the component ID for every vertex $v \in V$.

Algorithm 29 The CUDA kernel called by Algorithm 28.

function LABEL_COMPONENTS_KERNEL2($V_d, A_d, C_d, F1_d, F2_d, m_d$)

$i \leftarrow$ thread ID queried from the CUDA runtime //thread i processes vertex v_i
if $F1_d[i] =$ true **then**
  $F1_d[i] \leftarrow$ false
  $idx_i \leftarrow V_d[i]$ //the start index of A_i
  $idx_{i+1} \leftarrow V_d[i+1]$ //the start index of A_{i+1} (the exclusive upper bound for v_i)
  c_i, c_j //the colours of vertices v_i and v_j
  $c_i \leftarrow C_d[i]$ //read the colour of vertex v_i from global memory
  $c_{i(mod)} \leftarrow$ false //local variable to indicate if c_i has been changed
  for all $j \in A_i := \{A_d[idx_i], \dots, A_d[idx_{i+1}-1]\}$ **do**
    $c_j \leftarrow C_d[j]$ //read the colour of the adjacent vertex j from global memory
    if $c_i < c_j$ **then**
      call atomicMin($C_d[j], c_i$)
      $F2_d[j] \leftarrow$ true //v_j will be processed in the next iteration
      $m_d \leftarrow$ true
    else if $c_i > c_j$ **then**
      $c_i \leftarrow c_j$
      $c_{i(mod)} \leftarrow$ true
    end if
  end for
  if $c_{i(mod)} =$ true **then**
    call atomicMin($C_d[i], c_i$) //c_i was changed and needs to be written to global memory
    $F2_d[i] \leftarrow$ true //v_i will be processed again in the next iteration
    $m_d \leftarrow$ true
  end if
end if

Performance considerations: It takes at most $d+1$ iterations to label all vertices of a connected component, where d is the diameter of the component. This is the worst case scenario, which only occurs when v_{src} happens to be one of the end vertices of the shortest path between two vertices with geodesic distance d. Furthermore, it takes $|C|$ breadth-first searches to label the graph, where $|C|$ is the number of disjoint

Table 6.1.: This table summarises the different component labelling algorithms.

CPU Sequential	Operates on the vertices and their adjacency-lists. Uses a frontier array to keep track of vertices that need to be processed. Recursively visits neighbouring vertices with a higher colour value than that of the current vertex, up to a maximum recursion depth.
CPU TBB	Uses `tbb:parallel_reduce` to iterate over vertices and their adjacency-lists in parallel. The colours are stored using `tbb::atomic<int>` values, which can be updated atomically with `compare_and_swap` operations.
CUDA Kernel 1	Operates on the edge set. Compares the colour values of pairs of vertices and updates the larger value using `atomicMin`.
CUDA Kernel 2	Operates on the vertices and their adjacency-lists. Uses a frontier array to flag vertices whose colour has been been changed in one iteration to be processed in the next iteration. When being processed, a vertex compares its colour value to those of its neighbours and updates the larger value using `atomicMin`.
CUDA Kernel 3	Performs one breadth-first search for every connected component in the graph. Starting with a single vertex, it spreads the colour to all other vertices in the same component, processing all vertices at distance $s-1$ from the source during iteration s.

components. For this reason, the kernel performs better when the number of components is very small and the degree is uniformly large, as this means that the number of active vertices processed during an iteration increases more quickly than for a smaller degree, utilising more of the processing units available in the device. However, it turns out that Kernel 3 performs much worse than the other CUDA implementations for almost every type of graph, which is to be expected considering the large number of iterations needed to label a component and the relatively small number of vertices processed per iteration. For this reason, no pseudo-code is given for this algorithm.

6.1.4. Performance Results

This section compares the performance of the CUDA kernels to each other as well as to the CPU implementations. The measurements do not include the time needed to copy the graph data from host memory to device memory, as it is assumed that in a real scenario the component labelling algorithm would be used as an integral part of a simulation, as discussed in Section 7.2, or at least as part of a larger suit of analysis tools. The graph data is therefore expected to already reside on the device. The algorithms approach the task of labelling the vertices in a graph in terms of their membership to a particular connected component in very different ways, as highlighted by the overview given in Table 6.1. A variety of network structures are used for the performance measurements to fairly compare the algorithms and show their strengths and weaknesses.

Disjoint This algorithm generates graphs with a specified number of major components. Vertices are assigned to these sets at random. Edges are created between pairs of randomly selected vertices from the same set. This is a variation of the Erdős-Rényi $G_{n,m}$ model (Section 2.2) of generating random graphs that differs in the initial partitioning of the vertices into different sets. A graph generated with this algorithm has at least the specified number of components. It can have more components, but most of the additional components consist of a single disconnected vertex. This is due to the random way in which edges are generated and is more likely to happen when the number of edges in relation to the number of vertices is small.

Three different tests are performed with this algorithm, listed in Table 6.2 as Disjoint 1, 2 and 3. The first one varies the number of vertices $|V|$ in the graph, while keeping the number of edges ($|E| = 50,000,000$) and major components ($|C| = 100$) constant. The second one varies the number of

Table 6.2.: This table lists the properties of the graphs used to analyse the performance of the different labelling algorithms. Disjoint 1 and 2 have 100 major components that have been generated explicitly, but they also have a number of small components – often only consisting of a single disconnected vertex – that are a result of the random way in which edges are generated. The line, Barabási scale-free and Watts-Strogatz (W-S) small-world graph instances consist of a single component of size $|V|$. They show how well the algorithms perform when they are used to verify that a graph is fully-connected. The abbreviation 'm' stands for "millions".

Graph type	Vertices (m)	Edges (m)	Average degree	Major components
Disjoint 1	$2-16$	50	$6.3-50$	100
Disjoint 2	1	$10-60$	$20-120$	100
Disjoint 3	5	60	24	$1-1$m
Line	$0.2-1$	$0.2-1$	2	1
Scale-free	$1-5$	$5-25$	10	1
W-S, $p=0.01$	$2-14$	$10-70$	10	1
W-S, $p=0.1$	$2-14$	$10-70$	10	1
W-S, $p=1.0$	$2-14$	$10-70$	10	1

edges $|E|$, while keeping the number of vertices ($|V| = 1,000,000$) and major components ($|C| = 100$) fixed. And Disjoint 3 varies the number of major components ($|C| = \{10^0, 10^1, \ldots, 10^6\}$), while keeping the number of vertices ($|V| = 5,000,000$) and the number of edges ($|E| = 60,000,000$) constant.

Line A line graph is a simple one-dimensional graph where every vertex is connected to the vertex before and after itself. The first and last vertex are the only exceptions with one neighbour each. A line or chain is the graph with the highest possible diameter $d = |V| - 1$. As mentioned before, the performance of the CUDA algorithms depends heavily on the graph diameter. The line graph is thus the worst case scenario for them.

Scale-free networks have a power-law degree distribution, where a small number of vertices – the hub nodes – have a very large degree, but the majority of vertices have a relatively small degree. The algorithm used here implements the Barabási-Albert model of gradual growth and preferential attachment discussed in Sections 2.4 and 5.2.

Watts-Strogatz The Watts-Strogatz (W-S) β-model, described in Section 2.6, takes a regular one-dimensional graph where every vertex is connected to its k nearest neighbours and randomly rewires a fraction p of the edges. Depending on the value of p, the properties of the resulting graph range from large-world network to random graph.

The remainder of this section discusses the time needed to label the components of the different graph types. All data points are the averages of 20 measurements. Each one of these measurement uses a randomly seeded but otherwise equivalent graph instance as input to the labelling algorithm. Error bars representing the standard deviations are displayed but in most cases smaller than the symbol size. With the exception of the plots that present the results for the line graphs, no measurements are given for CUDA Kernel 3. The breadth-first search based algorithm is slower by at least an order of magnitude compared to the slowest of the given results in all other experiments.

Figure 6.2 shows the performance results for graphs of type Disjoint 1, where the number of vertices is varied from 2×10^6 to 1.6×10^7. The number of major components and edges is kept constant, which means that the average degree decreases with increasing system size. The GPUs clearly do not reach their full potential when processing the smallest of the tested graph instances, with the exception of the GTX260 using Kernel 1. It is interesting to observe that Kernel 1, with its larger number of threads compared to Kernel 2 – $|E|$ as opposed to $|V|$ – copes better with a smaller number of vertices, but is more affected

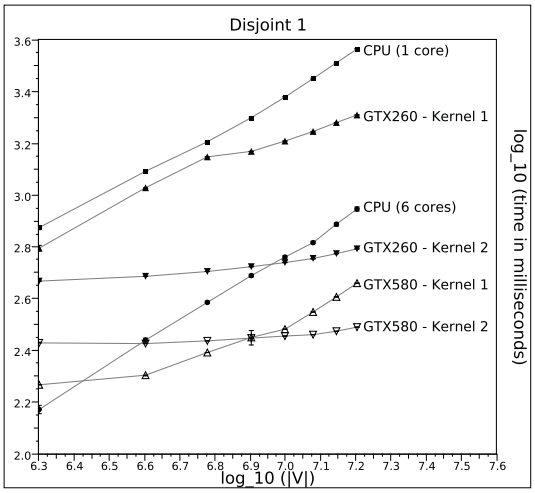


Figure 6.2.: Performance when processing graphs of type Disjoint 1.

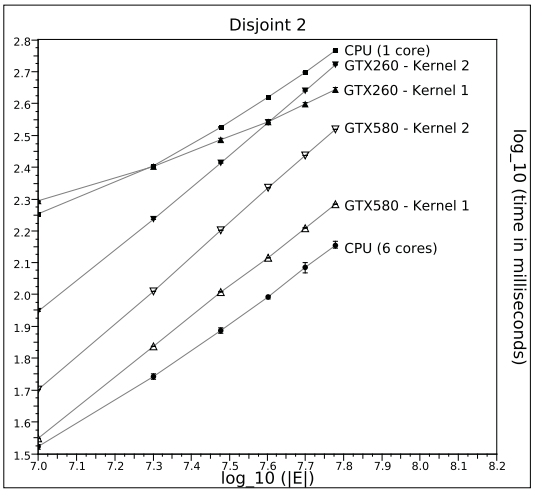


Figure 6.3.: Performance when processing graphs of type Disjoint 2.

by an increase in the system size, even though the number of edges remains constant. The reason for this is that, with increasing system size, the kernel needs to be called more often until the colour values stop changing. Kernel 2, on the other hand, is only marginally affected by the increase in the number of vertices, which can be attributed to the average degree, which decreases as an inverse function of the system size. The CPU implementations reach their full potential for much smaller systems, so that the multi-threaded implementation performs better than the GPU algorithms for the smallest of the tested graph instances. But it scales much more slowly with the increasing number of vertices and is significantly slower than Kernel 2 on the GTX580 when processing the largest graph instance.

The performance measurements for graphs of type Disjoint 2 are given in Figure 6.3. Here the number of vertices and major components is kept constant, while the number of edges and thus the average degree is increased. Kernel 1 clearly copes better with a large degree than Kernel 2, which is slowed down by the increased number of iterations when processing the adjacency-lists and the, in absolute numbers, larger deviations from the mean degree. While Kernel 1 is consistently faster than Kernel 2 on the GTX580, it is interesting to observe that the footprint of Kernel 2 appears to be significantly smaller than that of Kernel 1 on the GTX260 and a very large number of edges is needed to reach the point where Kernel 1 takes less time to complete than Kernel 2. The multi-threaded CPU implementation running on the Intel Core i7 970 is in its full element and completes the task in the least amount of time.

Figure 6.4 presents the results for graphs of type Disjoint 3. This test shows how much the algorithms are affected by the number of components, while the actual system size remains the same. As the number of major components is increased, the diameter of each individual component decreases on average. As discussed before, the diameter of the components is of significance and the results show that this is true for both the GPU and CPU implementations. The performance of Kernel 1 on the GTX260 improves relatively consistently as the graph diameters decrease, with the exception of the last data point. The same algorithm on the GTX580, as well as Kernel 2 on either GPU and also the multi-threaded CPU implementation, benefit much less from the decreasing diameters, except for the last data point where the number of components is the highest and the execution times drop significantly. It is likely that the opposite behaviour of Kernel 1 on the GTX260 compared to the GTX580 when the number of components is at its maximum has to do with caching. The average component consists of only 5 vertices when the number of components is 10^6. Therefore, multiple edges between the same vertices become the rule instead of the exception. The L1/L2 caches only available on the Fermi-architecture based GTX580 can be very effective when the colour values

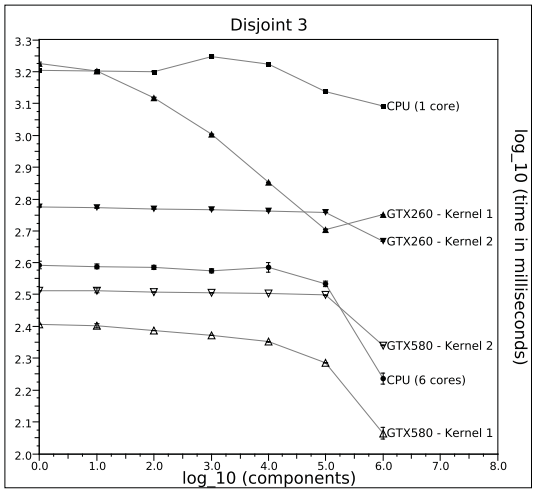


Figure 6.4.: Performance when processing graphs of type Disjoint 3.

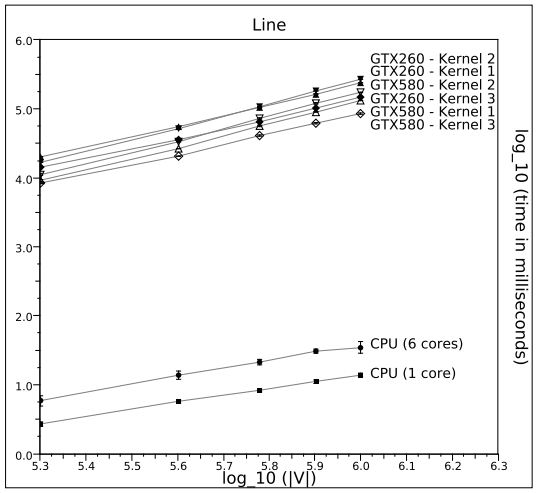


Figure 6.5.: Performance when processing a chain of vertices.

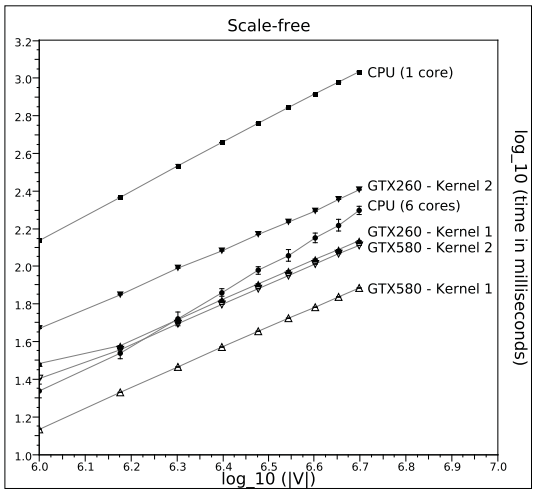


Figure 6.6.: Performance when processing Barabási-Albert scale-free graphs.

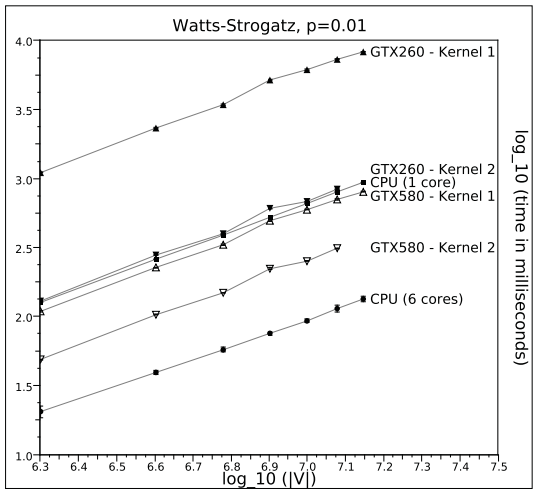


Figure 6.7.: Performance when processing Watts-Strogatz graphs with $p = 0.01$.

of the same vertices are compared repeatedly as the edges connecting them are processed.

The line graph is the worst case scenario not only for the GPU algorithms, but also for the TBB implementation for multi-core CPUs. Figure 6.5 shows the results. On the contrary, it is the best case scenario for the recursive algorithm used by the sequential CPU implementation, as it can very quickly iterate through the chain of vertices. All other algorithms have to slowly propagate the smallest colour one vertex at a time and thus require close to $|V|$ iterations of the entire algorithm. A graph with a structure similar to this scenario is obviously very unlikely to occur by chance, but it has been included here as an example that simply can not be parallelised efficiently. If such a graph structure is used by design, then parallel algorithms are not the right choice to process it. Notably, Kernel 3 achieves the best performance out of the CUDA implementations. While it also has to spread the lowest colour ID from one vertex to the next over exactly $|V|$ iterations, it at least does not waste much time comparing the colours of other vertices in vain as only one vertex is active in its frontier during each kernel call.

Figure 6.6 shows the performance measurements with scale-free graphs as input to the algorithms. As can be expected, Kernel 1 deals better with the strongly varying degrees than Kernel 2 on the GPUs. Both

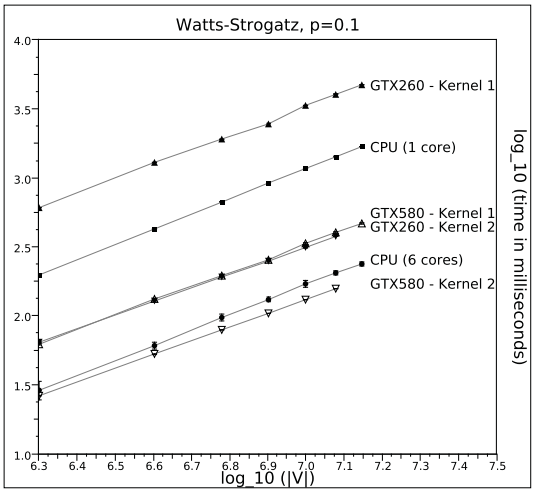


Figure 6.8.: Performance when processing Watts-Strogatz graphs with $p = 0.1$.

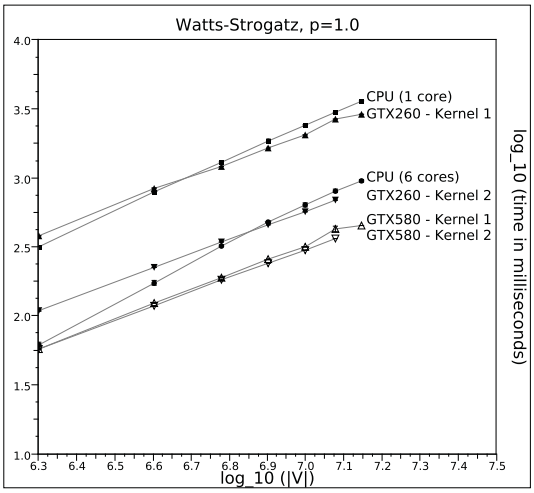


Figure 6.9.: Performance when processing Watts-Strogatz graphs with $p = 1.0$.

graphics devices used for the tests complete the tasks in less time than the TBB implementation when using Kernel 1 for all but the smallest of the tested graph instances. The slopes of the least square linear fits to the data sets, listed in Table 6.3, show that the execution times of the GPU algorithms are directly related to the system size. The multi-threaded CPU implementation scales well to the additional cores, completing the task of processing the largest of the measured graph instances about 5.5 times faster than the sequential implementation.

Figures 6.7, 6.8 and 6.9 show the results for Watts-Strogatz graphs with $p = 0.01$, $p = 0.1$ and $p = 1.0$ respectively. While graphs with $p = 0.01$ retain a largely regular structure, graphs generated with $p = 1.0$ can be considered as random for the purpose of this discussion. Contrary to the obvious intuition about the different hardware architectures, the performance of the CUDA algorithms changes much less from small to large p than the performance of the multi-threaded CPU implementation. In fact, $p = 0.01$ takes the longest to process when using Kernel 1 and comes in second when using Kernel 2. For both CPU implementations, on the other hand, there is a definite loss in performance with increasing rewiring probability. While the GPU architecture certainly is better at regular problems than at irregular ones, the algorithm needs to explicitly take advantage of that fact, which it does not do in this case, as the algorithms are optimised for complex graphs with arbitrary data structures. The CPU code is not optimised for the data locality in the more regular graph instances either, but automatically profits from it thanks to the advanced caching and data prefetching algorithms employed by today's processors.

While the TBB implementation performs better than the CUDA algorithms for $p = 0.01$, this changes for the larger values of p, where first the GTX580 and for the largest graph instances generated with $p = 1.0$ also the GTX260 are faster than the multi-core CPU when using Kernel 2. Kernel 1 is slower than Kernel 2 in all tests with Watts-Strogatz graphs. Kernel 2 benefits from the relatively small deviations from the mean vertex degrees in these graphs. The difference between the two kernels becomes rather small on the GTX580 for $p = 1.0$. No results for Kernel 2 are available for the largest graph instances with $|V| = 1.4 \times 10^7$ vertices, as these graphs exceed the size limit of $2^{27}/2$ for the edge set, which is imposed due to the maximum size of the texture reference as discussed in Section 6.1.3 on page 79.

Table 6.4 provides a concise overview of the performance results. The parallel implementations are compared to the sequential CPU implementation and the relative performance is reported. For each graph type, the fastest result is emphasised. Once again, there is no one best algorithm. The optimal choice depends strongly on the size of the vertex and edge sets as well as the general graph structure. The CPU is clearly bet-

Table 6.3.: The slopes of the least square linear fits to the data sets on a log-log scale. CPU (1) stands for the sequential implementation and CPU (6) for the TBB algorithm running on all six cores of an Intel Core i7 970. The numbers following the graphics devices represent implementation Kernel 1 and Kernel 2 respectively. For graphs of type Disjoint, not all data points are considered for the linear fit to improve the results. The first three data points are not included in the fit to Disjoint 1, the first data point is discarded for the fit to Disjoint 2 and the last data point is not taken into account for Disjoint 3.

Graph type	**CPU (1)**	**CPU (6)**	**GTX260 (1)**	**GTX260 (2)**	**GTX580 (1)**	**GTX580 (2)**
Disjoint 1	0.88	0.83	0.49	0.23	0.87	0.13
Disjoint 2	0.74	0.86	0.49	1.01	0.94	1.06
Disjoint 3	−0.01	−0.01	−0.14	0.00	−0.03	0.00
Line	1.00	1.16	1.53	1.74	1.74	1.78
Scale-free	1.29	1.40	1.06	1.05	1.06	1.04
W-S, $p = 0.01$	1.03	0.96	1.03	1.02	1.04	1.03
W-S, $p = 0.1$	1.10	1.09	1.05	0.99	0.99	0.99
W-S, $p = 1.0$	1.22	1.39	1.05	1.02	1.05	1.03

Table 6.4.: This summary lists the performance achieved by the parallel algorithms relative to the sequential CPU implementation. The largest of the measured graph instances of each type is used for this comparison, with the exception of the Watts-Strogatz graphs, where the second largest instance is used, as the largest graphs can not be processed by Kernel 2. For Disjoint 3, the results are for the graphs with the largest number of major components. CPU (1) stands for the sequential implementation and CPU (6) for the TBB algorithm running on all six cores of an Intel Core i7 970. The numbers following the graphics devices represent implementation Kernel 1 and Kernel 2 respectively.

Graph type	**CPU (1)**	**CPU (6)**	**GTX260 (1)**	**GTX260 (2)**	**GTX580 (1)**	**GTX580 (2)**
Disjoint 1	1.0	4.1	1.8	6.0	8.1	**11.9**
Disjoint 2	1.0	**4.1**	1.3	1.1	3.0	1.8
Disjoint 3	1.0	7.2	2.2	2.7	**10.7**	5.7
Line	**1.0**	0.4	0.00006	0.00005	0.0001	0.00008
Scale-free	1.0	5.5	7.9	4.2	**14.2**	8.4
W-S, $p = 0.01$	1.0	**7.1**	0.1	1.0	1.1	2.6
W-S, $p = 0.1$	1.0	7.0	0.4	3.8	3.5	**9.1**
W-S, $p = 1.0$	1.0	3.7	1.1	4.3	7.0	**8.2**

ter suited to process some specific cases, in particular when the structure is fundamentally sequential. The CPU can also more easily take advantage of data locality when the algorithm is not specifically optimised for it, which is mainly due to its larger cache size compared to the GTX580 and other recent GPUs. Older devices, like the GTX260, do not have a cache of this type at all. However, GPU code can be optimised greatly to explicitly utilise fast on-chip memory for such data locality when it is expected, as demonstrated in Sections 7.1 and 8.3.

Among the CUDA implementations, the edge-based Kernel 1 has the upper hand when the vertex degrees deviate significantly from the mean, as the concept of adjacency-lists does not exist in its implementation. When the vertex degrees are relatively similar, on the other hand, Kernel 2 generally performs better, because it only performs certain operations once, like loading the colour of the source vertex of an arc, compared to the, on average, $|A_i|/2$ times necessary for Kernel 1.

6.2. Clustering Coefficient

The clustering coefficient defined by Newman et. al. [54] is a graph metric that is commonly used when analysing social networks. It measures the network transitivity – the “fraction of transitive triples” [75]

in the network – or in other words, the probability of two connected nodes sharing a common neighbour, thus forming a cycle consisting of three nodes and three edges. The definition of the clustering coefficient is given in Algorithm 2.10 in Section 2.5. To compute the clustering coefficient, the number of triangles and connected triples in the graph have to be determined. This proves to be computationally demanding for large networks. Other authors have proposed computationally cheaper algorithms that approximate the clustering coefficient [129], but this chapter focuses on computing the exact result for the metric using highly parallel algorithms to reduce the execution time [2].

The clustering metric described by Newman et. al. only looks at elementary circuits of length three and thus reports a clustering coefficient of zero for graphs like a regular square lattice. However, unlike a simple ring, which also has a clustering coefficient of zero, a lattice has a highly local structure, only that this particular structure requires an additional intermediate node to connect two neighbours of a vertex. To measure such local structures not considered by the clustering metric, the clustering profile has been proposed by Abdo and Moura [130]. It answers the question of how closely related the neighbours of a node are. However, it makes it necessary to count circuits of length greater than three, which very quickly becomes infeasible in terms of memory or time complexity for large graphs even with efficient counting algorithms [119, 131, 132]. For example, Johnson's algorithm [119] for counting elementary circuits in directed graphs is bounded in time by $O((n+e)(c+1))$ for graphs with n vertices, e directed edges and c circuits, which is expensive since the number of circuits c grows very quickly with the network size and connectivity. Because the networks analysed in this thesis tend to be rather large, with hundreds of thousands of vertices and more, the work reported here focuses on the original clustering metric proposed by Newman et. al., which can be computed in a reasonable time for graphs of this size as Section 6.2.4 shows.

The clustering coefficient has been implemented and optimised for a number of different parallel hardware architectures: x86 multi-core CPU, GPU and CellBE. The performance of the algorithms on these heterogeneous platforms is compared using small-world and scale-free networks. Each one of the different hardware architectures uses a very different approach to parallelism and thus requires an algorithm that is specifically tailored for its architecture to achieve peak performance.

6.2.1. CPU - Sequential, PThreads & TBB

For reference purposes and to better explain the algorithm, a serial CPU code implementation of the clustering coefficient calculation is given in Algorithm 30. The outermost loop iterates over every vertex $v_i \in V$. The algorithm follows the links to neighbouring vertices in a depth-first search like fashion up to a maximum distance of three, where it can determine if a particular path ends at the source vertex v_i, thus forming a cycle of length three. The search can be aborted prematurely if it is clear that the current path can not lead to a valid cycle. This is particularly the case if any vertex other than the first and last is visited more than once.

If the entire process were to be performed for every vertex, then each cycle would be counted three times, once for each of its distinct vertices. This would obviously be inefficient and therefore the conditional statements in lines 12 and 18 guarantee that the path is only counted when processing the vertex with the smallest ID of the three. It would be equally correct to test for the vertex with the largest ID. While it does not matter which of the two approaches is used when the adjacency-list lengths of the graph do not follow a particular pattern, it does make a significant difference in the performance of the algorithm if the nodes

[2] This section extends on work first published in [92] K. A. Hawick, **A. Leist**, and D. P. Playne, "Mixing Multi-Core CPUs and GPUs for Scientific Simulation Software," *Research Letters in the Information and Mathematical Sciences*, vol. 14, no. ISSN 1175-2777, pp. 25–77, 2010 and in [114] **A. Leist**, K. A. Hawick, and D. P. Playne, "GPGPU and Multi-Core Architectures for Computing Clustering Coefficients of Irregular Graphs," in *Proc. International Conference on Scientific Computing (CSC'11)*, no. CSC2720, Las Vegas, USA, July 2011.

Algorithm 30 Pseudo-code for the sequential CPU implementation of the clustering coefficient.

```
function CLUSTERING(G)
 1: Input parameters: The graph G := (V,A) is an array of adjacency-lists, one for every vertex v ∈ V. The arc set A_i ⊆ A
    of a vertex v_i is stored in position V[i]. |V| is the number of vertices in G and |A_i| is the number of neighbours of v_i.
 2: R ← determine reverse adjacency-list lengths
 3: t ← 0 //triangle counter
 4: p ← 0 //paths counter
 5: for all v_i ∈ V do
 6:     for all v_j ∈ A_i do
 7:         if v_j = v_i then
 8:             p ← p − R[v_i] //correct for self-arcs
 9:             continue with next neighbour v_{j+1}
10:         end if
11:         p ← p + |A_j|
12:         if v_i < v_j then
13:             for all v_k ∈ A_j do
14:                 if v_k = v_i then
15:                     p ← p − 2 //correct for cycles of length 2 for both v_i and v_j
16:                     continue with next neighbour v_{k+1}
17:                 end if
18:                 if v_k ≠ v_j AND v_i < v_k then
19:                     for all v_l ∈ A_k do
20:                         if v_l = v_i then
21:                             t ← t + 1
22:                         end if
23:                     end for
24:                 end if
25:             end for
26:         end if
27:     end for
28: end for
29: return (3t)/p //the clustering coefficient
```

Algorithm 31 Pseudo-code for the multi-core CPU implementation of the clustering coefficient using PThreads.

```
function CLUSTERING(G)
  R ← determine reverse adjacency-list lengths
  mutex m // mutual exclusion for v_curr
  v_curr ← 0 //set the current vertex
  n ← number of CPU cores
  do in parallel using n threads: call PROCESS(G,R,v_curr,m)
  wait for all threads to finish processing
  t ← sum of triangles found by threads
  p ← sum of paths found by threads
  return (3t)/p //the clustering coefficient
```

at one end of the range tend to have a higher degree than the nodes at the opposite end. An example for such a case are the Barabási-Albert scale-free networks, where a node with a smaller ID is "older" than a node with a larger ID, which has the effect that it is likely to be more highly connected. The reason why this matters is that the number of times that the adjacency-list of the vertex with the highest degree has to be processed is minimised if the process is completed while the corresponding vertex is the source of the search and not one of the vertices visited during the search.

The iterations of the outermost loop do not interfere with each other and can thus be executed in parallel. It is merely necessary to sum up the numbers of triangles and paths found in each of the parallel iterations to get the total counts for the graph before the clustering coefficient can be calculated. The PThreads implementation for multi-core x86 CPUs, given in Algorithms 31 and 32, is therefore fairly straight forward.

Instead of using the low-level PThreads library, the algorithm can also be implemented using the TBB

Algorithm 32 Algorithm 31 continued. Function `PROCESS` runs in parallel.

function PROCESS(G, R, v_{curr}, m)
  $t \leftarrow 0$ //local triangle counter
  $p \leftarrow 0$ //local paths counter
  repeat
    acquire lock on mutex m
    $v_s \leftarrow v_{curr}$ //start vertex for this thread
    $v_e \leftarrow v_s +$ work block size //end vertex for this thread, must not exceed $|V|$
    $v_{curr} \leftarrow v_e$
    release lock on mutex m
    for all $v_i \in V_i \equiv \{v_s, \ldots, v_e\} \subseteq V$ **do**
      count triangles and paths as described for the sequential CPU implementation
    end for
  until $v_s \geq |V|$
  return $\{t, p\}$

Algorithm 33 Pseudo-code for the multi-core CPU implementation of the clustering coefficient using TBB.

function CLUSTERING(G)
  $R \leftarrow$ determine reverse adjacency-list lengths
  tbb::blocked_range $br(0, |V|, 10)$ //the iteration range and grain size for the partitioner
  do in parallel: call tbb::parallel_reduce(br, PROCESS_TASK(G, R), tbb::simple_partitioner())
  retrieve results and calculate the clustering coefficient

multi-tasking library. Again, the parallelism is applied to the outermost loop. TBB's `parallel_reduce` is used to obtain the path and triangle counts. The heuristic used to determine the chunk size by the `tbb::auto_partitioner` works well when each iteration takes approximately the same amount of time to complete. This is not the case when processing scale-free graphs, though. Tests have shown that the `tbb::simple_partitioner` with a small grain size works significantly better when processing scale-free graphs with the clustering algorithm, while maintaining nearly the same performance to the `tbb::auto_partitioner` for small-world graphs. The grain size C chosen here is 10. The chunk size generated by the `tbb::simple_partitioner` is in the range $[C/2, C]$.

Algorithm 33 shows how the full iteration range is defined and passed to `parallel_reduce`. TBB recursively splits the iteration range into sub-ranges until a certain automatically determined threshold is reached. Then TBB uses available worker threads to execute `PROCESS_TASK` (Algorithm 34) in parallel. When the two halves of a range have been processed, TBB invokes function `JOIN` (Algorithm 35) to combine the results. Eventually, all sub-ranges have been processed and the results have been joined into the root of the task tree. TBB returns and the results can be extracted from the root object.

Algorithm 34 TBB executes `PROCESS_TASK` in parallel.

function PROCESS_TASK(br, G, R)
  $t \leftarrow 0$ //task local triangle counter
  $p \leftarrow 0$ //task local paths counter
  $v_s \leftarrow br.begin()$ //start vertex
  $v_e \leftarrow br.end()$ //end vertex
  for all $v_i \in V_i \equiv \{v_s, \ldots, v_e\} \subseteq V$ **do**
    count triangles and paths as described in the sequential CPU implementation
  end for

Algorithm 35 TBB calls `JOIN` to combine the results of the two halves of a range.

function JOIN(x, y)
  Input parameters: x and y are task objects.
  $x.t \leftarrow x.t + y.t$
  $x.p \leftarrow x.p + y.p$

Algorithm 36 Pseudo-code for the CUDA implementation of the clustering coefficient. It operates on the arc set A, executing one thread for every arc $a_i \in A$ for a total of $|A|$ threads. Self-arcs are filtered out by the host as they never contribute to a valid triangle or path. The host program prepares and manages the device kernel execution.

function CLUSTERING(V, A, S)

Input parameters: The vertex set V and the arc set A describe the structure of a graph $G := (V, A)$. Every vertex $v_i \in V$ stores the index into the arc set at which its adjacency-list A_i begins in $V[i]$. The vertex degree $|A_i|$ is calculated from the adjacency-list start index of vertex v_{i+1} ($V[i+1] - V[i]$). In order for this to work for the last vertex $v_N \in V$, the vertex array contains one additional element $V[N+1]$. $|A|$ is the number of arcs in G. $S[i]$ stores the source vertex of arc a_i.

allocate $V_d[|V|+1], A_d[|A|], S_d[|A|], t_d, p_d$ in device memory
copy $V_d \leftarrow V$
copy $A_d \leftarrow A$
copy $S_d \leftarrow S$
$t_d \leftarrow 0, p_d \leftarrow 0$ //initialise the triangle and path counters in device memory
do in parallel on the device using $|A|$ threads: call KERNEL(V_d, A_d, S_d, t_d, p_d)
copy $t \leftarrow t_d$ //copy the triangle counter from device memory
copy $p \leftarrow p_d$ //copy the paths counter from device memory
return $(3t)/p$ //the clustering coefficient

6.2.2. GPU - CUDA

Due to the data-parallel architecture of today's GPUs and the low-level performance tuning necessary to achieve high performance on the accelerator, the CUDA implementation is much more complex than the multi-core CPU implementations. Arbitrary graphs, like small-world or scale-free networks, where the structure is not known beforehand, can be represented in different ways in graphics memory as demonstrated in the previous sections. The CUDA implementation of the clustering coefficient described here uses the same data structure as Kernels 2 and 3 of the component labelling algorithm. Two arrays, one for the vertices and one for the arcs, store the necessary information as illustrated in Figure 6.1 (d) on page 75.

A particular issue when processing arbitrary graphs with CUDA is that the adjacency-lists differ in length, and that it is often necessary to iterate over such a list of neighbours. And since in the SIMT architecture all 32 threads of a warp are issued the same instruction, iterating over the neighbours-lists of 32 vertices can cause warp divergence if these lists are not all of the same length. In the case of warp divergence, all threads of the warp have to execute all execution paths, which in this case means they all have to do x iterations, where x is the longest of the 32 adjacency-lists. And as described in the CPU implementation, the clustering coefficient algorithm uses nested loops, which make the problem even worse.

However, the outermost loop can be avoided when the implementation iterates over the arc set A instead of the vertex set V. This improves the performance of the CUDA kernel considerably and also changes the total number of threads from $|V|$ to $|A|$. $|A|$ is usually much larger than $|V|$, giving CUDA more threads to work with, which it can use to hide memory latencies and which also means that the implementation should scale better to future graphics cards with more processing units. Furthermore, since the algorithm only proceeds when the ID of the source vertex is smaller than that of its neighbours, as discussed before, the number of iterations of the inner loops and at the same time the amount of warp divergence are significantly reduced when processing scale-free graphs similar to those generated by the Barabási-Albert algorithm. The correct treatment of such graphs thus has an even bigger impact on the performance of the CUDA algorithm than it does on the performance of the CPU algorithm. The implementation is described in Algorithm 36.

The implementation described so far shall be referred to as Kernel 1. As the performance results in Section 6.2.4 show, this implementation performs well for graphs with only slightly varying vertex degrees, like Watts-Strogatz small-world networks. For graphs with a significant variance in the vertex degrees, like scale-free graphs with their power-law degree distributions, a small variation of this implementation is tested. In this second approach, referred to as Kernel 2, the input array S not only references the source

Algorithm 37 The device kernel called in Algorithm 36.

```
function KERNEL(V, A, S, t, p)
  i ← thread ID queried from CUDA runtime
  v_i ← S[i] //arc source
  v_j ← A[i] //arc end
  p ← p + |A_j|
  if v_i < v_j then
    for all v_k ∈ A_j do
      if v_k = v_i then
        p ← p − 2 //correct for cycles of length 2 for both v_i and v_j
        continue with next neighbour v_{k+1}
      end if
      if v_i < v_k then
        for all v_l ∈ A_k do
          if v_l = v_i then
            t ← t + 1
          end if
        end for
      end if
    end for
  end if
```

vertex of an arc, but also uses a second integer to store the end vertex. Furthermore, the host sorts this array by the degree of the arc end vertices before passing it to the CUDA kernel. Even though this means that the end vertex of each arc is stored twice, once in S and once in A, it makes it possible to process the arcs based on the vertex degrees of their end vertices, which determine the number of iterations done by the outer one of the two loops in the CUDA kernel. This means that threads of the same warp can process arcs with similar end vertex degrees, thus reducing warp divergence considerably. The sorting is done by the host using `tbb::parallel_sort`.

The GTX580 used for the performance measurements has an issue that, intermittently, causes the CUDA kernel to never return if it takes too long to complete. This is not the known execution time limitation that is active when the X window system is running on the same GPU as the kernel, as no X-server is active on the test systems. The issue is most likely driver related, as the author has been able to easily reproduce it on multiple GTX580 devices, but never on GTX480 devices – which have the same compute capability as the GTX580 – nor on previous generation GTX260 devices. To fix the problem on the GTX580, the number of threads per kernel call is limited to 30720 and the kernel is called repeatedly until the entire system is processed. The chosen limit for the number of threads works well when processing the graph instances used for the performance measurements, but different graphs with larger maximum vertex degrees may require an adjustment. A future driver update by NVIDIA will hopefully eliminate the issue altogether, as performance is slightly improved when the entire system can be processed in a single kernel call.

Further CUDA specific optimisations applied to both versions of the clustering kernel are shown in Algorithm 38. They include counting the triangles and paths found by each individual thread in its registers, before writing them to shared memory, where the total counts for a thread block are accumulated, which are eventually written to global memory with a single atomic transaction per counter and thread block. Furthermore, texture fetches are used when iterating over the adjacency-lists of vertices, taking advantage of data locality. And because the caching done when fetching the neighbours v_k of vertex v_j may be overwritten by the inner loop, a constant number of arc end vertices are pre-fetched and written to shared memory. This pre-fetching is only done for older devices like the GTX260, as the latest generation of Fermi-based NVIDIA GPUs, which includes the GTX580, provides automatic caching in L2 (shared by all multiprocessors) and L1 (on each multiprocessor) caches. The overhead of manually caching the data in shared memory can decrease the performance on these devices.

When multiple GPUs are available in the same host system, then it may be desirable to utilise all of them

Algorithm 38 CUDA performance optimisations.

```
// shared memory counters
__shared__ unsigned int nTrianglesShared;
__shared__ unsigned int nPaths2Shared;
if (threadIdx.x == 0) {
  nTrianglesShared = 0;
  nPaths2Shared = 0;
}
__syncthreads();

// each thread uses registers to count
unsigned int nTriangles = 0;
int nPaths2 = 0;
...
// NOTE: this explicit caching can be counter-productive on Fermi devices!
const int prefetchCount = 7;
__shared__ int nbr2Prefetch[prefetchCount* BLOCK_SIZE];
if (srcVertex > nbr1) {
  atomicAdd(&nPaths2Shared, (unsigned int)nPaths2);
} else {
  for (int nbr2Idx = 0; nbr2Idx < nArcsNbr1; ++nbr2Idx) {
    //pre-fetch nbr2 to shared mem. to take
    //advantage of the locality in texture fetches
    int nbr2;
    int prefetchIdx=nbr2Idx%(prefetchCount+1);
    if (prefetchIdx == 0) {//global mem. read
      nbr2 = tex1Dfetch(arcsTexRef, nbr1ArcsBegin+nbr2Idx);
      for (int i=0; i < prefetchCount; ++i) {
       nbr2Prefetch[i*blockDim.x+threadIdx.x] =
          tex1Dfetch(arcsTexRef, nbr1ArcsBegin+nbr2Idx+i+1);
      }
    } else { // read from shared memory
      nbr2 = nbr2Prefetch[(prefetchIdx-1) * blockDim.x+threadIdx.x];
    }
    ...
    for (int nbr3Idx=0; nbr3Idx < nArcsNbr2; ++nbr3Idx) {
      nTriangles +=
        tex1Dfetch(arcsTexRef, nbr2ArcsBegin+nbr3Idx) == srcVertex ? 1 : 0;
    }
  }

  // write local counters to shared memory
  atomicAdd(&nTrianglesShared,nTriangles);
  atomicAdd(&nPaths2Shared, (unsigned int)nPaths2);
}
// write to global mem (once per thread block)
__syncthreads();
if (threadIdx.x == 0) {
  atomicAdd(nTotalTriangles, (unsigned long long int)nTrianglesShared);
  atomicAdd(nTotalPaths2, (unsigned long long int)nPaths2Shared);
}
```

to further reduce the execution time of the algorithm. And because the iterations of the outermost loop are independent from each other with no need for synchronisation, the work can be distributed over the available GPUs in the same way as multiple CPU cores are utilised by threads. One PThread is created for every GPU and controls the execution of all CUDA related functions on this particular GPU. The data structure of the graph is replicated on all graphics devices and instead of executing $|A|$ CUDA threads to count all triangles and paths with just one kernel call, a work block of N arcs $\{a_i,\ldots,a_{i+N-1}\} \subseteq A$ is processed during each kernel call. A new work block is determined in the same way as it is done when using PThreads to execute

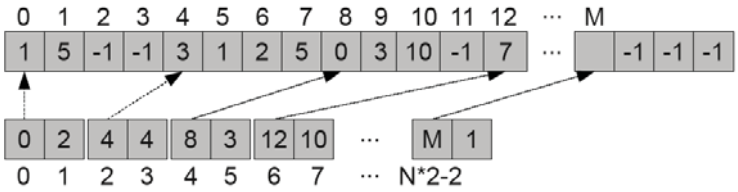


Figure 6.10.: The data structure used to represent the graph in system memory of the Cell BE. It shows the vertex set V (bottom) and the arc set A (top). Every vertex $v_i \in V$ stores the start index of its adjacency-list A_i at index $i \times 2$ of the vertex array. The adjacency-list length $|A_i|$ is stored at the next index. The vertex array contains $|V| \times 2$ elements. Every adjacency-list in the arcs array is padded to the next multiple of 16-bytes (4-bytes per value) in order to conform with the memory alignment requirements. The padding elements have the value -1, which is an invalid vertex ID.

on multiple CPU cores. The work block size depends on the available graphics hardware and the size of the thread blocks in the CUDA execution grid: $N =$ (number of threads per block)$\times$(blocks per streaming multiprocessor)$\times$(number of streaming multiprocessors). The goal is to make it large enough to allow CUDA to fully utilise the hardware and small enough to keep all available GPUs busy for roughly the same amount of time.

6.2.3. Cell Processor - PS3

Like the CUDA implementation, the implementation for the Cell Broadband Engine (BE) requires a lot of architecture specific tuning to achieve good performance. The memory layout used is similar to the one used by the CUDA kernels, using one array for the vertices and one array for the arcs. However, the requirement that the direct memory accesses used to transfer data from main memory to the local stores of the SPEs are aligned on 16-byte boundaries makes some changes necessary. See Figure 6.10 for an illustration and description of the memory layout.

The main task of the Cell's PPE is to manage the SPEs as illustrated in Algorithm 39. It is used to load the graph and store it in system memory using the memory layout described before. Then it initialises the SPEs, which do most of the actual computation (See Algorithms 40 and 41). However, the PPE would not be fully utilised if providing the SPEs with further work was all it did. Therefore, it performs some of the same computational tasks in its spare time, further improving the overall performance. The implementation of the triangle and paths counting algorithm on the PPE is basically the same as the single-threaded CPU implementation described in Algorithm 30, except that it uses the PPE's vector unit in the same way as the SPE implementation does in its innermost loop. These vector operations are described in Algorithm 42.

Traversing an arbitrary graph as it is done by the triangle and path counting algorithms requires many reads from unpredictable memory addresses. And since the local store of the SPEs with its 256KB capacity is relatively small, much too small to hold the entire graph structure of anything but very small graphs, it is necessary to load the required parts of the graph from system memory into local memory when needed. For example, when processing a certain vertex, then its adjacency-list has to be copied into the local store. This is done by issuing a DMA request from the SPU to its MFC (every SPE has one SPU and one MFC). However, the performance of the implementation would not be good if the SPU were to stall until the requested data becomes available. Instead, the implementation for the SPE is split into phases as illustrated by Figure 6.11 and Algorithm 41. A phase ends after a DMA request has been issued and the following phase, which uses the requested data, is not executed until the data is available. This implementation of multi-buffering uses 16 independent buffers to process the work block issued to the SPE. When a buffer

Algorithm 39 Pseudo-code for the Cell BE implementation of the clustering coefficient. This algorithm describes the tasks of the PowerPC Processor Element. It operates on the vertex set V, issuing blocks of vertices to the the Synergistic Processor Elements for processing, as well as processing small work chunks itself when it has nothing else to do. Self-arcs are filtered out beforehand, as they never contribute to a valid triangle or path.

```
function CLUSTERING(V,A)
  Input parameters: The vertex set V and the arc set A describe the structure of a graph G := (V,A). Every vertex v_i ∈ V
  stores the index into the arc set at which its adjacency-list A_i begins in V[i × 2] and its degree in V[i × 2 + 1]. |V| is
  the number of vertices in G. SPE = {spe_0, spe_1, ..., spe_5} is the set of SPEs.
  for all spe_i ∈ SPE do
    initialise spe_i and start processing a block of vertices
  end for
  while more vertices to process do
    for all spe_i ∈ SPE do
      if inbound mailbox of spe_i is empty then
        write the start and end vertices of the next work block to the mailbox
      end if
    end for
    process a small work block on the PPE
  end while
  for all spe_i ∈ SPE do
    send interrupt signal and wait until spe_i finishes processing
  end for
  aggregate results and calculate clustering coefficient
```

Algorithm 40 The pseudo-code for the SPE implementation of the clustering coefficient on the Cell BE. See Algorithm 39 for the PPE implementation and Algorithm 41 for the different execution phases.

```
function CLUSTERING(v_s, v_e)
  Input parameters: Each SPE receives an initial work block [v_s, ..., v_e) ⊆ V of source vertices to process.
  copy initial data from system memory to the local store
  initialise buffers B = {b_0, b_1, ..., b_15}
  repeat
    v_curr ← v_s //initialise current vertex v_curr
    for all b_i ∈ B do
      b_i.phase ← phase1 //set the next phase of b_i
      mark buffer as "ready"
    end for
    //process the current work block
    set all buffers as active
    while at least one buffer is active do
      b ← any "ready" buffer
      call b.phase //execute the next phase of b
    end while
    //check if there is more work to do
    v_s ← read next value from inbound mailbox
    if no interrupt signal received (v_s ≠ −1) then
      v_e ← read next value from inbound mailbox
    end if
  until interrupt signal received
  copy the results back to system memory
```

is waiting for data, the implementation switches to another buffer that is ready to continue with the next phase.

The Cell PPE and SPE units all have their own vector units and 128-bit wide vector registers. This allows them to load four 32-bit words into a single register and, for example, add them to four other words stored in a different register in a single operation. A program for the Cell BE should be vectorised where possible to fully utilise the available processing power. Algorithm 42 describes how the innermost loops of the PPE and SPE implementations make use of the vector units.

Algorithm 41 Algorithm 40 continued. The phases of the SPE implementation execute on a buffer b. Each phase models a step in the process of counting the triangles t and paths p. A phase ends after a DMA request to load data into local storage has been issued or when the end of a loop is reached.

```
function phase1(b)
  b.v_i ← v_curr //set the source vertex for this buffer
  v_curr ← v_curr + 1
  if b.v_i ≥ v_e then
    set buffer as inactive //end of work block reached
  else
    copy_async b.v_i.dat ← load adjacency-list info
    b.phase ← phase2
  end if
function phase2(b)
  copy_async b.A_i ← use b.v_i.dat to load A_i ⊂ A
  b.phase ← phase3
function phase3(b)
  if end of adjacency-list b.A_i reached then
    b.phase ← phase1 //loop condition not fulfilled
  else
    b.v_j ← next value in b.A_i
    copy_async b.v_j.dat ← load adjacency-list info
    b.phase ← phase4
  end if
function phase4(b)
  b.p ← b.p + |A_j|
  if b.v_j < b.v_i then
    b.phase ← phase3 //do not count triangle thrice
  else
    copy_async b.A_j ← use b.v_j.dat to load A_j ⊂ A
    b.phase ← phase5
  end if
function phase5(b)
  if end of adjacency-list b.A_j reached then
    b.phase ← phase3 //loop condition not fulfilled
  else
    b.v_k ← next value in b.A_j
    if b.v_k = b.v_i then
      b.p ← b.p − 2 //correct for cycles of length 2
      b.phase ← phase5
    else if v_k < v_i then
      b.phase ← phase5 //don't count triangle thrice
    else
      copy_async b.v_k.dat ← load adj.-list info
      b.phase ← phase6
    end if
  end if
function phase6(b)
  copy_async b.A_k ← use b.v_k.dat to load A_k ⊂ A
  b.phase ← phase7
function phase7(b)
  for all b.v_l ∈ b.A_k do
    if b.v_l = b.v_i then
      b.t ← b.t + 1 //triangle found
    end if
  end for
  b.phase ← phase5
```

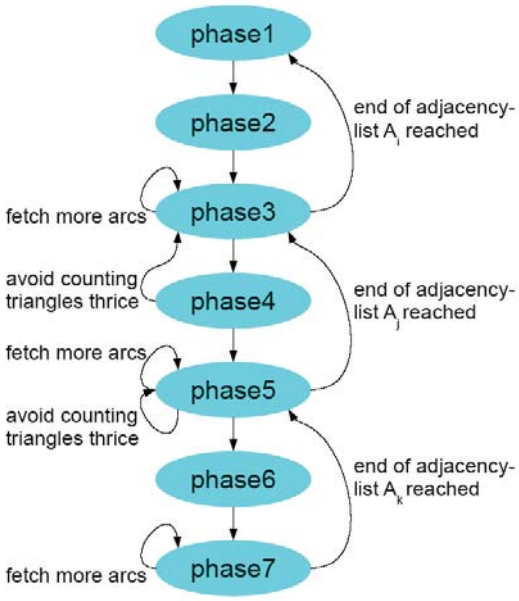


Figure 6.11.: The phases of the SPE implementation and how they are connected to each other. The progression from phase x to phase $x+1$ is always due to phase x issuing a DMA request to copy data from system memory into local memory, which is needed for phase $x+1$ to execute. Phases with an odd number end after they issue a request to fetch the start index and length information about a particular adjacency-list, whereas phases with an even number end after they issue a request to fetch the actual adjacency-list data for a particular vertex. The figure illustrates under which conditions a phase is repeated or the execution path goes back up towards *phase1*. See Algorithm 41 for the pseudo-code of the phases implementation.

Algorithm 42 Vector operations are used to speed-up the execution of the innermost loop (phase7) of the Cell BE PPE and SPE implementations. The comparison of vertex ID v_i with $v_l, v_{l+1}, v_{l+2}, v_{l+3}$ is done concurrently using the 128-bit vector unit. As the vector unit executes instructions in SIMD fashion, it is necessary to eliminate the branch. Several intrinsic instructions can be used to get the same effect as the if-condition: *spu_cmpeq* compares two vectors for equality and returns a bit-mask which represents true and false results; *spu_sel* selects one of two values (0 if the vertex IDs are not equal and 1 if a triangle has been found) based on this bit-mask; and *spu_add* adds the selected values to a vector that is used to count the number of triangles.

```
vec_uint4 case0 = spu_splats((uint32)0);
vec_uint4 case1 = spu_splats((uint32)1);
for (int nbr3Idx=0; <loop condition>; nbr3Idx+=4) {
  buf.nTrianglesVec =
    spu_add(buf.nTrianglesVec,
      spu_sel(case0, case1,
        spu_cmpeq(*((vec_int4*)&buf.arcsBuf3[nbr3Idx]), buf.vertexId)
      )
    );
}
```

It turns out that the performance gain from using both the PPE and the SPEs to process the data is smaller than expected compared to using either only the PPE or only the SPEs to do the actual data crunching. It appears that the memory system is the bottleneck when using all of the available processing units on the Cell processor on a data-intensive problem like the one at hand.

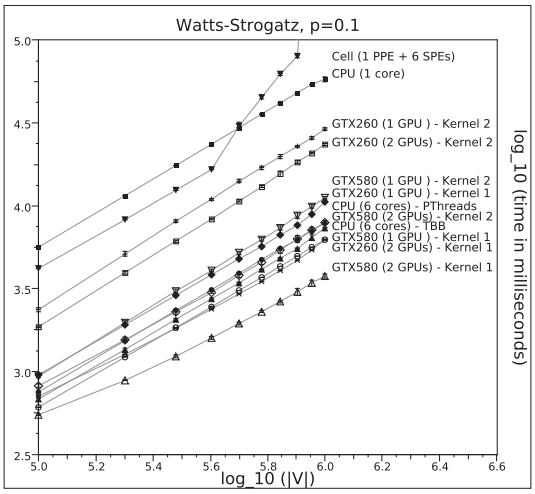


Figure 6.12.: The timing results for Watts-Strogatz small-world graphs with rewiring probability $p = 0.1$ and degree $k = 50$. The number of vertices $|V|$ ranges from 10^5 to 10^6. All data points are the mean values of 20 measurements. Error bars showing the standard deviations are smaller than the symbol size.

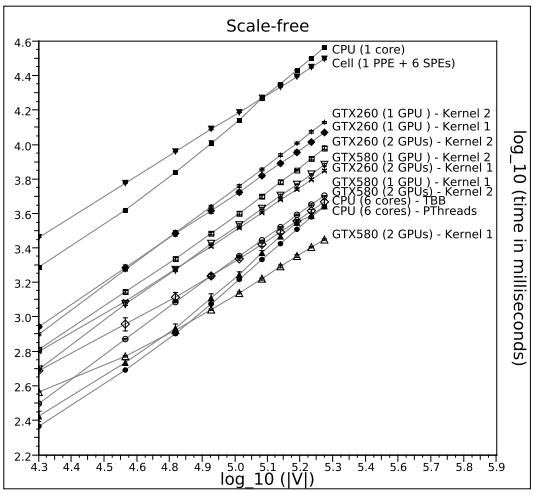


Figure 6.13.: The timing results for Barabási-Albert scale-free graphs with degree $k \approx 50$. The number of vertices $|V|$ ranges from $20,000$ to $200,000$. All data points are the mean values of 20 measurements. Error bars showing the standard deviations are smaller than the symbol size.

6.2.4. Performance Results

This section compares the performance of the different clustering coefficient implementations. The measurements reported here include the time required to allocate memory on the compute accelerator and copy the graph data between host and device memory. Contrary to the component labelling algorithm, the clustering coefficient is not likely to be computed repeatedly as an integral part of a simulation. It is more likely to be computed once for a static graph structure, although it may be part of a collection of metrics computed on the same graph and the copy times can therefore be spread out over multiple analysis algorithms. Either way, the copy times only make up about 1% of the execution time. However, for Kernel 2 the measurements also include the time required to sort the edges by the degree of the end vertex using `tbb::parallel_sort` on the CPU. This time is included in the measurements, as it is specific to the implementation of Kernel 2. For the largest of the tested scale-free graphs with $|V| = 200,000$ vertices, this preparation time is responsible for about 25% of the execution time on the Intel Core 2 Quad Q8200 CPU that is driving the GTX260 and for about 12% on the Intel Core i7 970 CPU that is driving the GTX580.

Two different graph structures are used to test the performance of the algorithms. The Watts-Strogatz β-model (see Section 2.6) is used to generate small-world graphs with mean degree $k = 50$ and rewiring probability $p = 0.1$. The performance results are given in Figure 6.12. As expected for small-world networks, the clustering coefficient $C \approx 0.53$ is high. The vertex degrees do not deviate much from k.

The Barabási-Albert scale-free network model (see Section 2.4) is used to generate graphs with a power-law degree distribution. The vertex degrees in the resulting graphs vary considerably. The graphs generated for the performance measurements have mean degree $k \approx 50$. The performance results are given in Figure 6.13. The clustering coefficient $C \sim 0.01$ is small for this graph structure.

The timing results show that the type of graph used as input to the algorithms has a considerable effect on the execution times. The scale-free graphs take significantly longer to process than the small-world graphs, because even though only few vertices have a degree that is much higher than the average, most vertices are connected to one of these hub nodes and the algorithms therefore often have to iterate over the long

Table 6.5.: Performance summary for the different implementations of the clustering algorithm. The speed-up values are relative to the sequential CPU implementation for the largest measured graph instance. An exception are the results for small-world graphs on the Cell processor, which are compared using size $|V| = 400,000$ as explained in the main text. The slopes of the least square linear fits to the data sets on a log-log scale measure how well the algorithms scale with the graph size. The first two data points are not taken into account when the fits are computed, as these graph instances are too small to show the proper scaling when using dual GPUs. The results for the Cell processor are the exception again, as only results for graph instances with $|V| \leq 400,000$ are used to compute the slopes.

Compute Device	**Small-world**		**Scale-free**	
	Speed-up	**Slope**	**Speed-up**	**Slope**
Core i7 970 (1 core) - Sequential	1.0	1.02	1.0	1.37
Core i7 970 (6 cores) - PThreads	7.4	1.02	8.3	1.40
Core i7 970 (6 cores) - TBB	7.9	1.08	8.3	1.33
CellBE (1 PPE & 6 SPEs)	1.4	1.00	1.2	1.02
GTX260 (1 GPU) - Kernel 1	5.5	1.08	3.1	1.12
GTX260 (2 GPUs) - Kernel 1	9.3	1.00	5.2	1.09
GTX260 (1 GPU) - Kernel 2	2.0	1.05	2.7	1.22
GTX260 (2 GPUs) - Kernel 2	2.5	1.12	3.8	1.22
GTX580 (1 GPU) - Kernel 1	9.3	1.02	7.2	1.18
GTX580 (2 GPUs) - Kernel 1	15.5	0.91	12.9	1.01
GTX580 (1 GPU) - Kernel 2	5.2	1.10	4.8	1.16
GTX580 (2 GPUs) - Kernel 2	7.3	1.06	7.8	1.08

adjacency-lists of these few vertices.

Table 6.5 gives an overview of the performance measurements and compares the results with each other. It shows that both the PThreads implementation – which uses one thread for each of the 12 logical cores of the Core i7 970 – as well as the TBB implementation with its automatic task management, are about 7 to 8 times faster than the sequential implementation for the largest measured instances of the small-world and scale-free graphs. Even though the processor only has 6 physical cores, Intel's hyper-threading technology enables it to maximise the throughput and achieve such speed-ups. The simpler TBB multi-tasking library has no significant overhead over the low-level PThreads implementation. Its ease of development and automatic scaling to different system configurations thus makes it a powerful alternative.

The Cell implementation positions itself between the single- and multi-threaded CPU implementations when processing the scale-free graphs or the smaller instances of the small-world graphs. The timing results for the small-world graphs suddenly increase at the $|V| = 400,000$ mark and even more considerably at the $|V| = 800,000$ mark. This is caused by the minimalistic 256 MB of main memory available in the PlayStation 3, which forces the system to start paging memory to the hard drive. The results are therefore filtered out above $|V| = 400,000$ when calculating the slope for these graphs to report the true scaling of the CellBE. The performance comparison to the sequential CPU implementation is done using graph instances of size $|V| = 400,000$.

The CUDA threads in Kernel 1 access the array of arcs A in the given order, whereas Kernel 2 uses a second array of arcs which is sorted by the degrees of the arc end vertices to determine which arc is processed by each thread. This second kernel uses $|A| \times$ sizeof(int) more space and introduces some processing overhead. It therefore comes as no surprise that Kernel 1 is faster when processing small-world graphs. That it is also faster when processing scale-free graphs is due to the fact that the vertex degrees of the scale-free graphs tested here tend to decrease with increasing vertex ID, which means that the node with the highest degree in a given cycle is more likely to be processed in the outermost loop than in the innermost loop. This is a good situation for Kernel 1, while Kernel 2 does not fully benefit from the additional work that it has to

perform, as vertices with sequential IDs already tend to have similar degrees. However, if the graph structure was less predictable, with the hub nodes spread out more randomly throughout the graph, then Kernel 2 would likely be faster. This claim has been verified with a simple test: the if-condition that compares the vertex ID of the source node to that of its neighbours (see lines 12 and 18 in Algorithm 30) is changed so that the algorithm continues when the ID of the source vertex is larger than that of its neighbours, instead of smaller. This has the effect that the innermost loop has to iterate over the long adjacency-lists of hub nodes more frequently than before. While this increases the execution times of both kernels, Kernel 1 on the GTX580 now takes about 13 times longer to finish processing the scale-free graphs with $|V| = 200,000$, whereas Kernel 2 only takes about 2 times longer. For comparison, the sequential CPU implementation takes about 6 times longer to finish the same task with this change to the algorithm.

The multi-GPU implementations running on two GTX260 or GTX580 need a large enough graph to reach their full potential. The results illustrate that Kernel 1 running on dual GTX580s does not begin to scale consistently until the graph size reaches about 10^5 vertices with 2.5×10^6 edges for the scale-free graphs and about $10^{5.5}$ vertices with close to 8×10^6 edges for the small-world graphs. Smaller systems do not have enough vertices to evenly distribute the work over two GPUs. The chunk size chosen for these experiments is 51200 for the GTX260 and a more moderate 30720 for the GTX580, due to the problems with this particular GPU discussed before. Smaller chunks increase the overhead of handing control back to the host and limit the number of threads available to the CUDA runtime. But the chunk size must not be too large either to keep both GPUs busy for approximately the same amount of time. Assigning a single chunk of half the system size to each GPU would only be a good solution if processing each vertex took exactly the same amount of time. However, this is not the case for this algorithm and it is therefore better to assign relatively small chunks to each GPU as needed.

6.3. Discussion

Both graph analysis algorithms discussed in this chapter make it clear that the GPU performance tends to depend heavily on the graph structure and that there is no one best implementation for all situations. This is particularly the case for algorithms that have to frequently traverse the links between neighbouring vertices. If the graph structure is not known beforehand, then it can be a good idea to spend a little bit of time on the attempt to automatically determine the type of the input graph. It may be enough to compare the degrees and possibly some other properties of a small fraction of the vertices to get a rough idea of what the graph structure is like. This can easily pay off in the total execution time, assuming that a CUDA implementation that is optimised for the detected graph type is available or in some cases even by the decision that this type of graph is best processed by the CPU.

Adjacency-lists of varying lengths are a particular challenge on data parallel architectures. A number of approaches to deal with this have been proposed in this chapter. For graphs with strongly varying vertex degrees, an edge-based approach like the one used in CUDA Kernel 1 of the component labelling algorithm can be very effective. This approach shifts the focus from individual vertices and their neighbours to the edges or arcs that connect a pair of vertices. The downside is that it tends to come at the cost of having to load some information about a particular vertex multiple times. The L2 and L1 caches on Fermi-architecture based devices can mitigate this cost as long as all edges that are adjacent to a particular vertex are processed while the required data remains in the caches.

The second approach to dealing with varying adjacency-list lengths, used by CUDA Kernel 2 of the clustering coefficient, is to sort the vertices such that sequential threads process vertices with similar degrees. While this can be very effective, it often introduces some overhead in the kernel itself, like having to look up the vertex ID from global memory instead of being able to infer it from the thread ID, in addition to the time required to sort the vertices. While this overhead has proven to be too high in the performance

results presented here, the discussion in the performance section of the clustering coefficient has described a scenario where it can significantly improve the performance compared to other approaches.

Considering the age of the Cell BE at the time of writing, the performance results for the clustering coefficient metric are quite impressive and show the potential of this hybrid CPU architecture. However, the implementation is considerably more complicated than the algorithms for multi-core x86 CPUs. Especially the memory management requires a lot of low-level planning and optimisation, even more than what is necessary for the GPU. Where CUDA devices automatically try to switch between threads that are waiting for some memory request to be fulfilled and threads that are ready to run, the programmer has to do this explicitly for the Cell processor.

The CUDA SIMT programming model is also more powerful than the SIMD instructions used by the Cell BE, as it makes it easier for the programmer to think in terms of individual threads and let the device handle things like warp divergence. While it is of course necessary to minimise warp divergence to achieve high performance, the programmer does not have to worry about doing anything special for situations where some small divergence can not be avoided. For example, if the total number of threads used to process a particular system is slightly larger than the system size, then a simple if-condition can be used to ensure that no thread accesses the data array outside its bounds. Only a single warp would experience diverging threads in this scenario, and the impact on the performance would therefore be minimal.

Part III.

Complex Systems Simulation & Analysis

Chapter 7

The Rewired Ising Model

The Ising model [32,133] is a model of a computational ferromagnet. It is used to calculate the critical point of metal alloy phase transitions. While analytical methods have been used to successfully analyse the Ising model in one [32] and two [134] dimensions, no such methods are known for three or more dimensions. In fact, Istrail [135,136] has shown that the complexity of solving the Ising model on any non-planar lattice is NP-complete. Monte Carlo simulations [34], which use random sampling to approximate results when it is infeasible or impossible to compute the exact result for a physical or mathematical system, are often used instead.

The immense interest in the Ising model over the last decades is not so much due to its physical realism, but rather, as Newell and Montroll [137] put it:

> [The] widespread interest in the model is primarily derived from the fact that it is one of the simplest examples of a system of interacting particles which still has some features of physical reality in it. The model forms an excellent test case for any new approximate method of investigating systems of interacting particles. If a proposed method cannot deal with the Ising model, it can hardly be expected to be powerful enough to give reliable results in more complicated cases.

Simulations of the Ising model typically start with a random "hot" system. The system is then quenched to a specific temperature. If this temperature is below a critical "cold" temperature, then a system of two or more dimensions undergoes a phase transition where like spin values begin to clump together, creating order in the initially random system. This transition from paramagnetic (non-ferromagnetic) to ferromagnetic regime occurs at the Curie temperature T_c. No phase transition occurs in the one-dimensional (1D) case and the model does not display ferromagnetic behaviour [32].

The Ising model has just two possible spin values, "up" and "down", but can be extended to the Q-state Potts model [138] that uses Q spin values. A system quenched to a temperature very close to the critical temperature shows clusters of like-like spins on all possible length scales. Figure 7.1 illustrates a two-dimensional (2D) Ising model simulation.

The spins in the ferromagnetic Ising model interact with their nearest neighbours according to an energy function or Hamiltonian of the form [139]:

$$H = -\sum_{\langle i,j \rangle} J_{ij}\sigma_i\sigma_j, \tag{7.1}$$

where $\sigma_i = \pm 1$, $i = 1,2,...N$ sites. J_{ij} is $|J| = 1/k_BT$ is the ferromagnetic coupling over neighbouring sites i and j on the network, T is the temperature and k_B is the Boltzmann constant. The total energy E of

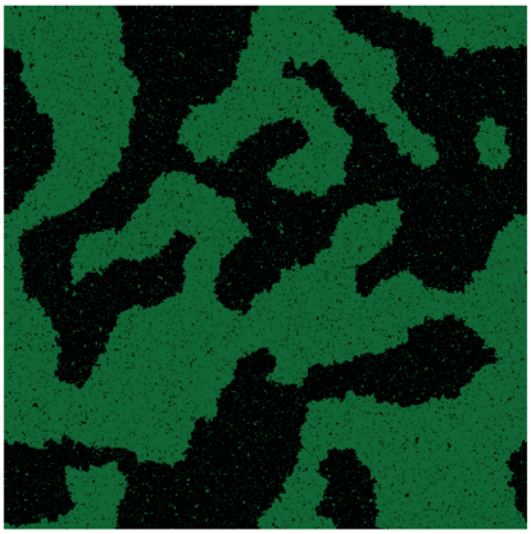


Figure 7.1.: A 1024×1024 Ising model simulation with temperature $T = 2.0$ after 1000 simulation steps.

a single configuration is obtained from the Hamiltonian. The magnetisation M is measured from a single configuration as [139]:

$$M = \frac{1}{N}\left|\sum_i \sigma_i\right| \tag{7.2}$$

The Ising model can be used to study a large number of natural phenomena in systems where pairwise correlations between neighbouring nodes on the microscopic scale give rise to macroscopic behaviour. For example, in the social sciences the propagation of opinions has been studied using the Ising model [140]. Such systems are, however, often more accurately described by irregular graphs instead of regular lattices, prompting Ising model studies on small-world [140–142] and scale-free networks [143].

Barrat and Weigt [141] determined the small-world effect of the Watts-Strogatz β-model on the 1D Ising model. They showed that, unlike the regular 1D case studied by Ising, it undergoes a ferromagnetic phase transition at a critical temperature $T_c(p > 0) > 0$ for networks that lie in the region of disorder that classifies a small-world graph (see Section 2.5) and that $T_c(p)$ depends asymptotically on p for $p \ll 1$ such that $T_c(p) \propto -\frac{k}{ln(p)}$.[1]

Herrero [142] has also studied the Ising model on small-world networks that are based on the Watts-Strogatz β-model, generated from two- and three-dimensional regular lattices by rewiring edges with probability p and excluding sites that are not connected to the giant component from the final network. The largest network sizes tested were $N = 200^2$ for the 2D and $N = 40^3$ for the 3D Monte Carlo simulations using the Metropolis update algorithm [144]. Periodic boundary conditions were assumed and the critical temperature $T_c(p)$ marking the ferromagnetic transition for a given p was determined using Binder's fourth-order cumulant method [145]. Herrero shows that the shift in critical temperature $\Delta T_c = T_c(p) - T_c(p = 0)$ goes as a power-law in p following $\Delta T_c \sim p^s$ for $p \lesssim 0.01$. The exponent s derived from his Monte Carlo simulations is 0.96 ± 0.04 in the 3D case. However, the expected change in the critical temperature for small p is [142]:

$$\Delta T_c \sim p^{1/\nu d}, \tag{7.3}$$

[1] Note that Barrat and Weigt [141] use k to mean the vertex connectivity in either direction of the unperturbed ($p = 0$) ring, that is half the degree in the one-dimensional scenario, whereas here k is used to denote the actual degree.

where d is the number of dimensions and the critical exponent $\nu \approx 0.63$ for the regular 3D cubic lattice Ising model, as obtained in good agreement from both renormalization group studies [146–148] and numerical studies [149].

Critical exponents like ν capture the behaviour of interesting quantities, in terms of a power-law, near a continuous phase transition at temperature T_c. When seemingly dissimilar models exhibit the same critical exponents, then they are said to be in the same universality class. It turns out that there are much fewer universality classes than macroscopic phenomena. This is why the critical exponents of the relatively simple Ising model have received a lot of attention in the literature (see for example [55, 59, 149–153]).

When $\nu \approx 0.63$ is inserted into Equation 7.3, the expected result for ΔT_c is $\sim p^{0.53}$, which is not in agreement with Herrero's result. Herrero states [142]: "It seems that this discrepancy appears because the smallest p value employed in our simulations ($p = 10^{-3}$) is still too large to observe the small-p behavior".

This is where the results presented in this chapter come into play. In order to remain in the small-world regime while studying smaller values of p, it is necessary to increase the system size according to the scaling law described in Algorithm 2.11 on page 31. CUDA algorithms that perform the Ising model simulation on the GPU are introduced. These algorithms utilise the highly parallel architecture to process systems of up to size $N = 512^3$. These large systems are then used to determine the critical temperature for values of p down to 10^{-7}. The hypothesis is that ΔT_c, the perturbation in the critical temperature, goes as a power-law in p following Equation 7.3 for the 3D Ising model.

7.1. Metropolis Updates

A number of different Monte-Carlo update algorithms for the Ising model have been proposed over time [144, 154–157]. The Metropolis algorithm [144], which was later generalised by Hastings [158], has formed the basis for Monte-Carlo statistical mechanics [159, 160] and has been used widely for Ising model simulations [151, 152, 161, 162]. It is a Marcov chain Monte-Carlo (MCMC) method, where the transitions from one state to the next only depend on the current state and not on the past. Using the Metropolis update algorithm for the Ising model simulation, at each discrete time step, a new system configuration is chosen at random by picking a spin to "hit" and flipping its value. If the energy E of the proposed configuration is lower than or equal to the current energy, $\Delta E \leq 0$, then the move to the new configuration is always accepted. Otherwise, the new configuration is accepted with probability $exp(-\Delta E/k_B T)$. The current configuration is retained if the move is rejected.

Here[2] a simulation step is used to refer to a Metropolis update of the entire system. Every spin is considered exactly once per simulation step. It is possible to process the individual spin updates in parallel, as long as no spin is considered simultaneously with its neighbouring cells. Concurrent updates of neighbouring spins would cause race-conditions, as the change in energy from a proposed spin flip depends on the current spin values of the neighbours. To avoid this situation, the updates are performed using a checkerboard pattern. Every cell can be thought of as being assigned one of two colours, for example red and black, and no neighbouring cells are allowed to have the same colour. In the regular lattice, two cells are considered neighbours only if their coordinates differ in exactly one dimension by ± 1 (periodic boundary conditions apply).

7.1.1. Generating the Small-World Lattice

The small-world system is constructed from a regular d-dimensional lattice. A number of edges are then rewired randomly to create the shortcuts common to small-world networks. Either end of an edge, which

[2] This section extends on work first published in [112] K. A. Hawick, **A. Leist**, and D. P. Playne, "Regular Lattice and Small-World Spin Model Simulations using CUDA and GPUs," *International Journal of Parallel Programming*, vol. 39, no. 2, pp. 183–201, April 2011.

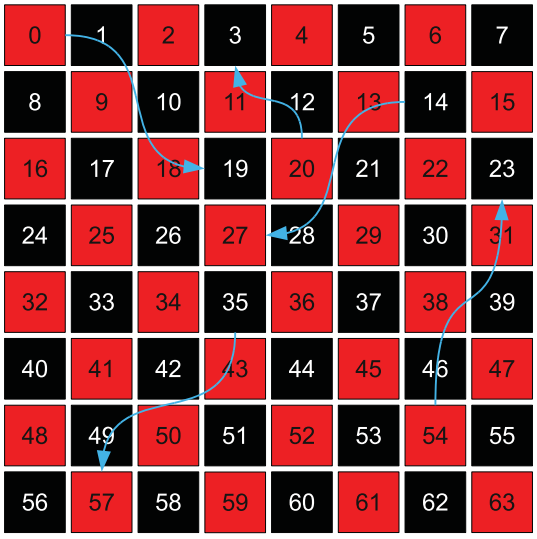


Figure 7.2.: A 2-dimensional lattice where every cell is initially connected to its 4 direct neighbours using periodic boundaries, with a small number of rewired edges. The numbers represent the cell IDs.

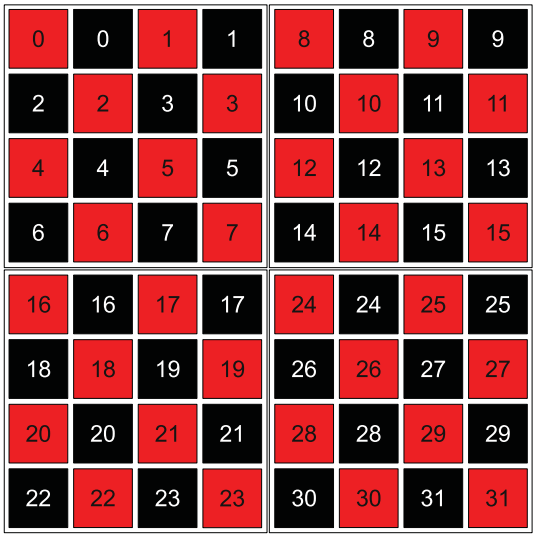


Figure 7.3.: The ID of the thread that processes a particular cell (no rewiring) for thread blocks of size 2×4. The actual implementation uses thread blocks of size 8×16 for 2D and $4 \times 8 \times 8$ for 3D simulations.

connects two cells, is independently considered for rewiring with probability $\frac{1}{2}p$, thus rewiring a fraction p of all edges. This process is similar to the Watts-Strogatz β-model graph algorithm, described in Section 2.6, extended to d-dimensions.

A new neighbour is selected randomly from all cells of the same colour as the previous neighbour, thus preserving the condition that no cells of the same colour are connected to each other. This limitation makes it possible to update cells of the same colour in parallel without the risk of running into race conditions. Figure 7.2 illustrates such a rewired lattice. In this example of a 2D system, the edge connecting cells 0 and 1 was rewired and now connects cells 0 and 19, decreasing the degree of cell 1 to 3 and increasing the degree of cell 19 to 5.

The CUDA implementation uses thread blocks of size 8×16 in 2D and $4 \times 8 \times 8$ in 3D when processing either the red or black cells of a 16×16 and $8 \times 8 \times 8$ block of cells respectively. Figure 7.3 illustrates which thread processes a particular cell when no edges are rewired. As some edges may be rewired, the actual cell processed by a particular thread is looked up from memory and may be part of the next cell block. Cells that are not affected by rewiring and those that are affected are processed by different kernels as explained in the following sections. In the example given in Figure 7.2, the red cells 0 and 27 in the first block are affected by rewiring. Thus, when processing the red cells not affected by rewiring, threads $0, 1, 2, \ldots$ would process cells $2, 9, 11, \ldots$ respectively, and threads 6 and 7 would process cells 4 and 6 from the next cell block.

7.1.2. Rewired Irregular Data Structure

The neighbour coordinates of cells that are not affected by rewiring can be calculated and do not need to be stored explicitly, thus conserving memory and memory bandwidth. However, for all other cells, the neighbours can not be deferred and need to be stored and looked up. Two vertex arrays, one for the red cells and one for the black cells, are used to store the index into the arc-array at which the neighbour information for the cells affected by rewiring is stored. As only cells of the same colour are processed in one iteration,

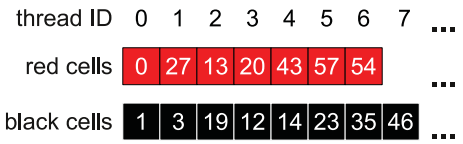


Figure 7.4.: The two vertex-arrays for red and black cells respectively. The figures illustrate the IDs of the cells whose data is stored at the respective positions. The actual data stored are the indices into the arc-array (Figure 7.5) at which the adjacency-list information begins.

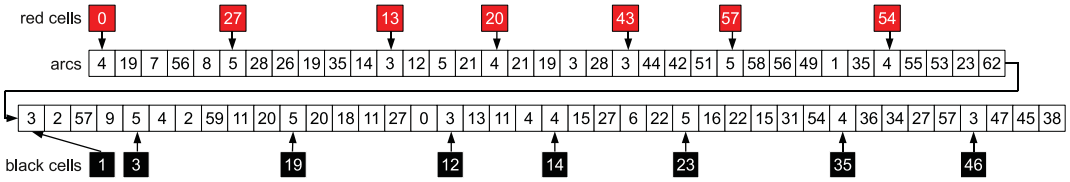


Figure 7.5.: The arc-array only stores the adjacency-lists of cells that are affected by rewiring. The neighbours of all other cells can be deferred from their own coordinates, thus reducing the memory storage and bandwidth requirements. The arrows between the vertex and arc-arrays represent the index into the arc-array stored in the vertex-arrays. The first element of an adjacency-list is its length.

two different vertex-arrays are used to enable threads to access sequential elements of the arrays. However, this is not enough to achieve coalesced memory access, as the threads of a half-warp must access sequential memory addresses which adhere to strict alignment requirements for coalescing to work. Because threads may process cells from different rows or slices of the 2D or 3D cell blocks, the information needs to be stored in the order of the thread IDs instead of the cell IDs. Figure 7.4 illustrates this for the graph structure given in Figure 7.2.

The arc-array, illustrated in Figure 7.5, contains all explicitly stored adjacency-lists. The first element of every such list is its length, followed by the IDs of the cells that are adjacent to the respective source cell.

The spin states are stored as 2D or 3D arrays in global memory using `unsigned char (Uchar)` values. Only the least significant bit (LSB) is used to record the spin state in the Ising model. Additional bits can be used to store more spin states, which makes it easy to extend the algorithm to the Potts model.

An additional global memory array that stores the mapping from the thread IDs to the cell IDs is needed for each of the red/black and modified/unmodified cell combinations. These four arrays have a total length of N, where N is the system size.

7.1.3. The CUDA GPU Implementation

The small-world Ising simulation running on the GPU updates either the red or the black cells of the checkerboard pattern in parallel. This allows it to avoid race conditions, as all the neighbours of a red cell are black and vice-versa. It also uses different kernels to process unmodified cells and cells affected by rewiring. This means that it takes multiple kernel calls to perform a full simulation step. Algorithm 43 describes the host code that prepares and manages the CUDA kernel execution. The spin-, vertex-, arc- and mapping-arrays are described above. The random number array is used to temporarily store the random numbers required by the Ising kernel for modified cells and is filled by a dedicated random number kernel. The random number function implements Marsaglia's lagged-Fibonacci random number generator as described in Chapter 4.

Algorithm 43 Evolve the model by STEPS simulation steps. This is the host code that manages the graphics processing unit. N is the system size. The number of cells of a particular type (**r**ed or **b**lack, **m**odified or **u**nmodified) is defined as the length of the respective vertex set $|V_{\{ru,rm,bu,bm\}}|$.

allocate device memory for the spin-array S, vertex-arrays $V_{\{r,b\}}$, arc-array A, mapping arrays $M_{\{ru,rm,bu,bm\}}$, random number array R and the arrays needed for the random number generators
copy metropolis table T_M to constant device memory
copy initial spin values to the device
copy seed values for the random number generator instances to the device
do in parallel on the device using T threads: initialise the RNGs
for $i \leftarrow 1$ to STEPS **do**
  //process the red cells
  do in parallel on the device using T threads: process all unmodified red cells
  do in parallel on the device using T threads: generate $|V_{rm}|$ random numbers and store them in R
  do in parallel on the device using $|V_{rm}|$ threads: process all modified red cells
  //process the black cells
  do in parallel on the device using T threads: process all unmodified black cells
  do in parallel on the device using T threads: generate $|V_{bm}|$ random numbers and store them in R
  do in parallel on the device using $|V_{bm}|$ threads: process all modified black cells
end for
copy final spin values back to the host

Algorithm 44 This is the CUDA kernel for unmodified cells in 3D. It is executed by T threads, each of which processes a number of cells until all $|V|$ unmodified cells have been processed. S is the spin-array, M the thread ID to cell ID mapping array, L the dimension length and T_M the Metropolis table.

$t \leftarrow$ thread ID queried from the CUDA runtime
load variables for the RNG instance used by this thread from global memory to registers
for all $tid \in V_t := \{t, t+T, t+2*T, \ldots; t < |V|\}$ **do**
  $id \leftarrow M[tid]$ //load the cell ID
  $ix \leftarrow id$ AND $(L-1)$ //calculate the x-coordinate
  $iy \leftarrow (id >> log2(L))$ AND $(L-1)$ //calculate the y-coordinate
  $iz \leftarrow id >> log2(L*L)$ //calculate the z-coordinate
  $s \leftarrow S(ix, iy, iz)$ //load the spin value
  $b \leftarrow 0$ //change in like-like bonds if spin gets flipped
  //calculate the neighbour coordinates
  $n\_ix \leftarrow ix = 0\ ?\ L-1 : ix-1$
  $n\_s \leftarrow S(n\_ix, iy, iz)$ //load the spin value for the cell on the left
  $b \leftarrow b + (s = n\_s\ ?\ -1 : 1)$ //compare the spins and update the like-like bond counter
  repeat the previous 3 lines for all other neighbours (right, above, below, front, behind)
  $r \leftarrow$ generate uniform random number
  if $b \geq 0$ or $r < T_M[-b]$ **then**
    $S(ix, iy, iz) \leftarrow (s$ XOR 1) //write the flipped spin value to global memory
  end if
end for
write the updated local variables for the RNG instance back to global memory

The Ising kernels compare the spins of all cells to the spins of their neighbours and flip each spin if this either increases its like-like bonds, thus decreasing the system energy, or with a random probability that follows $exp(-\Delta E/k_B T)$. If the dimension lengths L are a power of 2, then certain bitwise operations, shown in the pseudo-code, can be performed to improve the performance. The values for the base 2 logarithms of the dimension length and squared dimension length used by the CUDA kernels are calculated once by the host and passed as parameters to the kernels.

Algorithm 44 describes the kernel for unmodified cells. This kernel is executed by T threads, which call the RNG function directly. The number of threads T depends on the hardware as described in Section 4.3. The random numbers only exist in the registers and do not need to be copied to global memory. The kernel can calculate the neighbour coordinates instead of having to look them up explicitly from global memory, as the neighbours of a cell can be deduced from its own coordinates. Every thread processes $|V_t|$ cells per kernel call, where $|V_t|$ is the number of **u**nmodified **r**ed or **b**lack cells $|V_{\{ru,bu\}}|$ divided by T. If this does

Algorithm 45 This is the CUDA kernel for modified cells in 3D. Every CUDA thread processes one cell, thus $|V_{\{rm,bm\}}|$ threads are executed to process all modified red/black cells. S is the spin-array, V the vertex-array, A the arc-array, M the thread ID to cell ID mapping array, R the random number array, L the dimension length and T_M the Metropolis table.

$tid \leftarrow$ thread ID queried from the CUDA runtime
$id \leftarrow M[tid]$ //load the cell ID
$ix \leftarrow id$ AND $(L-1)$ //calculate the x-coordinate
$iy \leftarrow (id >> log2(L))$ AND $(L-1)$ //calculate the y-coordinate
$iz \leftarrow id >> log2(L*L)$ //calculate the z-coordinate
$s \leftarrow S(ix,iy,iz)$ //load the spin value
$b \leftarrow 0$ //change in like-like bonds if spin gets flipped
$idx \leftarrow V[tid]$ //load the index into the arc-array
$c \leftarrow A[idx]$ //load the adjacency-list length
for all $n\_id \in \{A[idx+1], A[idx+2], \ldots, A[idx+c]\}$ **do**
  $n\_ix \leftarrow n\_id$ AND $(L-1)$ //calculate the neighbour's x-coordinate
  $n\_iy \leftarrow (n\_id >> log2(L))$ AND $(L-1)$ //calculate the neighbour's y-coordinate
  $n\_iz \leftarrow n\_id >> log2(L*L))$ //calculate the neighbour's z-coordinate
  $n\_s \leftarrow S(n\_ix, n\_iy, n\_iz)$ //load the neighbour's spin
  $b \leftarrow b + (s = n\_s\ ?\ -1 : 1)$ //compare the spins and update the like-like bond counter
end for
$r \leftarrow R[id]$ //load the random number generated for this cell
if $b \geq 0$ or $r < T_M[-b]$ **then**
  $S(ix,iy,iz) \leftarrow (s$ XOR $1)$ //write the flipped spin value to global memory
end if

not work out evenly, then some threads have to process an additional cell. This implementation performs better than creating $|V_{\{ru,bu\}}|$ threads and processing exactly one cell with each thread, as it does not require a separate kernel to generate the random numbers and store them to global memory and it also reduces the overhead of creating CUDA threads. Only the `if`-condition with its single-line body or the last iteration of the loop can cause threads to diverge.

The Ising kernel for cells that are affected by rewiring is described in Algorithm 45. It uses $|V_{\{rm,bm\}}|$ CUDA threads, one for each **m**odified **r**ed or **b**lack cell. This kernel has to iterate over the explicitly stored adjacency-lists of the cells, which can slightly differ in length and therefore are likely to cause threads to diverge. While thread divergence always hurts performance, it does so to a smaller degree if there is no outer-loop like in Algorithm 44. However, this means that if the RNG function was called directly by this kernel, then it would require $|V_{\{rm,bm\}}| \times 400$ bytes of global memory just for the RNGs, which can be a lot if the system is large or the rewiring probability is high. Therefore, a separate kernel, which is executed by T CUDA threads, is used. Every thread generates x random numbers per kernel call, where x is the next multiple of T that is equal to or greater than $|V_{\{rm,bm\}}|$.

The spin-array is bound to a texture reference to read the spin values for the 2D kernels. Texture fetches are optimised for spatial locality, which is exactly what is needed, as every cell requires the spins of its neighbours and most of its neighbours are stored spatially close to it unless the value of p is rather large. Typically, texture references are bound to CUDA arrays, which are "opaque memory layouts optimized for texture fetching" [90]. However, they are also read-only[3] and here the spin value needs to be updated. This problem could be solved by writing to a second array, which is stored in global memory and copied to the CUDA array before every iteration. This copy, however, turns out to be too expensive even though it is a fast device-to-device copy. Simply reading from global memory is faster than using texture fetches that require this copy operation. For the 2D simulation exists another solution though. CUDA allows 1D and 2D texture references to be bound directly to linear memory, which means that the texture can be bound to

[3] While CUDA arrays have traditionally been read-only, CUDA 3.1 introduced surface references, which make it possible to update one and two-dimensional CUDA arrays. They are only supported on devices with compute capability ≥ 2.0 (i.e. Fermi-architecture based GPUs) and do not offer any advantages here, as they can currently not be bound to 3D CUDA arrays. In the 2D case linear memory can be used for the same effect.

Algorithm 46 The parallel CPU implementation of the Metropolis algorithm. The model is evolved by `STEPS` simulation steps. T tasks are used, where T is the dimension length of the highest dimension.

```
prepare the metropolis lookup table T_M
create T random number generator instances
do in parallel using T threads: randomly initialise the spin array S
for i ← 1 to STEPS do
    do in parallel using T threads: call update_checkerboard(0, 1) to update all red cells
    do in parallel using T threads: call update_checkerboard(1, 0) to update all black cells
end for
```

Algorithm 47 Function `update_checkerboard` for 3D systems. It is executed in parallel using T tasks, where T is equal to dimension length L. S is the spin array and A_i is the adjacency-list of vertex i. The structure `myRng` is the task's private RNG instance. The offsets specify if the red cells or the black cells are to be processed.

```
function update_checkerboard(offset1, offset2)
  iz ← taskID
  for iy ← 1 to L do
    //initialise ix to offset1 if iz and iy are both even or both odd and to offset2 if one is even and the other one odd
    ix ← ((iz AND 1) XOR (iy AND 1) ? offset1 : offset2 )
    while ix < L do
      id ← iz × L^2 + iy × L + ix //the cell ID
      s ← S[id] //load the spin value
      b ← 0 //change in like-like bonds if spin gets flipped
      for all n_id ∈ A_id do
        b ← b + (s = S[n_id] ? −1 : 1) //compare the spins and update the like-like bond counter
      end for
      if b ≥ 0 or myRng.uniform() < T_M[−b] then
        S[id] ← (s XOR 1) //flip the spin value
      end if
      ix ← ix + 2
    end while
  end for
```

the same array that is used to write to. No copy operation is needed in this case. The written values are only guaranteed to be visible to other threads after the kernel call returns, but this is not an issue, as the checkerboard pattern ensures that a spin is never read and modified by different threads in the same call. Unfortunately, 3D texture references can not be bound to linear memory (as of CUDA 3.2), which means that the 3D implementation has to read directly from global memory. While this comes at a performance deficit compared to the 2D implementation, it frees up the texture cache, which can be useful in a different way. Binding a 1D texture reference to the arc-array allows the neighbours of a cell to be read using texture fetches, which provides a moderate performance boost.

7.1.4. Sequential & Parallel CPU Implementations

Both a sequential and a multi-threaded CPU implementation are used to compare the performance of the CUDA GPU implementation. Algorithms 46 and 47 describe the parallel implementation, which uses TBB to process the checkerboard pattern also used by the CUDA algorithm. The sequential implementation is essentially the same as the multi-threaded implementation, but only uses one thread and RNG instance to update the entire system.

The parallel implementation uses T tasks to parallelise the outermost loop, which iterates over either the z-dimension (3D simulation) or over the y-dimension (2D simulation) of the lattice. T is equal to the respective dimension length. A task is therefore either assigned to process a slice or a row of the lattice. Each task has its own instance of Marsaglia's random number generator and gets invoked using `tbb::parallel_for_each` in the same fashion described for the TBB implementation of the RNG in Sec-

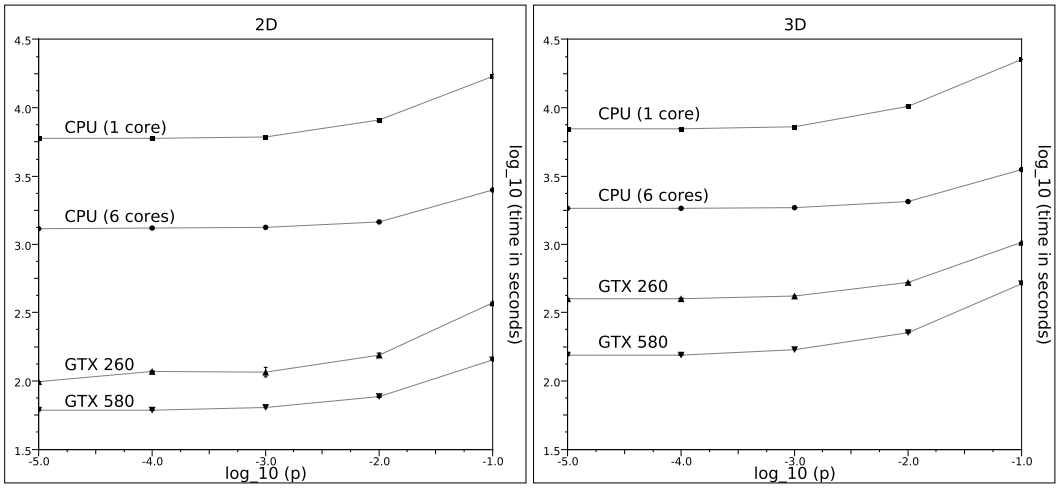


Figure 7.6.: The plots illustrate how the rewiring probability p affects the execution times of the Metropolis update algorithm in 2D (left) and 3D (right) simulations. The temperature and system size are set to $T = 2.269$ and $N = 4096^2$ for the 2D systems and to $T = 4.5115$ and $N = 256^3$ for the 3D systems. Error bars are mostly smaller than the the symbol size.

tion 4.2. Because the parallelism is applied to the outermost loop, each task has enough work to perform to keep the scheduling overhead relatively small. But at the same time enough tasks are created to provide the task scheduler with the flexibility it needs to optimally distribute the workload to the processing cores and thus maximise the performance, as shown for the parallel RNG implementation in Figure 4.1 on page 46.

A number of different implementations have been tested but do not achieve the same performance as the approach described here. The first of these alternative implementations uses T heavy-weight threads instead of TBB's light-weight tasks, where T is set to the number of logical cores available to the application. Each of these threads is then assigned an index range from the outermost loop that it has to process. The second alternative implementation uses `tbb::parallel_for`, which is called twice per simulation step to iterate over one part of the checkerboard pattern, that is $N/2$ sites, each time. This is similar to the way the CUDA kernel processes individual spins. While this algorithm gives the task scheduler a lot of flexibility, it makes it necessary to generate the random numbers for each simulation step separately from the update algorithm to assure the repeatability of the simulation. This implementation is slower than the other implementations, as it always generates N deviates, even though a random number is only needed when the energy of a site increases with the proposed spin flip. While this works on the GPU with its large number of processing cores and high memory bandwidth, it is not the best approach for the CPU.

7.1.5. Performance Results

This section shows how the execution times of the Metropolis update algorithm are affected by the system size N and rewiring probability p. The CUDA GPU measurements are compared to sequential and multi-threaded CPU implementations. The timings are for 16384 simulation steps. Every data point is the mean value of 10 simulation runs on the GPU and 5 simulation runs on the CPU using different random number seeds for every run.

Figure 7.6 shows how the different implementations cope with increasing rewiring probabilities both in 2D and 3D simulations. Not surprisingly, the execution times increase with the rewiring probability p and the corresponding perturbation of the underlying lattice. Every rewired edge affects three vertices – the source of the edge as well as the original and new end vertices – all of which have to be processed by the

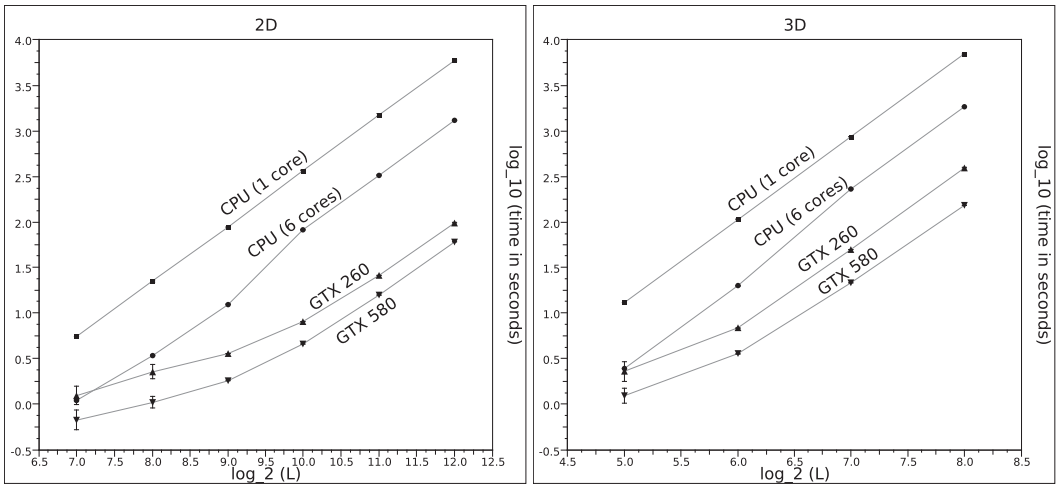


Figure 7.7.: The plots show how the performance of the Metropolis update algorithm scales with the system size $N = L^2$ for 2D (left) and $N = L^3$ for 3D (right) simulations. $L = \{128, 256, \ldots, 4096\}$ in 2D and $L = \{32, 64, 128, 256\}$ in 3D. The temperature is set to $T = 2.269$ (2D) and to $T = 4.5115$ (3D). The rewiring probability $p = 10^{-5}$. Error bars are mostly smaller than the symbol size. Note the different log-scales on the x- and y-axis.

kernel that uses explicit neighbour lookups in the CUDA implementation. The number of modified cells increases quickly with the value of p, as there are twice as many edges as vertices in the 2D model and thrice as many in the 3D model. While the CPU implementation always reads the neighbour information explicitly, rewired edges cause memory reads from random memory locations, which means a higher chance of a cache miss. Conveniently, very small values of p, like those of interest in this thesis, affect too few vertices to have a significant impact on the performance and only values larger than 10^{-3} increase the execution times more noticeably.

The results also show that the performance of the CUDA implementation is more strongly affected by the step from 2D to 3D simulation than the CPU implementations. This can be attributed to the fact that the 3D simulation does not make use of the texture cache for spin access as discussed in Section 7.1.3. The older GTX 260 is even more strongly affected by this than the Fermi-architecture based GTX 580, as the former does not have the L1/L2 cache structure to fall back to in order to improve the reads from the spin array when not using the texture cache. The speed-up achieved by the GTX 580 compared to the sequential CPU implementation running on an Intel Core i7 970 is 97.7 times for the 2D simulations and remains a respectable 45.4 times for the 3D simulations for $p = 10^{-5}$. The multi-threaded TBB implementation is 4.3 times (2D) and 3.7 times (3D) faster than the sequential code.

Figure 7.7 illustrates how the execution times scale with the system size. It shows that the GPUs are not fully utilised until the system size reaches several hundred thousand spins, where the timing results begin to scale linearly with the system size. The CUDA hardware can effectively hide memory latencies when it has enough threads and hardware resources – like registers and shared memory – available to switch between threads that are waiting for some resource and threads that are ready to process the next instruction. In the kernel for unmodified cells, a larger number of spins to process per thread also means that the constant time spent outside the main loop becomes less significant relative to the total execution time. A similar behaviour can be observed when all cores of the CPU are used, although here it is more likely caused by the number of instructions to be executed per TBB task. Tests on a different CPU have shown that the system size at which the transition occurs depends on the hardware.

The larger errors in the execution times of the GPU kernels observed for the smallest systems with less

Table 7.1.: Summary of the Metropolis update performance results. The speed-up values are for the largest measured system size with $p = 10^{-5}$ and relative to the respective sequential CPU implementation. The quoted slopes are for the least squares linear fits to the data sets on a $\log_{10}(x) - \log_{10}(y)$ scale, unlike the $\log_2(x) - \log_{10}(y)$ scale used for illustration purposes in Figure 7.7. Only the last three data points of each data set are used to calculate the slopes, as smaller system sizes do not fully utilise the graphics hardware.

Compute Device	**2D Simulation**		**3D Simulation**	
	Speed-up	**Slope**	**Speed-up**	**Slope**
Core i7 970 (1 core)	1.0	2.00	1.0	3.02
Core i7 970 (6 cores)	4.3	1.99	3.7	3.06
GTX 260	60.2	1.86	17.6	2.96
GTX 580	97.7	1.79	45.4	2.79

than about 200,000 spins is caused by the value of p used for the given measurements, which is too small to consistently cause any rewiring for systems of this size. The performance difference between the kernels for modified and unmodified cells is the reason for the irregularities. When using a large enough rewiring probability, the error values for these small systems are on the same scale as those for larger systems.

The slopes for the least squares linear fits to the last three data points of each set are given in Table 7.1. The execution times on the CPU are closely related to the system size, which is $N = L^2$ in 2D and $N = L^3$ in 3D. The GPUs, in particular the GTX 580, clearly benefit from larger system sizes, as the time per spin decreases slightly with increasing size. This can mainly be credited to the CUDA kernel for unmodified cells, which uses a constant number of threads to process an increasing number of cells, reusing some of the local values for the thread ID and RNG instances.

7.2. Wolff Cluster Updates

It is well known that the Metropolis algorithm slows down near the critical temperature, because the size of the correlated regions is large and the local update dynamics – where individual sites only interact with their direct neighbours – mean that it takes a long time for a region to lose its coherence. This effect is usually referred to as critical slowing down. Cluster update algorithms, like those introduced by Swendsen and Wang [155] or Wolff [156], do not suffer from this problem, as they update entire clusters of spins in one go. This section describes a GPU implementation of Wolff's cluster update algorithm, which works as follows [156]:

1. Pick a random lattice site, i, and mark it as the first member of a cluster to be built
2. Visit all links $\langle i, j\rangle$ connecting i to its nearest neighbours j
3. If spins σ_i and σ_j have the same value, then activate bond $\langle i, j\rangle$ with probability $p(\langle i, j\rangle) = 1 - e^{-2\beta}$, where β is the reciprocal temperature. Where this happens:
   - Mark site j as part of the cluster
   - Continue this process for all nearest neighbours of j that are not part of the cluster yet
4. When the process stops, flip the entire cluster.

Every possible cluster in the system is chosen with a bias proportional to its size. Each cluster flip is counted as one simulation step. Unlike the Metropolis algorithm, the cluster update algorithm does not "hit" all spins in the system during each simulation step. A typical cluster is small compared to the system size.

The procedure used to rewire the underlying lattice is the same as the one described for the Metropolis algorithm in Section 7.1.1. While the checkerboard pattern is not needed for the parallel implementation of the cluster update algorithm, the restriction that two cells can only be neighbours if they are assigned different colours in the checkerboard pattern is upheld. This makes the two algorithms compatible such that they can be used interchangeably.

7.2.1. Rewired Irregular Data Structure

The spin states are stored in the same way as it is done for the Metropolis algorithm. The `unsigned char` array S is once again bound to a texture reference for 2D simulations, enabling the CUDA kernels to utilise the on-chip texture cache when reading the spin values of neighbouring cells. 3D simulations have to directly read the spin values from global memory.

Cells not modified during the rewiring procedure again infer their neighbours implicitly from their own coordinates. The neighbour information for modified cells needs to be stored explicitly. Two arrays are used to do this. Vertex array V stores the length of the cell's adjacency-list, and arc array A stores the actual neighbour IDs. A thread with thread ID tid processes the cell whose adjacency-list length is stored at index tid in V. Respectively, the first neighbour ID is stored at index tid in array A. The second neighbour ID is stored at position $tid + |V_m| +$ padding and so on, where $|V_m|$ is the number of modified cells and padding increases $|V_m|$ to the next multiple of 32 (i.e. 128 bytes for 4 byte integers).

Storing the neighbours in this order means that memory transactions from the vertex and arc arrays can be fully coalesced, as sequential threads can access sequential elements. While this improves performance, it increases the memory requirements for the arc array to $|V_m| \times degree(max) \times sizeof(int)$, up from the $|V_m| \times degree(average) \times sizeof(int) + |V_m|$ required by the data structure used for the Metropolis algorithm. Depending on the rewiring probability and system size, this difference can be very significant and in some cases makes it possible to use the Metropolis algorithm where the Wolff algorithm would run out of memory. The disadvantage of uncoalesced memory reads when accessing the arc array in the implementation of the Metropolis algorithm is also somewhat mitigated by the L1/L2 cache structure on Fermi-architecture based devices. However, the performance results in Section 7.2.3 show that the implementation of the Wolff algorithm needs every bit of performance that it can get, which is why this data structure has been chosen for it.

The ID of the cell processed by a thread with ID tid is stored in arrays M_u and M_m for unmodified and modified cells respectively. This is necessary because different CUDA kernels are used to process the two types of cells to avoid conditionals that would cause warp divergence and thus hurt performance. The following section describes the kernel implementations in detail.

7.2.2. The CUDA GPU Implementation

The implementation of the Wolff cluster update algorithm for rewired Ising models uses a number of CUDA kernels to perform the updates in parallel. Algorithm 48 describes the tasks performed by the host system that are necessary to advance the simulation. Some of the kernels are executed by T threads, where T depends on the graphics hardware as described in previous chapters. The implementation uses 2D or 3D thread blocks based on the number of system dimensions. Fermi-architecture based devices work well with blocks of size 32×16 and $32 \times 4 \times 4$, whereas blocks of size 16×16 and $16 \times 4 \times 4$ work best on GT200 series devices.

The total number of cells $|V|$ is the sum of the number of unmodified cells $|V_u|$ and the number of modified cells $|V_m|$. It is equal to the system size N. Note that $<<$ and $>>$ are used to denote left and right binary shift operations respectively. The base 2 logarithm of the dimension length L used by the CUDA kernels is calculated once by the host and passed as parameter to the kernels. The random numbers generated on the

Algorithm 48 Evolve the model by `STEPS` simulation steps. This is the host code that manages the GPU.

allocate device memory for arrays $R, V, A, S, M_u, M_m, M2, X, F, P$
copy initial spin values to the device
copy RNG seeds to the device
do in parallel on the device using T threads: initialise the RNGs
for $i \leftarrow 1$ to `STEPS` **do**
  $k \leftarrow$ randomly pick a cell
  $s_k \leftarrow$ copy k's spin to the host and flip it
  do in parallel on the device using T threads: generate $|V_m|$ packed random results and store them in R
  do in parallel on the device using T threads: call Kernel 1 for unmodified cells
  do in parallel on the device using $|V_m|$ threads: call Kernel 1 for modified cells
  $F \leftarrow 0$ //initialise frontier array F to 0
  $F[k] \leftarrow 1$ //set cell k as active in the frontier
  $P \leftarrow 0$ //initialise processed array P to 0
  repeat
    $c \leftarrow 0$ //flag indicating changes
    if `local` Kernel 2 is enabled **then**
      do in parallel on the device using $|V|$ threads: call `local` Kernel 2
    end if
    do in parallel on the device using $|V_u|$ threads: call Kernel 2 for unmodified cells
    do in parallel on the device using $|V_m|$ threads: call Kernel 2 for modified cells
  until $c = 0$
end for
copy final spins back to the host

device use Marsaglia's lagged-Fibonacci RNG implementation for CUDA as described in Chapter 4. Every CUDA thread uses its own lag-table to produce an independent stream of random numbers, thus avoiding read-write race conditions. However, for the implementation to perform well, it is necessary that all threads in a half-warp collectively generate new random deviates, even if some of them are not actually used. This enables global memory reads and writes to be coalesced.

When a Wolff cluster is formed around a source cell k, neighbouring cells with the same spin value are only included in the cluster if the arc (i.e. the edge in one direction) used to reach them is active in this simulation step. This is determined by generating a uniform random number and comparing it to probability $\text{pWolff} = 1 - e^{-2\beta}$. Kernel 1 for unmodified cells, described in Algorithm 49 and executed by T threads, generates the random deviates itself. The random numbers for Kernel 1 for modified cells, which is described in Algorithm 51 and gets executed by one thread for each cell, are generated beforehand by a dedicated RNG kernel.

This dedicated RNG kernel not only generates the deviates, but also performs the comparison to pWolff. It only stores a bit value for every arc, 1 indicating that it is active and 0 that it is inactive. These bits are packed into a 32-bit `unsigned int` per cell, thus allowing a maximum degree of 32. The packed results are stored in array R. The same number of random deviates are generated for each cell, independent from its actual degree, to ensure that global memory accesses remain coalesced. As every spin i requires $|A_i|$ of these bit values, where $|A_i|$ is its degree, the maximum degree of any cell in the system is used. For example, if the cell with the most neighbours has a degree of 8, then every thread generates 8 random deviates, compares each of them to pWolff and packs the resulting bits into the value that gets written to R. The straightforward implementation would be to do this in an inner-loop and to provide the maximum degree as a parameter to the kernel. However, loop instructions create some overhead and unrolling a loop can improve the performance of the kernel as long as it does not increase the instruction count and register usage excessively. When a loop is unrolled, the loop instructions are removed and the instructions from the body of the loop are instead explicitly repeated. Algorithm 50 shows how this can be done with a template function to avoid having to manually implement it for all possible values of the maximum degree.

The task of Kernel 1 is to determine which arcs not only connect cells with the same spin value but

Algorithm 49 This is CUDA Kernel 1 for unmodified cells in 2D. It is executed by T threads, each of which processes a number of cells until all $|V_u|$ unmodified cells have been processed. S is the spin-array, M the thread ID to cell ID mapping array, X the traversable array, L the dimension length and D the number of dimensions.

```
t ← global thread ID queried from the runtime
load variables for the RNG instance used by this thread from global memory to registers
for all tid ∈ V_t := {t, t+T, t+2T, ... : t < |V_u|} do
    //compare random numbers to probability pWolff and store the result as a bit-value in r
    r ← 0
    for i ← 1 to 2D do
        r_tmp ← generate uniform random number
        r ← (r << 1) OR (r_tmp < pWolff )
    end for
    id ← M[tid] //load the cell ID
    ix ← id AND (L − 1) //the x-coordinate
    iy ← id >> log2(L) //the y-coordinate
    s ← S(ix, iy) //load the spin value
    //calculate the neighbour coordinates and mark traversable arcs in x
    x ← 0 //bits marking active arcs
    n_ix ← ix = 0 ? L − 1 : ix − 1
    n_s ← S(n_ix, iy) //spin value for left neighbour
    x ← (s = n_s) and (r AND (1 << 0)) ? (x OR (1 << 0)) : x
    n_ix ← ix = (L − 1) ? 0 : ix + 1
    n_s ← S(n_ix, iy) //spin value for right neighbour
    x ← (s = n_s) and (r AND (1 << 1)) ? (x OR (1 << 1)) : x
    and so on for all other dimensions ...
    X[tid] ← x //write the bit-mask for traversable arcs to global memory
end for
write the updated local variables for the RNG instance back to global memory
```

also pass the random comparison with value pWolff. Arcs that pass this test are called traversable and are marked in array X using the bit value 1, arcs that fail the test are marked with value 0. Separate traversable arrays of length $|V_u|$ and $|V_m|$ are used for Kernel 1 for unmodified and modified cells respectively to avoid uncoalesced memory access. The two versions of Kernel 1 can be executed in parallel using different CUDA streams on hardware that supports this feature.

Kernel 2 uses the traversable array to spread the new spin value in a breadth-first search manner. Starting from the source cell, it advances a frontier F across eligible connections to other cells, marking visited cells in P, until no more traversable arcs can be reached. At this point, all cells in the current Wolff cluster have been updated with the new spin value. Kernel 2 for unmodified cells is described in Algorithm 52. The version for modified cells is essentially the same as the one for unmodified cells, only that it loads the neighbour information from global memory as described in Algorithm 51. Both versions of Kernel 2 can run concurrently using different CUDA streams.

Since this implementation of Kernel 2 only advances the frontier by one step from the source cell during every kernel call, it does not perform very well once the Wolff clusters increase in size. To speed-up the propagation of the new spin value throughout the cluster, calls to a `local` version of Kernel 2 can be interleaved with the previously described `global` versions. In this `local` version of the kernel, the threads of a thread block use the fast shared memory of the streaming multi-processors to quickly propagate the spin to all cells processed by threads of the same block that are also part of the Wolff cluster. Algorithm 53 describes this process. As the kernel is executed by $|V|$ threads, the traversable array can not be accessed by thread ID as it is done in the `global` implementation. An additional mapping array $M2$ is used to map the thread ID to the correct index into the traversable array X. The `global` versions for unmodified and modified cells need to be called after one of these optional calls to the `local` kernel in order to advance the frontier across thread block boundaries and rewired edges. The performance analysis in Section 7.2.3 shows when it is beneficial to use the `local` kernel in addition to the `global` updates.

Algorithm 50 The dedicated RNG kernel generates random deviates, compares them to pWolff and packs the resulting bit values into an `unsigned int` for later use by Kernel 1 for modified cells (see Algorithm 51). Each one of the T threads used to execute this kernel generates the results for $count = |V_m|/T$ spins. The `DEGREE_MAX` iterations of the inner-loop are unrolled to improve the performance.

```
template<unsigned int DEGREE_MAX>
__global__ void rng_kernel(count, pWolff, R, <RNG parameters>) {
  ...
  for (int i = 0; i < count; ++i) {
    unsigned int result = 0;
    #pragma unroll
    for (unsigned int packIter = 0; packIter < DEGREE_MAX; ++packIter) {
      float myRand = rng_uniform(<...>); // generate the random deviate
      result = (result<<1) | (myRand < pWolff);
    }
    R[tid+i*T] = result;
  }
  ...
}

// host code kernel calls
switch (degreeMax) { //degreeMax is the maximum degree in the current lattice
  case 4 :
    rng_kernel<4><<<gridSize,blockSize>>> (count, pWolff, R, <RNG params>);
    break;
  ...
  case 32 :
    rng_kernel<32><<<gridSize,blockSize>>> (count, pWolff, R, <RNG params>);
    break;
}
```

Algorithm 51 This is CUDA Kernel 1 for modified cells in 2D. Every CUDA thread processes one cell, thus $|V_m|$ threads are executed to process all modified cells. S is the spin-array, V the vertex-array, A the arc-array, M the thread ID to cell ID mapping array, R the random results array, X the traversable array and L the dimension length.

$tid \leftarrow$ global thread ID queried from the runtime
$id \leftarrow M[tid]$ //load the cell ID
$ix \leftarrow id$ AND $(L-1)$ //the x-coordinate
$iy \leftarrow id >> log2(L)$ //the y-coordinate
$s \leftarrow S(ix,iy)$ //load the spin value
$r \leftarrow R[tid]$ //load the random results for this cell
//look-up the neighbours and mark traversable arcs in x
$r_{shift} \leftarrow 0$ //shift r by r_{shift} bits
$x \leftarrow 0$ //bits marking active arcs
$v \leftarrow V[tid]$ //load the adjacency-list length
for all $n\_id \in \{A[tid], A[tid+|V_m|], \ldots, A[tid+(v-1)|V_m|]\}$ **do**
  $n\_ix \leftarrow n\_id$ AND $(L-1)$ //the x-coordinate
  $n\_iy \leftarrow n\_id >> log2(L)$ //the y-coordinate
  $n\_s \leftarrow S(n\_ix, n\_iy)$ //load the neighbour's spin
  $x \leftarrow (s = n\_s)$ and $(r$ AND $(1 << r_{shift}))$? $(x$ OR $(1 << r_{shift})) : x$
  $r_{shift} \leftarrow r_{shift} + 1$ //increment the shift value
end for
$X[tid] \leftarrow x$ //write the bit-mask for traversable arcs to global memory

A small modification to Kernel 2 can substitute the conditional branches that decide if a neighbouring cell is set as active in the frontier array F by evaluating the traversable value stored in x with atomic operations of the form:

$$\text{atomicOr}(F[\ldots], ((x >> \{0,1,\ldots,2D-1\}) \text{ AND } 1))$$

This gives a comparable performance on Fermi-architecture based devices, but is considerably slower on

Algorithm 52 This is CUDA Kernel 2 for unmodified cells in 2D. Every CUDA thread processes one cell, thus $|V_u|$ threads are executed to process all unmodified cells. S is the spin array, M the thread ID to cell ID mapping array, X the traversable array, F the frontier array, P the processed array, L the dimension length, c a flag indicating changes in the cluster size and s the new spin value for all cells in the cluster.

```
tid ← global thread ID queried from the runtime
id ← M[tid] //load the cell ID
if F[id] and (NOT P[id]) then
    //this cell is part of the Wolff cluster
    P[id] ← 1 //mark as processed
    c ← 1 //set the flag to indicate the changes
    ix ← id AND (L − 1) //the x-coordinate
    iy ← id >> log2(L) //the y-coordinate
    S(ix, iy) ← s //store the new spin value
    x ← X[tid] //load the traversable flags
    //set nbrs as active in F if the arc is traversable
    n_ix ← ix = 0 ? L − 1 : ix − 1
    if x AND (1 << 0) then
        F[iy ∗ L + n_ix] ← 1 //frontier for left nbr
    end if
    n_ix ← ix = (L − 1) ? 0 : ix + 1
    if x AND (1 << 1) then
        F[iy ∗ L + n_ix] ← 1 //frontier for right nbr
    end if
    and so on for all other dimensions ...
end if
```

the older GT200 series cards, clearly highlighting the increased performance of atomic operations on the newer architecture. But because it does not provide any performance gain either, the implementation based on conditionals that performs well on both hardware generations is used.

An implementation of Kernel 2 that is based on labelling all components in the graph using a variation of the component labelling algorithm for arbitrary undirected graphs, described in Section 6.1, has also been tested. The component labelling algorithm has been modified to work for directed graphs based on the traversable array. This implementation requires an additional CUDA kernel, which flips the spins of the cluster that contains the source cell after all components have been detected. The hypothesis was that this algorithm would perform better than the breadth-first search based implementation when the Wolff clusters are relatively large on average, as it takes a few iterations for the current implementation to extend the frontier far enough to really utilise the parallel compute capabilities of the hardware. However, it turns out that this is not the case. The implementation described here performs better even for relatively low temperatures and therefore larger clusters.

7.2.3. Performance Results

The performance of the Wolff update algorithm depends on a number of factors. First it is necessary to decide if Kernel 2 propagates a new spin value with global updates only or if additional local updates in shared memory reduce the execution time. Then the relationship between system energy and processing time is tested. Finally, the performance scaling over several length scales of the rewiring probability p and dimension length L are evaluated and compared to the CUDA implementation of the Metropolis algorithm. All results presented in this section are averaged over 10 simulation runs. Each run is initialised with a random seed value.

Figure 7.8 compares the performance of Kernel 2 when using only global updates to the approach with additional local updates. The sudden drop in the execution times marks the phase transition from the more ordered state with large clusters of like spins found below the critical temperature to the random system configurations found above T_c. While the local updates come at the cost of an additional kernel call during

Algorithm 53 The shared memory propagation of the frontier performed by the `local` version of Kernel 2. The parameters and the local variables id, ix and iy are the same as in Algorithm 52. $M2$ is the thread ID to traversable index mapping and B the block length. $|V|$ CUDA threads are used.

```
btid ← block thread ID queried from the runtime
F_s[B*B] //frontier array in shared memory
X_s[B*B] //traversable array in shared memory
F_s[bid] ← F[id] //load the frontier into shared memory
X_s[btid] ← 0
if the cell has no rewired edges then
    idx ← M2[tid] //look-up traversable array idx
    X_s[btid] ← X[idx] //load the traversable flags
end if
repeat
    f_old ← F_s[btid] //remember the current value
    if btid = 0 then
        vote_s ← 0 //shared memory variable for votes
    end if
    synchronise threads in block
    //determine the shared memory index n_idx for the neighbours and set the cell as active in the frontier if the neighbour is active and the arc is traversable
    n_idx ← threadIdx.y * B + threadIdx.x + 1 //right
    F_s[btid] ← (threadIdx.x < (B − 1)) and (X_s[n_idx] AND (1 << 0)) and F_s[n_idx] ? 1 : F_s[btid]
    n_idx ← threadIdx.y * B + threadIdx.x − 1 //left
    F_s[btid] ← (threadIdx.x > 0) and (X_s[n_idx] AND (1 << 1)) and F_s[n_idx] ? 1 : F_s[btid]
    and so on for all other dimensions ...
    if X_s[btid] ≠ 0 and f_old ≠ F_s[btid] then
        vote_s ← 1 //the frontier changed
    end if
    synchronise threads in block
until vote_s = 0
if F_s[btid] and (NOT P[id]) then
    F[id] ← 1
    //process unmodified cells as described in Alg. 52
end if
```

each step of the update process, they attempt to make up for it by quickly propagating the new spin value over multiple edges in fast shared memory, thus requiring fewer steps than the global updates only approach. This particularly improves the performance when the clusters are large. The results indicate that the local kernel should always be used for 2D simulations. For 3D simulations, the overhead introduced by the local updates does not pay off at high temperatures. The larger number of edges in 3D means that, compared to a 2D system, fewer calls of Kernel 2 are required to propagate the new spin values no matter which approach is used. When the clusters are small and therefore the number of iterations is small to begin with, then the local updates can not improve the situation enough to justify their overhead. But because the local updates in no situation slow things down by a significant amount when very close to T_c, which is the temperature region of interest for this study, but have the potential of speeding the algorithm up dramatically under other circumstances, all following results of the Wolff algorithm make use of the local version of Kernel 2.

The effects of the cluster size on the performance are also visible in Figure 7.9, which shows the performance of the algorithm during the equilibration phase. As the system approaches the equilibrium state for a temperature $T \lessapprox T_c$, the average cluster size increases and the execution time goes up. The insets show how the system energy changes over the course of the simulation. Compared to the Metropolis algorithm, the Wolff algorithm takes many more update steps to equilibrate the system. The clusters in the initial chaotic configuration are small and thus changes to individual clusters only have a very minor effect on the macroscopic scale. To make things worse, each individual update step of the Metropolis algorithm takes significantly less time than the CUDA implementation of the Wolff algorithm during all phases of the simulation. However, as discussed before, the Wolff algorithm does not suffer from critical slowing down

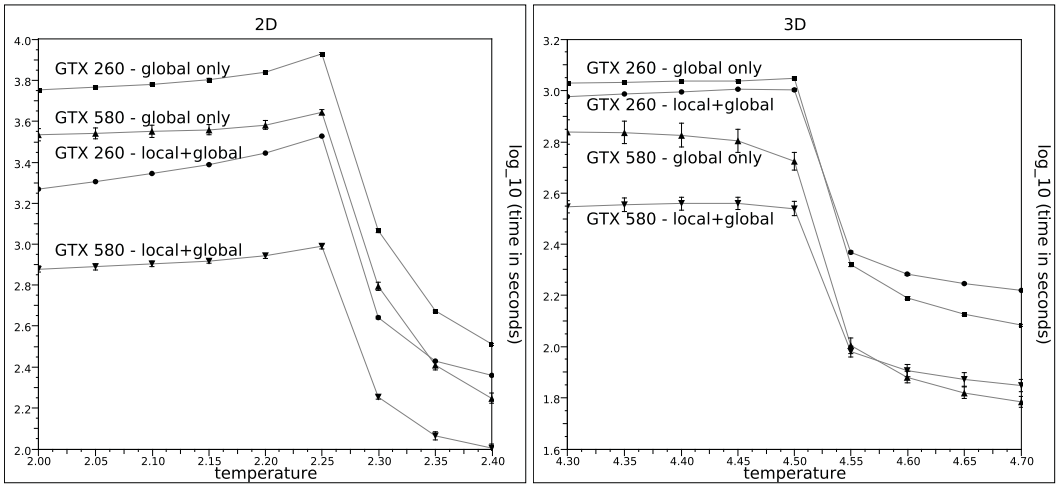


Figure 7.8.: The plots show how the local version of Kernel 2, which propagates new spin values in shared memory, compares to the global only version at different temperatures. The system size is $N = 2048^2$ for the 2D systems (left) and $N = 128^3$ for the 3D systems (right). The rewiring probability is $p = 10^{-4}$. The system is equilibrated for 4000 simulation steps using the Metropolis algorithm before the measurements are performed over 16384 simulation steps. Error bars are used to represent the standard deviation.

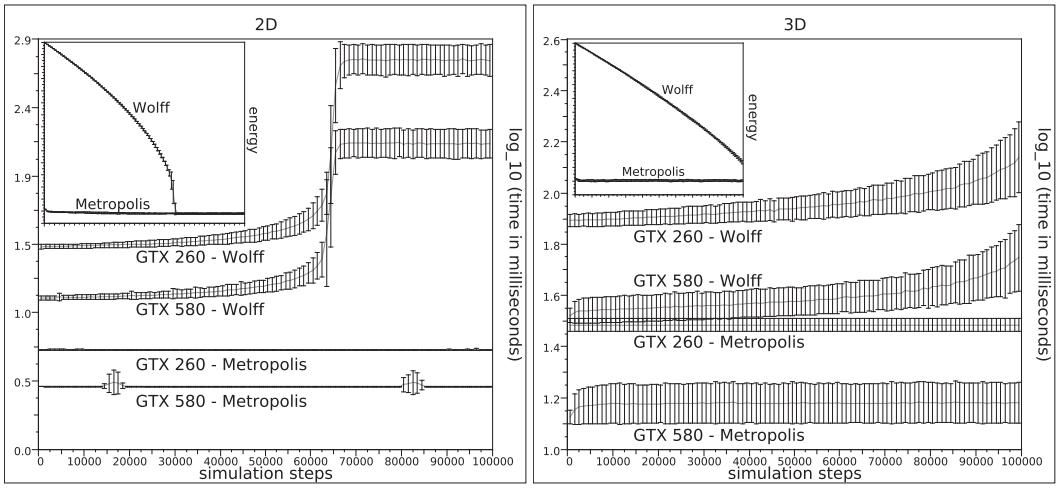


Figure 7.9.: The plots show how the execution time of the Wolff algorithm changes when the system approaches its equilibrium value for 2D (left) and 3D (right) simulations. The system size is $N = 1024^2$ for the 2D systems and $N = 128^3$ for the 3D systems. The rewiring probability is $p = 10^{-4}$. The execution time is measured every 10 simulation steps and the results represent the time since the last measurement. The insets illustrate the related change in system energy. The 3D system with Wolff updates does not reach its equilibrium value in the $100,000$ system updates performed. The results are averaged over 100 consecutive x-values. Error bars are used to represent the standard deviation.

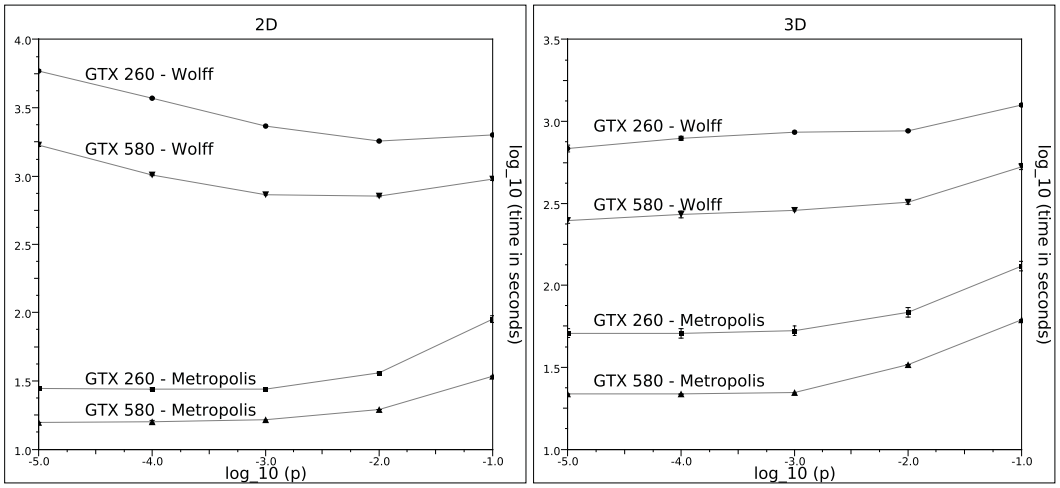


Figure 7.10.: The plots illustrate how the rewiring probability p affects the execution times of the Wolff update algorithm in 2D (left) and 3D (right) simulations. The temperature and system size are set to $T = 2.269$ and $N = 2048^2$ for the 2D systems and to $T = 4.5115$ and $N = 128^3$ for the 3D systems. The system is equilibrated for 4000 simulation steps using the Metropolis algorithm before the measurements are performed over 16384 simulation steps. Error bars are mostly smaller than the the symbol size.

Table 7.2.: The number of system updates per second (SUPS) is used to compare the performance of the Wolff and Metropolis algorithms. The results are for system size $N = 4096^2$ with temperature $T = 2.269$ and for $N = 256^3$ with $T = 4.5115$ respectively. The rewiring probability $p = 10^{-4}$. The quoted slopes are for the least squares linear fits to the data sets on a $\log_{10}(x) - \log_{10}(y)$ scale, unlike the $\log_2(x) - \log_{10}(y)$ scale used for illustration purposes in Figure 7.11. Only the last three data points of each data set are used to calculate the slopes, as smaller system sizes do not fully utilise the graphics hardware.

Compute Device	**2D Simulation**		**3D Simulation**	
	SUPS	**Slope**	**SUPS**	**Slope**
GTX 260 - Metropolis	157.7	1.78	40.2	2.95
GTX 260 - Wolff	1.0	2.06	1.8	3.35
GTX 580 - Metropolis	265.3	1.86	104.8	2.79
GTX 580 - Wolff	3.5	2.13	5.7	3.25

in the same way as the Metropolis algorithm. Sections 7.5.1 and 7.5.2 show whether this is enough to make up for the performance difference.

Figure 7.10 illustrates how the Wolff algorithm scales with the rewiring probability p. It is interesting to observe that for 2D simulations the execution times of the Wolff algorithm decrease as the value of p increases, whereas this behaviour is reversed for 3D simulations. While the less regular graph structure that is the result of higher values of p generally slows things down, it can help Kernel 2 to propagate the spin value in fewer steps, thus speeding the process up. The 3D simulation already has the upper hand in this process due to the larger number of connections and therefore does not benefit to the same extent from the rewired links.

To round things off, Figure 7.11 shows how the algorithms scale with increasing dimension length L. Unlike the Metropolis algorithm, the Wolff algorithm once again benefits from a larger number of connections and thus achieves a higher performance in 3D than in 2D. The performance difference between the algorithms, as well as the slopes of the least square linear fits to the data sets, are reported in Table 7.2.

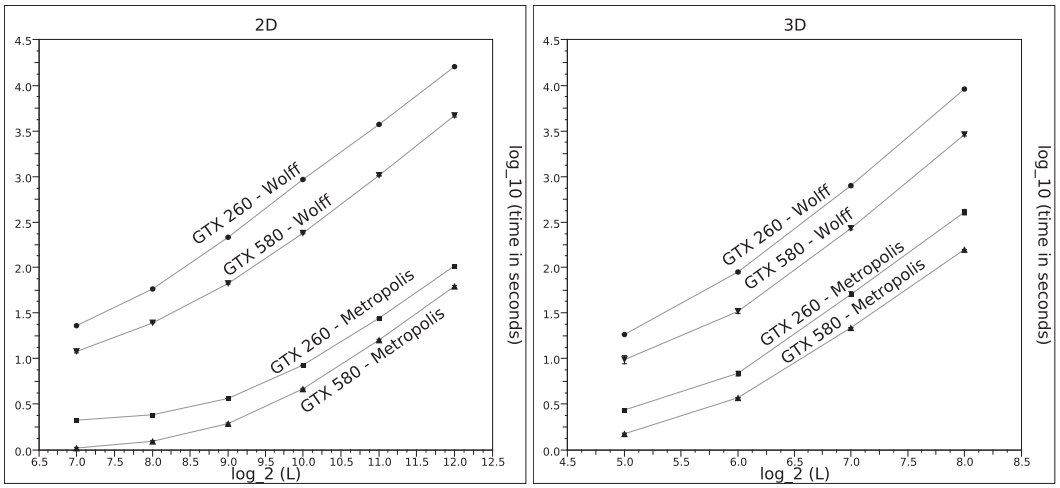


Figure 7.11.: The plots show how the performance of the Wolff update algorithm scales with the system size $N = L^2$ for 2D (left) and $N = L^3$ for 3D (right) simulations. $L = \{128, 256, \ldots, 4096\}$ in 2D and $L = \{32, 64, 128, 256\}$ in 3D. The temperature is set to $T = 2.269$ (2D) and to $T = 4.5115$ (3D). The rewiring probability $p = 10^{-4}$. The system is equilibrated for 4000 simulation steps using the Metropolis algorithm before the measurements are performed over 16384 simulation steps. Error bars are mostly smaller than the symbol size. Note the different log-scales on the x- and y-axis.

7.3. Visualisation of the Rewired Ising Model

The ability to visualise [4] a simulation of a complex system in real-time is often very useful to a scientific modeller. It can be used to reassure oneself that the simulation is working correctly, as it is sometimes much easier to spot irregularities or artifacts visually than it is to detect them in a purely numerical output. And once the simulation is working correctly, an interactively running model enables the scientist to experiment with different parameter combinations prior to committing computational resources to detailed statistical investigations. This might lead to a better insight into a particular phenomena or it might lead to a hypothesis or explanation as to a numerical observation such as a phase transition or other critical phenomena.

While it is usually relatively straight forward to visualise a 2D system, it is more difficult to "see inside" a block of 3D data that is associated with a simulation model. This is a common problem for many physical and engineering models in areas such as materials science, chemical engineering, fluid dynamics or medical imaging. Volume rendering is a long standing problem [165] in computer graphics. A number of approaches for seeing into volumes [166], for providing visual cues into volumes [167] and for identifying surfaces present in the interior of a data volume [168] have been explored.

The results presented here focus on the high-performance, interactive visualisation of the small-world rewired Ising model in two and three-dimensions. GPUs provide most of the computational power required to run not only the Ising model simulation, based on the Metropolis and Wolff update algorithms presented in the previous sections, but also to fulfil their more traditional task of graphically rendering the model. The synergistic effects of using CUDA and OpenGL together are used to reduce the amount of data that needs to be copied between the host machine and graphics accelerator, thus improving performance.

A number of techniques are discussed that enable the user to see inside a 3D block of data by teasing it apart in layers. Various dynamically computed metrics that can be plotted live, superposed on the

[4] This section extends on work first published in [163] **A. Leist**, D. P. Playne, and K. A. Hawick, "Visualising spins and clusters in regular and small-world Ising models with GPUs," in *Procedia Computer Science*, vol. 1, no. 1, May 2010 and in [164] **A. Leist**, D. P. Playne, and K. A. Hawick, "Interactive Visualisation of Spins and Clusters in Regular and Small-World Ising Models with CUDA on GPUs," *Journal of Computational Science*, vol. 1, no. 1, pp. 33–40, May 2010.

Algorithm 54 Pseudo-code describing the major tasks performed to generate a single frame.

Input parameters: *STEPS* simulation steps are performed before the visualisation is updated (default 1). *N* is the system size.
for $i \leftarrow 1$ to *STEPS* **do**
  do in parallel on the device: evolve simulation and collect data for the energy, magnetisation and spin changes
end for
do in parallel on the device using *N* threads: update cell colours vertex buffer object
render cells with OpenGL
render links, plots, etc. with OpenGL if required

visualisation, are used to help the user build up an intuition as to the meanings of parameters and the evolutionary processes in the model. Visualising the evolving simulated system is of great value in exploring how individual small-world links change its properties.

Even in the case of a regular Ising system it is not always trivial to understand what is happening when a cluster algorithm like that of Wolff is applied. Individual clusters can be highlighted in real-time or while manually advancing the simulation step by step to help build up an intuition of how the cluster formation scales affect the measured numerical statistics.

7.3.1. Implementation

The visualisation of the Ising model simulation extends the CUDA implementations of the Metropolis and Wolff cluster update algorithms. OpenGL is used for the visualisation of the simulation and rendering of additional information, such as graphs that show certain system properties and how they change over the duration of the simulation. Algorithm 54 describes the major steps performed by the system.

One of the main tasks is to visualise the current spin value of every cell using colour coded points or cubes. As CUDA is used to perform the actual simulation on the GPU, the spin values reside in graphics device memory. Using CUDA to also update the colour vertex buffer object (VBO), which is used by OpenGL to draw the vertices that make up the cells in the desired colours, is therefore an obvious step. Not only does this mean the massively parallel computational power of the GPU can be exploited once more, but it also means that the spin and colour data always stays in graphics device memory and does not need to be transferred back to host memory. Algorithm 55 shows the CUDA-OpenGL interoperability as well as the actual CUDA kernel implementation used to update the cell colours in a 3D simulation.

The original simulation code was extended to set the `MASK_FLIPPED` bit described in Algorithm 55 and to obtain data which is required to generate real-time plots for the fraction of spins flipped in the previous simulation step, as well as the current energy and magnetisation. Algorithm 56 illustrates how these changes are implemented.

7.3.2. High Performance Simulation and Visualisation

This section showcases how the combination of parallel, high performance simulation and visualisation can provide important insights into both the state of the system at a particular simulation step as well as the ongoing real-time state changes. In addition to visualising the current spin of every cell graphically using different colours, a number of live-plots show how certain system properties change over time. There are currently plots for the magnetisation, energy, and fraction of spins flipped in a single simulation step, which is equal to the cluster size when using Wolff's algorithm. There are also two graphs showing the standard deviations for the energy and fraction of flipped spins.

Figure 7.12 illustrates two 2D Metropolis Ising simulations with fixed temperatures after 1000 simulation steps. The system on the left uses a regular lattice with periodic boundaries, while the second image shows a lattice that had a small fraction of its edges randomly rewired. Several metrics are calculated after

Algorithm 55 The host function `updateColours` maps the VBO that specifies the vertex colours to OpenGL into the CUDA address space and executes the CUDA kernel `colours_kernel_3d`. This kernel updates the values according to the current spin values of the individual cells. The `MASK_FLIPPED` bit is set on spins that have been flipped since they have last been visualised and is used to highlight this recent change (see Figure 7.15).

```
void updateColours(cudaPitchedPtr spin, GLuint coloursVboId,
                   int dimXLen, int dimYLen) {
  cudaGLRegisterBufferObject(coloursVboId); // register VBO to CUDA
  float4* d_colours;
  // map VBO into CUDA namespace
  cudaGLMapBufferObject((void**)&d_colours, coloursVboId);

  colours_kernel_3d<<<gridSize,blockSize>>>(spin, d_colours, dimXLen, dimYLen);
  cudaThreadSynchronize();// block until the device has completed
  cudaGLUnmapBufferObject(coloursVboId); // unmap VBO
  cudaGLUnregisterBufferObject(coloursVboId);
}

__global__
void colours_kernel_3d(cudaPitchedPtr spin, float4* colours,
                       int dimXLen, int dimYLen) {
  const Uint tid = (blockIdx.y * gridDim.x + blockIdx.x) *
                    blockDim.x + threadIdx.x; // unique thread ID
  const Uint ix = tid % dimXLen; // the cell's x-coordinate
  const Uint iy = (tid / dimXLen) % dimYLen; // the cell's y-coordinate
  const Uint iz = tid / (dimXLen*dimYLen); // the cell's z-coordinate

  // accessing the 3-dimensional spin array
  size_t slicePitch = spin.ysize * spin.pitch;
  char* slice = (char*)spin.ptr + iz * slicePitch;
  Uchar* row = (Uchar*)(slice + iy * spin.pitch);
  Uchar myValue = row[ix]; // read the spin value including marker bits
  Uchar mySpin = myValue & MASK_SPIN; // remove marker bits

  row[ix] = myValue & (~MASK_FLIPPED);// unset the MASK_FLIPPED bit
  float4 colourSpin0 = myValue & MASK_FLIPPED ?
    make_float4(0.0f, 0.6f, 0.0f, 1.f) : make_float4(0.f, 1.f, 0.f, 1.f);
  float4 colourSpin1 = myValue & MASK_FLIPPED ?
    make_float4(0.35f, 0.35f, 0.35f, 1.f) : make_float4(0.f, 0.f, 0.f, 1.f);
  colours[tid] = mySpin == 0 ? colourSpin0 : colourSpin1; // write cell colour
}
```

every simulation step and visualised using plots to provide information about changes to important system properties. This is a useful tool when investigating a critical phenomena of the system.

The average degree of a cell increases from four on a 2D lattice to six on a 3D cubic lattice, where every cell is connected to the cell in front and behind of itself in addition to the cells on the right, left, above and below. Figure 7.13 visualises a 3D Metropolis Ising simulation with no rewiring after 1000 and after 2000 simulation steps. It also demonstrates how the cube can either be split open layer by layer or completely unfolded into a grid view to gain an insight into what is happening on its inside.

The ability to adjust simulation parameters while the simulation is running and the instant feedback provided by real-time visualisation are helpful tools when analysing a system. They can give new insights into the impact of a certain parameter or combination of parameters on the system properties and behaviour. This can speed-up the process of narrowing the parameters down to the most interesting values before extensive computational resources are committed for detailed statistical investigations. Figure 7.14 demonstrates this by reducing the system temperature in intervals of 250 simulation steps. The effects of these parameter changes on the system properties can immediately be seen on the plotted graphs.

Algorithm 56 This mix of pseudo-code and actual CUDA code describes how the original simulation was extended to count the number of spins flipped in one simulation step, the number of like-like bonds and the ratio of up vs. down spins. The latter two are required to calculate the energy and magnetisation. The results of these metrics can be visualised using real-time graphs.

```
if first thread in thread block then
    initialise shared memory counters to 0
end if
__syncthreads(); //barrier synchronisation for all threads in the same thread block
if spin value changed then
    set the MASK_FLIPPED bit on the spin value //see Algorithm 55
    atomicAdd(&blockFlipped, 1); //increment the shared memory counter for flipped spins
end if
c ← 0 //local counter for the like-like bonds between neighbouring cells
for all neighbouring cells do
    if equal spin values then
        c ← c + 1
    end if
end for
atomicSub(&blockE, c); //shared memory counter for the energy
atomicAdd(&blockM, (currentSpin == 0 ? -1.0 : 1.0)); //shared memory counter for the magnetisation
__syncthreads();
if first thread in thread block then
    //one atomic operation per thread block and counter to write the value to mapped system memory
    atomicAdd(magnetisationData, blockM);
    atomicAdd(energyData, blockE);
    atomicAdd(flippedSpins, blockFlipped);
end if
```

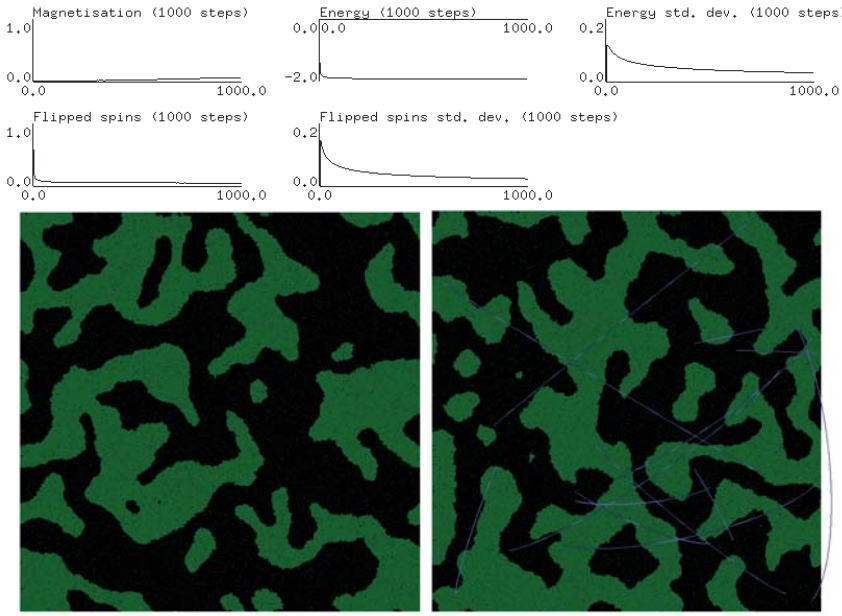


Figure 7.12.: 2D Metropolis Ising model simulation with 2048^2 cells, rewiring probability $p = 0.0$ (left) and $p = 10^{-6}$ (right) and temperature $T = 1.8$ after 1000 simulation steps. The plots show the evolution of the the rewired simulation.

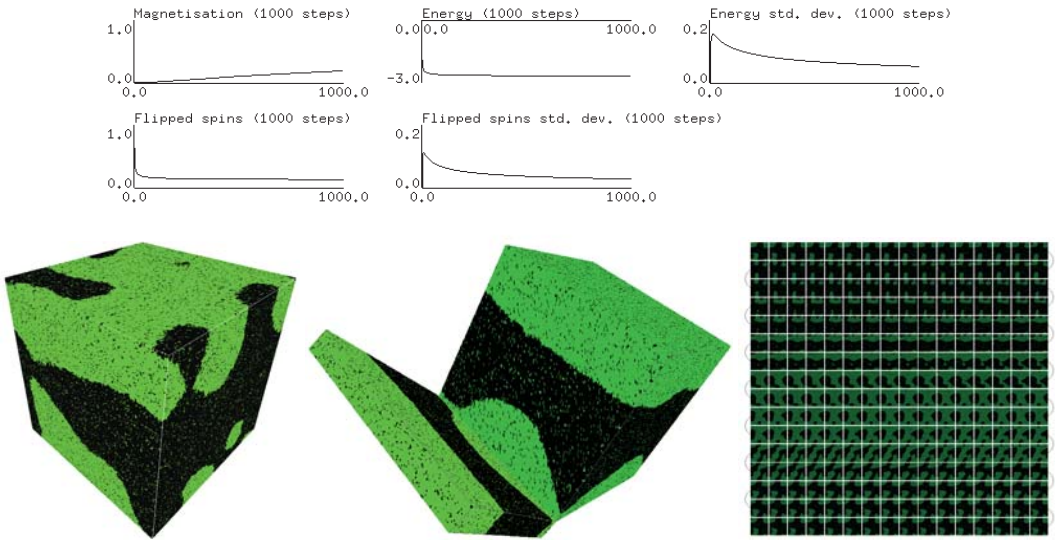


Figure 7.13.: Regular 3D Metropolis Ising model simulation with 256^3 cells and temperature $T = 3.5$ after 1000 (left and right) and after 2000 (middle) simulation steps. The visualisation in the middle demonstrates how the cube can be split open layer by layer to give an impression of what is happening on its inside. The grid on the right arranges the z-layers in a zigzag pattern, thus providing a 2D view of the 3D system. The plots show the system behaviour over the first 1000 steps.

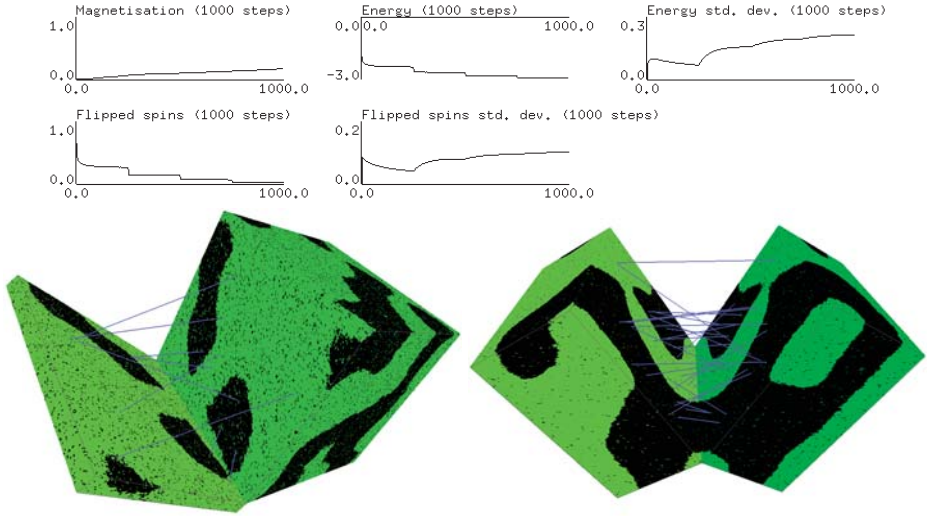


Figure 7.14.: 3D Metropolis Ising model simulation with 256^3 cells, rewiring probability $p = 10^{-6}$ after 500 (left) and 1000 (right) simulation steps. The temperature is initialised to $T = 4.0$ and reduced by 0.5 every 250 simulation steps (i.e. $T = 3.5$ at 250 steps, $T = 3.0$ at 500 steps and $T = 2.5$ at 750 steps). The plots show the system state changes for the first 1000 simulation steps.

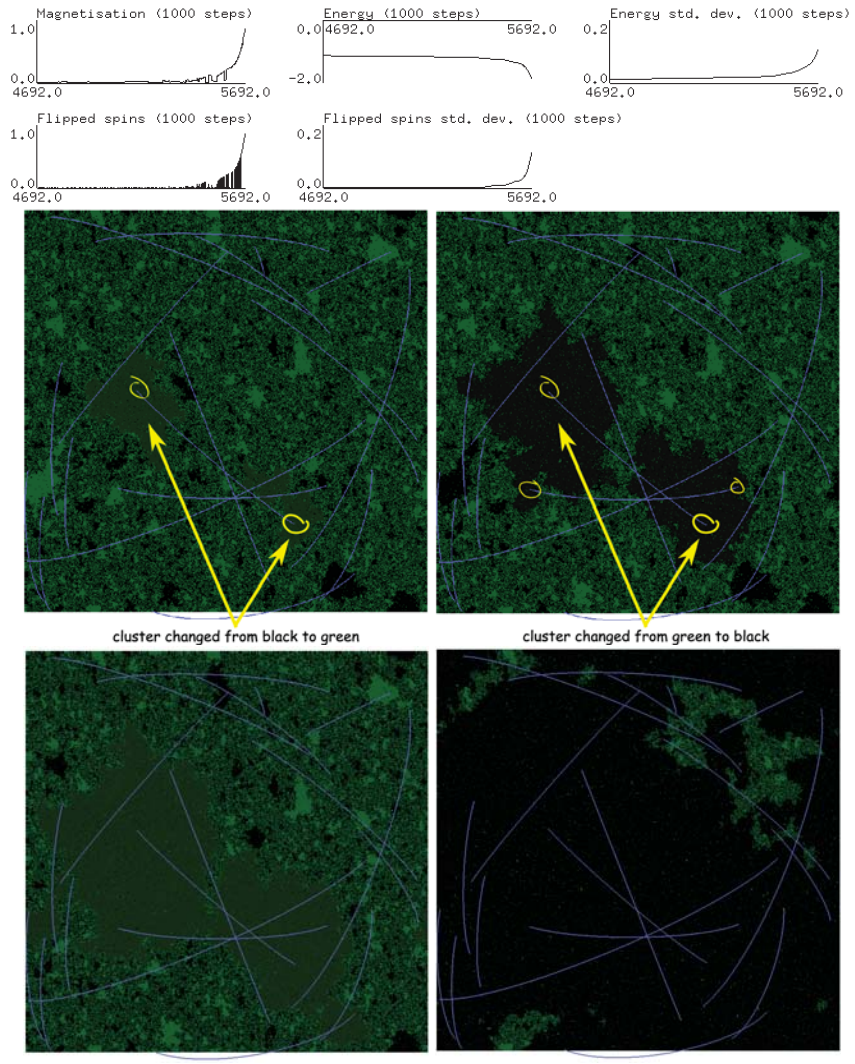


Figure 7.15.: Ising simulation using Wolff cluster updates with 1024^2 cells, rewiring probability $p = 10^{-5}$ and temperature $T = 1.8$. The images show the simulation after 5461 (top left), 5582 (top right), 5632 (bottom left) and 5692 (bottom right) steps. The plots show the system changes of the last 1000 simulation steps. The circles in the first two images highlight the endpoints of the rewired edges which join the two otherwise disconnected clusters. The darker green (top left) and grey (top right) highlight the cells that had their spins flipped to green and black respectively during the previous simulation step. The plots emphasise the much more abrupt state changes of the Wolff cluster updates as compared to the Metropolis updates.

Table 7.3.: Approximate performance data for the Metropolis Ising simulation in simulation steps per second. This is equal to frames per second for the single-GPU implementation. The dual-GPU version uses one device for the simulation and the other one for rendering. The latter renders as many frames per second as it can, possibly skipping a simulation step if it can not keep up with the simulation. The performance was measured for 2D and 3D simulations with no optional features enabled.

Metropolis Ising	**2D ($N = 4096^2$)**		**3D ($N = 256^3$)**	
Performance	**1 GPU**	**2 GPUs**	**1 GPU**	**2 GPUs**
GTX260, $p = 0.0$	10	33	11	23
GTX260, $p = 10^{-5}$	10	31	10	23
GTX580, $p = 0.0$	26	121	23	71
GTX580, $p = 10^{-5}$	26	144	23	71

Wolff's cluster update algorithm performs particularly well near the critical temperature, where the Metropolis Monte Carlo update algorithm is affected by the effect known as critical slowdown. It calculates the thermal probabilities for updating a whole cluster of cells instead of doing this for each cell individually. As shown by the plots in Figure 7.15, Wolff's algorithm gives very abrupt system changes after a long period of only marginal fluctuations. The images demonstrate how Wolff clusters are visualised. The cluster of cells which had their spins flipped during the previous simulation step is highlighted in a different colour that depends on the new spin value. A cluster is only highlighted until a new cluster replaces it in the next simulation step. Not only do the clusters constantly change because the boundary conditions are different every time a cluster is flipped, but Wolff clusters also have a random component that depends on the temperature. It is often a good idea to advance the simulation step by step when it gets close to a phase transition as this can happen very suddenly. The example also gives a good demonstration of how rewiring affects the simulation. If it was not for the single rewired edge in the top left image that has its endpoints highlighted by circles, the two clusters (pointed to by arrows) would be disjoined and the spins of their cells could not have been flipped together.

Table 7.3 gives approximate performance results for the visualisation of the Metropolis Ising simulation for large systems with $N = 4096^2$ or $N = 256^3$ cells. The values are for the bare-bones visualisation with no optional features – like statistics computation or the rendering of rewired links – enabled. The results show that even at this size the visualisation remains fluid and interactive. When only one of the available graphics devices is used for both the CUDA simulation and the OpenGL rendering, then it can only perform one of the two tasks at a time, rendering one frame for every simulation step. This slows the simulation down, as the rendering of large systems takes a considerable amount of time. When the two tasks are assigned to different GPUs, then the simulation can run independently from the rendering, and the rendering device simply skips a step if it can not keep up with the simulation. However, using two devices comes at the cost of having to copy the current spin states from the simulation device to host memory and then onto the rendering device. It is unfortunately not yet possible to copy data directly between two graphics devices[5].

If speed is of paramount importance, then the `colours_kernel_3d` described in Algorithm 55 can be made part of the simulation kernels for the single-GPU implementation, thus avoiding some overhead like the re-computation of cell coordinates. This has not been done here to preserve code readability by keeping the simulation and visualisation code separated.

[5]The upcoming version 4.0 of the CUDA toolkit will introduce GPUDirect™ 2.0, which enables direct communication between multiple GPUs over the PCI-express bus.

Algorithm 57 The steps performed to obtain a scattering pattern of the current spin configuration.

do in parallel on the device using N threads: create complex array from spin values
do in parallel on the device: execute in-place FFT using the CUFFT library
do in parallel on the device using N threads: shift data by [L/2,L/2,L/2] and compute the magnitude
copy data from device to host
write the data to a file

7.4. Spectral Analysis

The magnetisation and energy are not the only properties that give an insight into a particular system state. Spectral methods are another way to analyse the Ising model. The Fourier Transform of the Ising system yields an effective scattering pattern, which in turn reveals the length scales present in the system. This is especially useful because these scattering patterns can be added together to get a meaningful average, which is not possible in the spatial domain. With this it is particularly interesting to study the effects of long range small-world shortcuts to the regular nearest neighbour lattice.

The NVIDIA CUDA Fast Fourier Transform (CUFFT) library, which is part of the CUDA Toolkit, is used to compute the Fourier Transform in parallel on the graphics accelerator. The steps are illustrated in Algorithm 57. First the current spin configuration is written to an array of complex values, using 1 for the "up" spin and -1 for the "down" spin (instead of 0 for the *down* spin as used by the simulation code). The CUFFT plan, which is created once at the beginning of the simulation, is executed and performs the Fourier Transform in-place, using the array of complex values both as input and output. Then the results are shifted by half the length of the system dimensions L, so that frequency zero is at the center of the output. The results are copied to the host and written to a file. These files can be used to compute the average scattering pattern over many independent simulation runs.

Figure 7.16 illustrates the averages of 1000 simulation runs of the 3D Ising model, showing how the spherical mean of the Fourier Transform changes with the number of Metropolis update steps performed. The figure also shows the effects of small-word rewirings, which change the distances between individual cells in the model. The increasing magnitude of the lower frequency components in the regular lattice Ising system (top left) as the simulation progresses represents the growth of tightly packed clusters with like spins. The introduction of small-world rewired long distance links (top right) flattens the distribution of length scales somewhat in comparison to the regular lattice system.

The tightly packed clusters, or lumps of like spins, detected by the FFT are similar to what would be perceived as a cluster when looking at a graphical representation of the system. However, Figure 7.17 shows that the actual size distribution of the connected components looks quite different. After only one simulation step, when the system is still mostly random, almost all the cells are part of one of the two major clusters, one for "up" spins and the other for "down" spins. This is because with six edges connecting every cell to its neighbours (exactly for $p = 0.0$ and on average for $p > 0.0$), it is possible to reach almost all other cells with the same spin value from any given source cell. The spectral analysis does not know about these connections and only looks at the spatial distribution of spin values.

When the temperature T is below or near the critical temperature T_c, then the system becomes less and less random the more update steps are performed. Lumps of like spins form and grow in size, which shows up with an increase in the magnitude of the lower frequencies in the FFT results. The distributions for the size of connected clusters, on the other hand, broaden around the median of the system size until they eventually become bimodal, with distinct peaks for the "up" and "down" spin clusters. A slightly bimodal behaviour is expected very close to the critical temperature due to the fluctuations of the system. A more pronounced bimodal behaviour with clearly distinct peaks, however, suggests that the chosen temperatures are lower than the critical temperature for the system configuration. If the temperatures are slightly increased to $T = 4.52$ for the regular lattice system or to $T = 4.6$ for the rewired system with $p = 10^{-2}$,

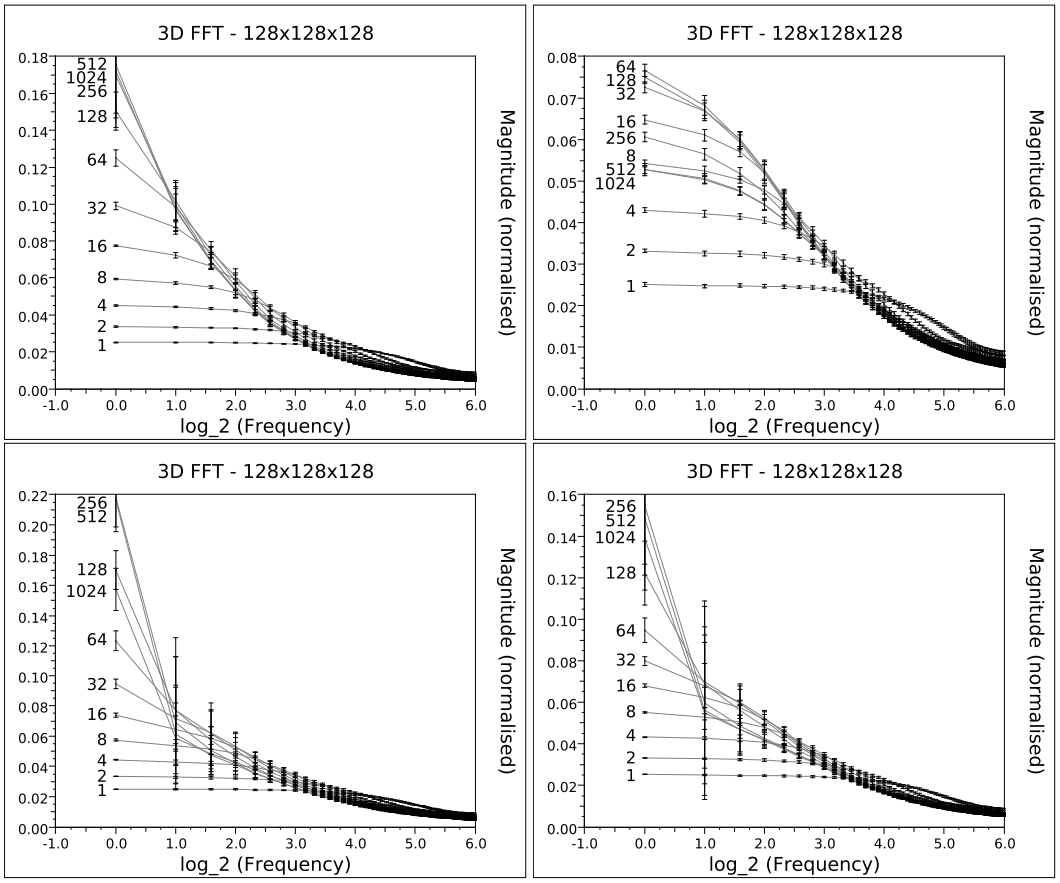


Figure 7.16.: The spherical mean of the 3D FFT computed after $\{1,2,4,\ldots,1024\}$ simulation steps of the Ising model using the Metropolis update algorithm. The system size $N = 128^3$. The plot in the top left shows the results for the regular cubic lattice Ising model with temperature $T_c \approx T = 4.5115$. The other plots illustrate rewired irregular lattice Ising models with rewiring probability $p = 10^{-2}$ and $T_c \approx T = 4.55$, where the arcs were randomly rewired in all dimensions (top right), only the x-dimension (bottom left) or the x- and y-dimensions (bottom right). Each data point represents the mean value of 1000 simulation runs. The error bars illustrate the standard deviations. The error has been rescaled by a factor of 0.1 for frequency $f = 2^0$ of the two plots in the bottom row.

which are above the respective critical temperatures, then the cluster size distributions retain a single peak in the equilibrated system.

The much quicker broadening of the cluster size distributions for $p = 10^{-2}$ as compared to $p = 0.0$ supports the intuitive assumption that the rewired connections facilitate domain growth. While the shape of the regular lattice distribution still changes between steps 1024 and 4096, no significant changes can be observed after 1024 simulation steps for the rewired system. The FFT results may appear to contradict this conclusion, as the low frequencies are less dominant in the rewired system than in the regular lattice system. However, it needs to be considered that a large cluster in the rewired system is likely to consist of two or more spatially separated smaller clusters that are only weakly connected by a few long distance links, which is not picked up by the spectral analysis.

Another interesting effect clearly visible in the results for the rewired system (Figure 7.16, top right) is that after the distributions initially become more and more front-heavy, this effect reverses after about 64 simulation steps and the distributions become flatter again. This is likely due to individual clusters

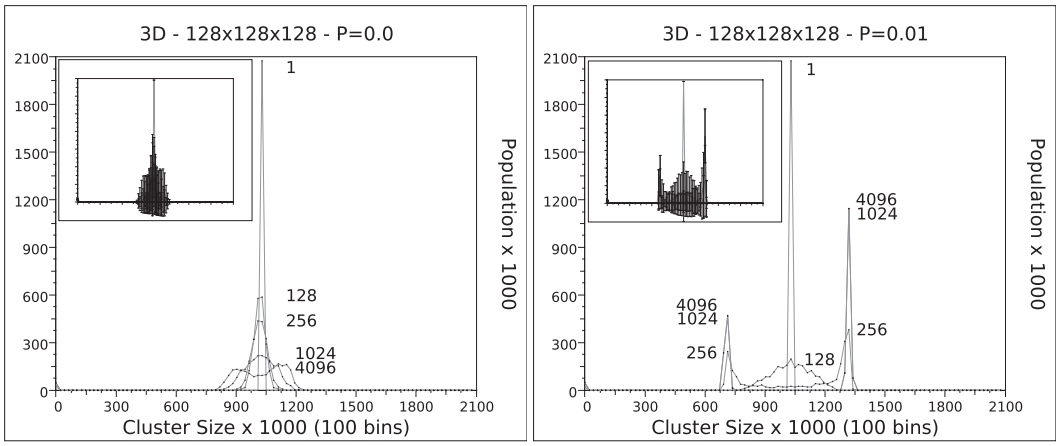


Figure 7.17.: The cluster size distribution of the 3D Ising system computed after $\{1,2,4,\ldots,4096\}$ simulation steps using the Metropolis update algorithm. The system size $N = 128^3$. The temperature for the regular lattice Ising system (left) $T_c \approx T = 4.5115$. The rewiring probability $p = 10^{-2}$ for the small-world rewired Ising system (right) and the temperature $T_c \approx T = 4.55$. Each data point represents the mean value of 1000 simulation runs. The insets show the standard deviations as error bars on the data sets.

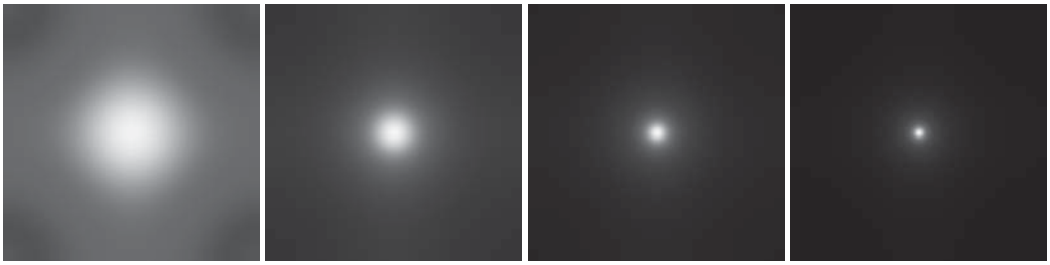


Figure 7.18.: Visualisation of the frequency domain of a regular lattice 2D Ising model after $\{1,4,16,64\}$ simulation steps (from left to right). The system size $N = 1024^2$ and temperature $T = 2.269$. The magnitude is represented by the intensity of the pixels in the grey-scale image, varying from black at the weakest intensity to white at the strongest.

beginning to join together, thus forming larger clusters of different shapes, which changes the scattering patterns produced by the FFT.

The figure also shows the effects of restricting the rewiring procedure to certain dimensions. When edges are only rewired in the x-dimension (bottom left), then this means that the y- and z-coordinates of the current neighbour are "frozen" and only the x-value is randomly modified. The total number of rewired edges remains the same – on average – for a given rewiring probability p, no matter if the selection of new neighbours is restricted to certain dimensions or not. The deviations from the spherical mean values are much higher for the results where only one or two-dimensions were rewired, as opposed to all dimensions or no rewiring at all. The reason for this becomes clear when the FFT results are visualised graphically.

Visualising the data of the FFTs graphically can make it more tangible and thus easier to understand. A typical representation of a 2D FFT is given in Figure 7.18. The visualisation of a 3D FFT is more complicated, as a way needs to be found to look into the data set. One such way is to use the marching cubes algorithm [169] to visualise hyper-surfaces found in the 3D FFTs of the Ising model. The marching cubes algorithm constructs a surface so that it intersects the system coordinates where the data values match or exceed a given ISO value. For the spectral analysis, the data values correspond to the magnitudes present in

the frequency domain. This enables the user to "peel away" layers of increasing magnitude. NVIDIA provides a CUDA implementation of the marching cubes algorithm, which is used here in a slightly modified form.

The more gradual decrease in magnitude for the rewired 3D Ising model compared to the regular lattice model, as discussed before and shown in the data sets given in Figure 7.16, can also be seen when comparing the respective hyper-surfaces illustrated in Figures 7.19 and 7.20. Each surface is surrounded by a box with a value specifying the edge length to give an indication of the scale. Lower values stand for a higher magnification.

The hyper-surfaces for Ising systems rewired only in the x-dimension or in the x- and y-dimensions are given in Figures 7.21 and 7.22 respectively. The shape of the hyper-surfaces for these systems explains the large error in the respective measurements of the spherical mean. Limiting the rewiring to the x-dimension reduces the distance between cells in this dimension only, allowing domains of like-spins to grow more quickly. This produces a disk-like shape for large ISO values (the center value is the constant of the FFT and has been set to 0 as it is of no interest here), and a growing hole in the middle of the disk for even larger ISO values. The explanation for the disk-like shape is that the magnitude increases less steeply for the lower frequencies in the x-dimension as it does in the other dimensions. The hole increases in size when the magnitude in the y- and z-dimensions is larger than at any frequency in the x-dimension. This is the same effect visible in Figure 7.16 when the regular lattice Ising system is compared to a system that is rewired in all dimensions. Likewise, when the x- and y-dimensions but not the z-dimension are rewired, then the hyper-surfaces at high ISO values represent a pipe with a piece missing in the middle.

7.5. Ising Model Simulation

With the Metropolis and Wolff update algorithms working correctly, it is time to determine which one of them is the better choice for a long running simulation. To determine the critical temperatures for small p, the simulation must repeatedly perform millions of system updates, with different starting conditions and equilibrium temperatures each time, using large system sizes of up to 512^3. The performance of the chosen algorithm on the given hardware is therefore very important.

The simulation can be split into two phases. The equilibration phase, where an initially random "hot" system is quenched to its equilibrium state, needs to be performed before any measurements are taken that can be used to draw conclusions of the system at the given temperature. The equilibration phase is followed by the actual simulation phase, where independent system configurations are analysed. The key word here is *independent*, because a single system update using either the Metropolis or Wolff cluster update algorithms is not enough to decorrelate the system configuration. It is therefore necessary to determine how many simulation steps need to be performed before the system configuration is sufficiently decorrelated and a new measurement of the system state can be performed.

As discussed above, the Metropolis algorithm slows down near the critical temperature, because the size of the correlated regions is large and the update dynamics only have local effects. Cluster update algorithms, like the one from Wolff, are not affected by this critical slow down, as they update entire clusters of spins in one go. However, each individual update step using the Wolff algorithm takes longer than a Metropolis update of the entire system. It is therefore important to investigate which algorithm can equilibrate the system and then obtain sufficiently decorrelated measurements in the least amount of time. An interesting option is also to use a combination of fast Metropolis update steps and slow, but more effective, Wolff cluster updates.

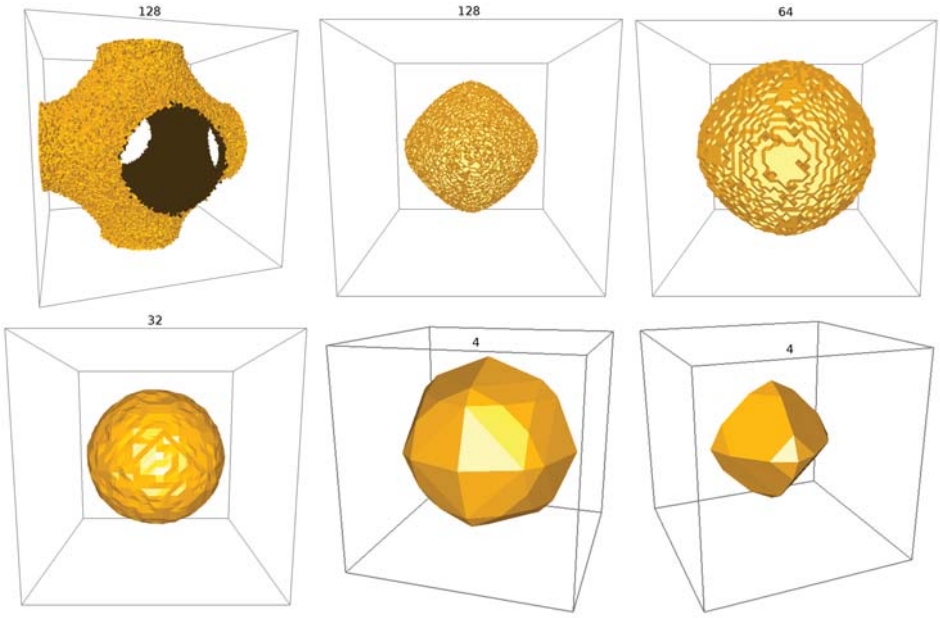


Figure 7.19.: 3D FFT of the Ising model with a regular lattice after 256 simulation steps. $T = 4.5115$, ISO values (top left to bottom right) are $\{0.02, 0.03, 0.04, 0.10, 0.70, 0.90\}$.

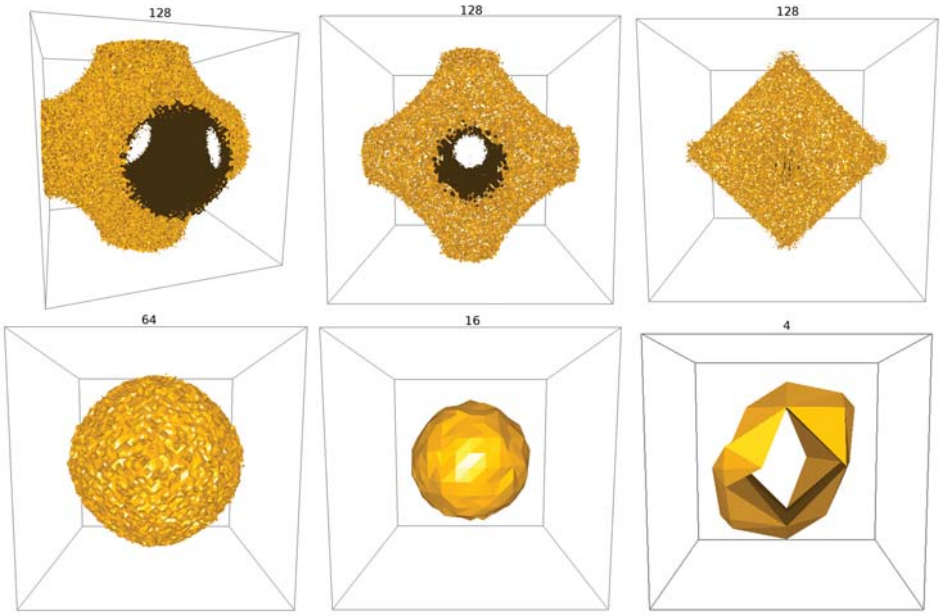


Figure 7.20.: 3D FFT of the Ising model with rewiring in all dimensions after 256 simulation steps. $p = 0.01$, $T = 4.55$, ISO values (top left to bottom right) are $\{0.10, 0.11, 0.12, 0.19, 0.70, 0.96\}$.

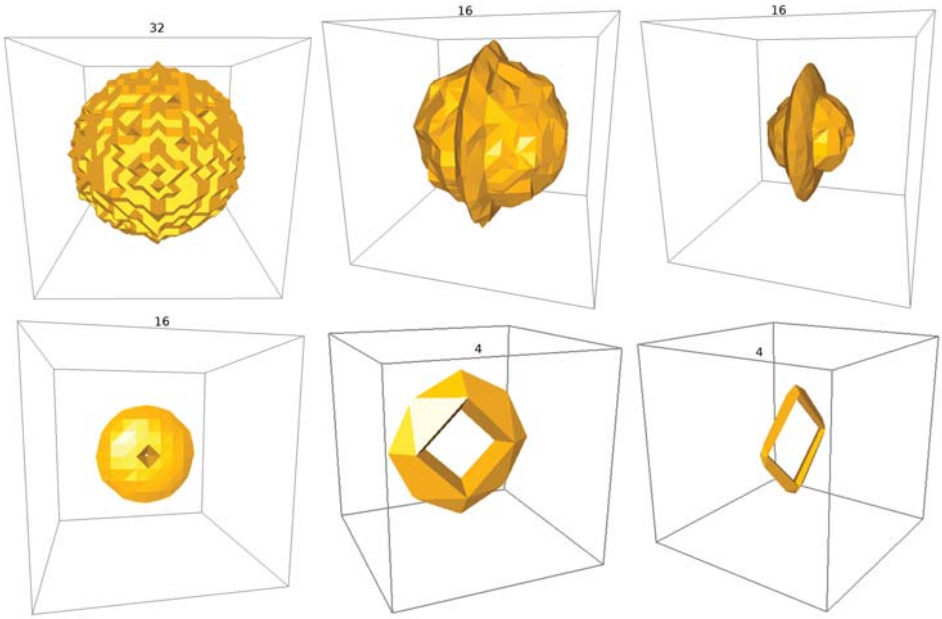


Figure 7.21.: 3D FFT of the Ising model with rewiring in the x-dimension after 256 simulation steps. $p = 0.01$, $T = 4.55$, ISO values (top left to bottom right) are $\{0.03, 0.06, 0.08, 0.11, 0.70, 0.90\}$.

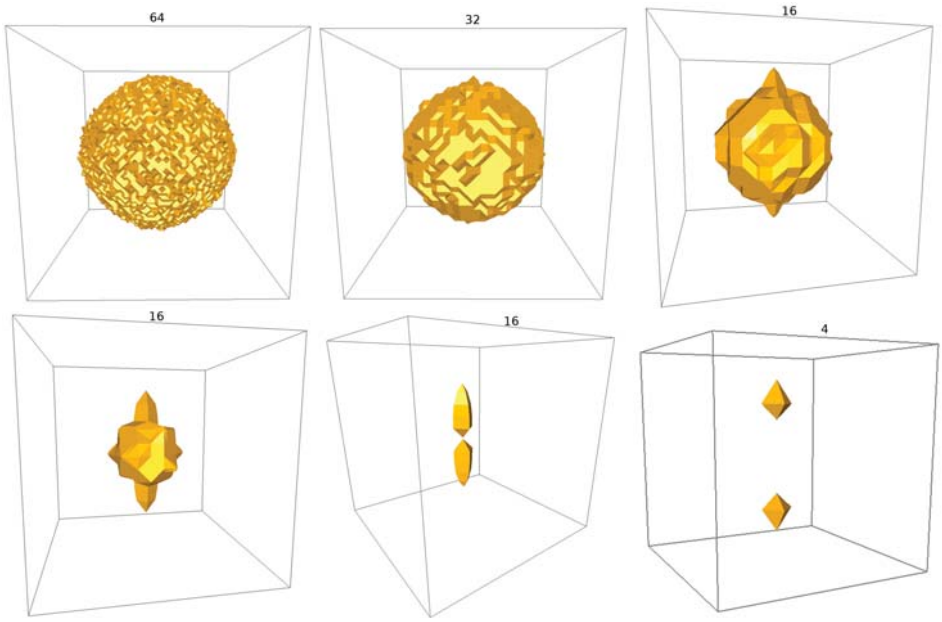


Figure 7.22.: 3D FFT of the Ising model with rewiring in the x- and y-dimensions after 256 simulation steps. $p = 0.01$, $T = 4.55$, ISO values (top left to bottom right) are $\{0.01, 0.02, 0.04, 0.05, 0.06, 0.70\}$.

7.5.1. Equilibration Phase

The following ratios of **M**etropolis and **W**olff update steps (**M:W**) are compared: 1:0, 16:1, 8:1, 4:1, 2:1 and 1:1. Ratios with more Wolff updates than Metropolis updates are not included as the results for the specified ratios, given in Figure 7.23, clearly indicate that a higher ratio of Wolff updates would not reduce the wall clock time to equilibration. The author has also shown this in [170], where ratios with more Wolff updates are tested.

The results presented in this section are for a 3D system with $N = 384^3$ spins, rewiring probability $p = 10^{-4}$ and an equilibration temperature of $T = 4.514$, which is an approximation of the so far unknown critical temperature T_c. The results are representative for all other Ising simulations performed to determine the critical temperatures of rewiring probabilities ranging from 10^{-2} to 10^{-7}. While there are small differences between the results, affected by the choice of N, p and the guess of $T \approx T_c(p)$, they do not change the conclusions drawn here. The only noticeable difference is the number of simulation steps needed to be confident that the system has reached its equilibrium state. But because this depends on factors like the choice of T, the results can only be seen as the lower-bound and a safety margin is added to it for the actual simulations.

The results given in Figure 7.23 illustrate how the system energy develops during the equilibration phase, both in terms of the number of simulation steps and the wall clock time until equilibration. The two plots in the top row show how the energy quickly drops within the first few simulation steps. The Wolff algorithm is not very effective during this phase, as the clusters in the initially random system are too small to significantly affect the measurements. This is reflected by the step function visible for all ratios that include Wolff updates. And not only do the Wolff updates have a small effect on the energy of the system, they also take significantly longer than the Metropolis updates, thus showing up as a step function when measured relative to time too.

However, the results given in the two graphs in the bottom row indicate that the Wolff algorithm should not necessarily be dismissed entirely during the equilibration phase. They show a magnified view of the system energy as it settles down and begins to fluctuate around the average equilibrium value. And here the results indicate that the ratios that include a Wolff update step every now and then actually settle down to a slightly lower energy. While the difference is within the margin of error and therefore not significant, it is interesting to observe this behaviour for most of the measurements. None of the measurements show the opposite behaviour, but for the results with $p \leq 10^{-6}$ the data sets showing the energy development as a function of the number of simulation steps are too close together to differentiate them from each other. On the other hand, all results agree that pure Metropolis updates are the quickest way to reach the equilibrium temperature when using the CUDA implementations of the two algorithms. This is therefore the default choice for all further simulations. A combination of different update ratios, where the first $1/3$ or $1/2$ of the equilibration phase uses only Metropolis updates to quickly drop the energy and the remaining steps use an 8:1 ratio, has been used experimentally for the simulations with $p = 10^{-4}$ and $p = 10^{-3}$ without any noticeable effect.

The system appears to have reached its equilibrium energy after about 2000 simulation steps. With another 1000 steps added to be on the safe side, the histograms of the fluctuations around the equilibrium energy, shown for three of the ratios in Figure 7.24, have an approximately Gaussian shape. This is the expected distribution around the average energy in the equilibrium state.

7.5.2. Decorrelated Measurements

After the system has been equilibrated so that its properties are representative of the quench temperature, it is important to determine how many simulation steps have to be performed to obtain independent system configurations and the associated lack of bias in the measurement statistics. For this it is useful to consider

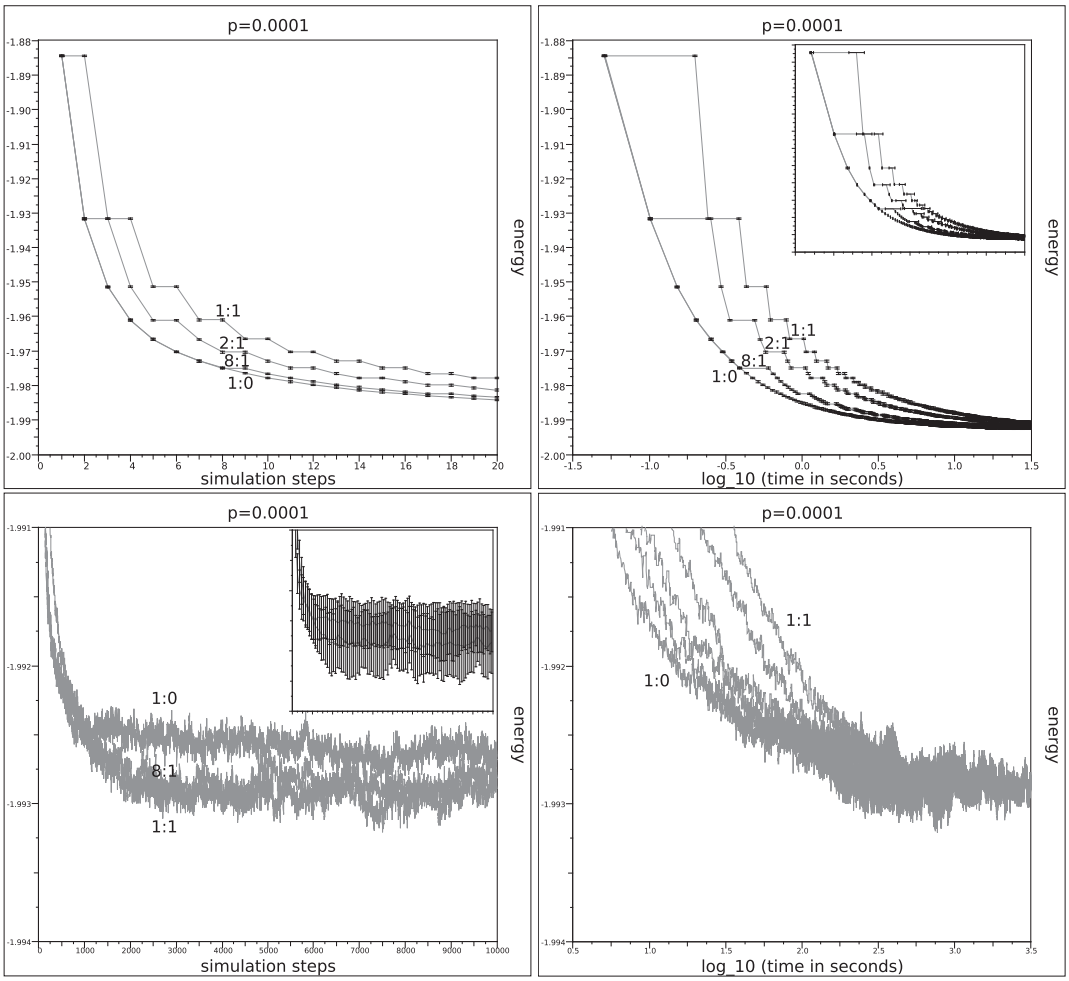


Figure 7.23.: The equilibration phase of the 3D Ising model with $N = 384^3$ sites, rewiring probability $p = 10^{-4}$ and temperature $T_c \approx T = 4.514$. Different ratios of **M**etropolis and **W**olff update steps (**M:W**) are compared (1:0, 16:1, 8:1, 4:1, 2:1 and 1:1). Not all ratios are visible in all plots, but the hidden data sets fall into the area between the extreme cases (1:0 and 1:1) and do not show any unexpected behaviour. The graphs show how many simulation steps (left) and how much time (right) it takes for the different combinations of the update algorithms to get the system into its equilibrium state. The plots in the top row show the full energy range of the first 20 steps and approximately 30 seconds of the simulation respectively. The plots in the bottom row offer a magnified view of the transition into the equilibrium state. Each data point is averaged over 30 simulation runs. Error bars are displayed for the deviation from the measured energy at each data point of the first two plots and additionally for the deviation from the measured time in the inset of the top right plot. No error bars are displayed in the last two plots, but the inset of the bottom left plot shows the standard deviations for ratios 1:0 and 1:1, averaged over 100 consecutive x-values, to give an idea of their extend at the given level of magnification.

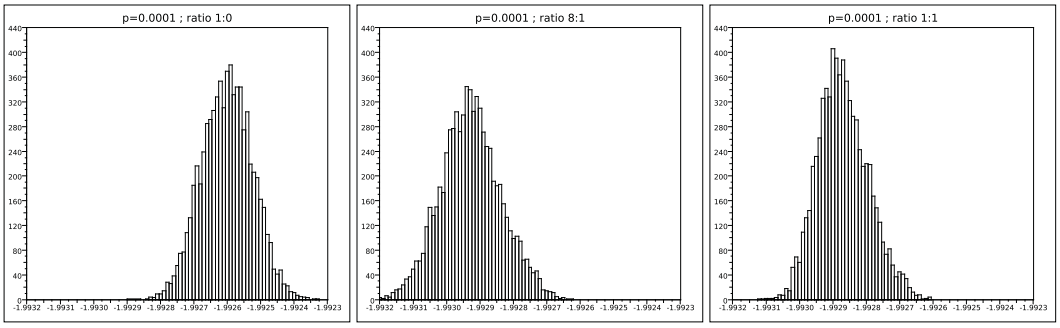


Figure 7.24.: The histograms show the energy distribution for simulation steps 3000 to 10000 of the three Metropolis:Wolff update step ratios (from the left: 1:0, 8:1 and 1:1) given in the first plot in Figure 7.23. The approximately normal distribution of the values indicates that the system is equilibrated.

the Pearson product-moment correlation coefficient ρ [171, 172]. This population correlation coefficient is defined as:

$$\rho_{X,Y} = \frac{E[(X-\mu_X)(Y-\mu_Y)]}{\sigma_X \sigma_Y}, \tag{7.4}$$

where E is the expected value operator, μ_X and μ_Y are the expected values and σ_X and σ_Y the respective standard deviations. The coefficient ρ is $+1$ in the case of perfect correlation, -1 in the case of perfect anticorrelation and some value in between for partial (anti)correlation. If the populations are independent, then $\rho = 0$.

Two subsequent system configurations can be highly anticorrelated (close to -1) when the Wolff cluster update algorithm is used at low temperatures, as it might simply flip a large part of the system from one state to the other. This is just as undesirable as highly correlated configurations. Therefore, the results presented here use the absolute value $|\rho|$ of the coefficient, treating both correlation and anticorrelation as equal.

The correlation should ideally be zero between configurations used for measurements. For practical purposes, a coefficient of $|\rho| \leq 10^{-2}$ is deemed to be sufficiently decorrelated for the results presented here. Figure 7.25 shows how the correlation changes with both the algorithmic steps and wall clock time for different ratios of Metropolis and Wolff. This assists in determining the ideal hybrid algorithmic ratio to maximise the number of independent measured configurations computed per second. The system parameters for the results presented in the first two plots are the same as those used for the equilibration phase, that is $N = 384^3$ spins, rewiring probability $p = 10^{-4}$ and temperature $T = 4.514$. The results in the bottom row are for $N = 352^3$, $p = 10^{-2}$ and $T = 4.592$.

The outcomes for $p = 10^{-4}$ are representative for the Ising model simulations performed to determine the critical temperatures of rewiring probabilities ranging from 10^{-3} to 10^{-7}. The only significant difference is the number of simulation steps needed to get the correlation coefficient to fall below 10^{-2}. However, this also depends on the choice of the temperature value. The results can therefore only be seen as a lower-bound to get good statistical measurements for the given system parameters. The graphs show that the correlation between system configurations decreases more quickly, in terms of the number of simulation steps, the more Wolff cluster updates are performed. This is in agreement with the expected critical slowing down of the Metropolis algorithm.

While the number of simulation steps needed to sufficiently decorrelate the system configuration decreases as the percentage of rewired edges rises, the results for $p = 10^{-2}$ are the first to demonstrate that the correlation values eventually settle around an equilibrium value. This is to be expected for all other cases too, it merely takes more update steps to get there. The main difference between these results to those for

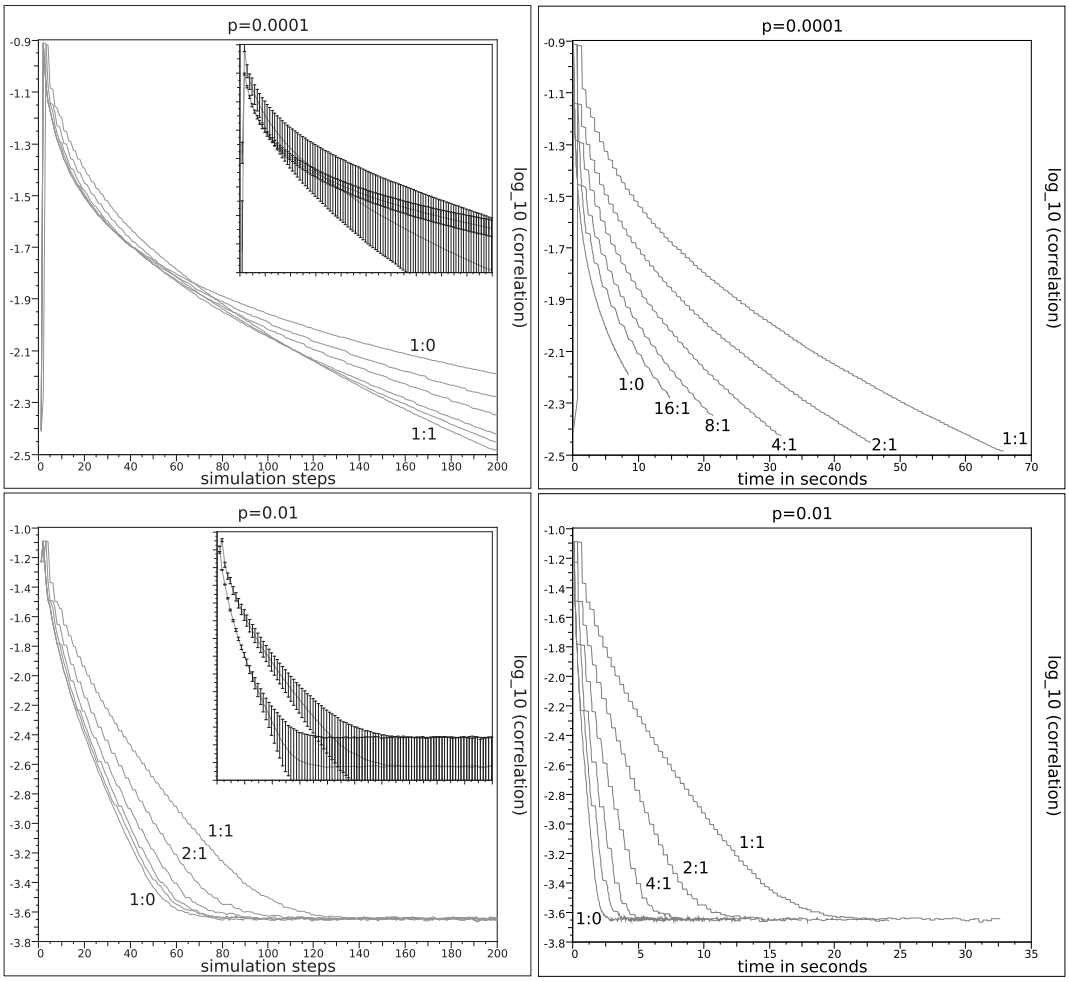


Figure 7.25.: The graphs show how the correlation between system configurations of the 3D Ising model develops over 200 simulation steps. The plots on the left shows the results related to the simulation steps, whereas the plots on the right shows the amount of time it takes to perform those updates. System size $N = 384^3$ for rewiring probability $p = 10^{-4}$ (top) and $N = 352^3$ for $p = 10^{-2}$ (bottom). Different ratios of Metropolis and Wolff update steps are compared. The system is equilibrated before correlation measurements are performed. Each data point is averaged over 3000 measurements from 30 independent simulation runs. During each simulation run, 100 measurements are taken, starting at intervals of 200 simulation steps (the reference configuration) to which the subsequent 200 simulation steps are compared. Error bars are not displayed in the main plots, but the insets on the left show the standard deviations for ratios 1:0 and 1:1, averaged over 2 consecutive x-values, to give an idea of their extend. The results for all other measured ratios (16:1, 8:1, 4:1, 2:1) lie in between those displayed.

smaller values of p is that here the pure Metropolis updates decorrelate the system more quickly than the hybrid Metropolis-Wolff updates, not only in terms of the wall clock time, but also in terms of the number of simulation steps. It appears that the Metropolis algorithm benefits more from the perturbation of the lattice than Wolff's algorithm. The long distance shortcuts created during the rewiring procedure effectively decrease the size of correlated regions, thus reducing the critical slowing down effect. While individual spins still only interact with their direct neighbours, these neighbouring cells can be in entirely different regions of the lattice.

Metropolis updates are significantly faster than Wolff cluster updates. In fact, they are so much faster

that for all tested values of p, it takes less wall clock time to reach a certain level of decorrelation when only using the Metropolis algorithm, even when a larger number of simulation steps needs to be performed. The 1:0 ratio using only Metropolis updates thus maximises the number of independent measured configurations computed per second and is used for all further simulations.

7.6. Statistics Computation with CUDA

A number of system properties and statistical metrics are used to analyse the Ising model in the sections above. These include the energy, magnetisation, component size distribution and correlation between system configurations. Since the simulation is running on the graphics accelerator and the data needed to compute these metrics is stored in device memory, it makes sense to also use CUDA to perform most of the analysis.

The current system energy is computed from the sum of the like neighbour bonds, whereas the magnetisation is the difference of the number of spins with value $+1$ minus the number of spins with value -1. The data for both metrics is collected by the same CUDA kernel. The implementation is similar to that of the Metropolis update kernels, where two different kernels are used to process spins that are affected by rewiring and those that are not. However, there is no need for the checkerboard update pattern. All cells can be processed in parallel, as the spin values are not modified. The kernel used to process the cells affected by rewiring is executed by $|V_m|$ threads, the number of such cells, each thread collecting the data for a single spin. The kernel for unmodified cells is again executed by T threads, where T depends on the number of processing cores on the device, and each thread collects and aggregates the data for multiple spins. After the data is collected, the threads of both kernels write their local results to shared memory and synchronise with the other threads in the same block. Then two parallel `plus_reduce` operations are used to compute the respective sums of the data collected by the thread block in shared memory. Finally, the first thread in each block performs two `atomicAdd` operations to add these partial sums to the values for the entire system, which are stored in global memory. When the kernels return, these values are copied from device memory to host memory, where they are used to compute the average energy and magnetisation for the current system configuration.

To compute the correlation ρ between two system configurations X and Y, the current average spin value μ_X needs to be determined. If the magnetisation is measured at the same simulation step, then this value is already known. Otherwise, a CUDA kernel computes it as described above. Then the sum of the squared deviations from μ_X is computed on the GPU to get the variance and with that the standard deviation σ_X for the current population X. The third CUDA kernel uses the average spin value for the current configuration μ_X and the value for the configuration that it is compared to μ_Y to compute $E[(X - \mu_X)(Y - \mu_Y)]$. Immediately after a spin value from configuration Y has been used, the kernel overwrites it with the current value from configuration X so that X can be used as the "old" configuration for the next correlation measurement, assuming that the correlation is computed in regular intervals. With all variables known, ρ can be computed as defined in Equation 7.4.

The cluster size distribution presented in Figure 7.17 was computed using a slightly modified version of the component labelling algorithm described in Section 6.1. It uses the same breadth-first search strategy, but with the added restriction that two vertices connected by an edge are only considered as part of the same cluster if their spins have the same value. The implementation has also been adapted to work on the data structure used for the Ising simulation, including the implicit neighbour calculation for cells not affected by rewiring. Again, two different kernels are used for spins with implicit and explicit neighbours. Both kernels can be executed concurrently if the hardware supports this feature.

Table 7.4.: The simulation settings used for the Binder cumulant measurements for different rewiring probabilities p. The largest system size used for each p is $N = L^3$, where L is the dimension length of the cubic system. Additional Binder cumulant curves are computed for $L-32$ and $L-64$.

		Number of simulation steps to:	
p	L	equilibrate	decorrelate
10^{-2}	352	4000	60
10^{-3}	352	4000	80
10^{-4}	384	3000	130
10^{-5}	384	2000	180
10^{-6}	384	4000	190
10^{-7}	512	4000	200

7.7. Critical Temperatures

The critical temperature of the regular lattice 2D Ising model is known exactly from the analytical expression $T_c = 2J/ln(1+\sqrt{2}) \approx 2.269$ [134]. As no analytical solution is known for the 3D Ising model, Monte Carlo simulations are typically used together with one of a number of approximation methods [145, 161] to determine the critical temperature numerically. In this thesis Binder's fourth-order magnetisation cumulant U [145] is used to determine $T_c(p)$ for the small-world 3D Ising model with p as small as 10^{-7}.

The Binder cumulant is a well established method widely used in the literature to compute the critical temperatures of the Ising model [141, 142, 149, 173]. Ferrenberg and Landau [149] used it to compute two slightly different approximations of the inverse critical temperature $K_c = 0.2216595 \pm 0.0000026$ and $K_c = 0.2216576 \pm 0.0000022$ for the regular lattice 3D Ising model. Results from Monte Carlo renormalisation group calculations estimate T_c to be slightly higher, for example Pawley et. al. [146] obtained $K_c = 0.221654 \pm 0.000006$ and Baillie et. al. [147] obtained $K_c = 0.221652 \pm 0.000003 \pm 0.000001$.

The Binder cumulant for a system of size $N = L^d$, with lattice dimension length L, is defined as [145]:

$$U_N = 1 - \frac{\langle M^4 \rangle_N}{3 \langle M^2 \rangle_N^2}, \tag{7.5}$$

where M is the magnetisation as defined in Equation 7.2. The Binder cumulant produces an S-shaped curve when computed over a range of temperatures. The curves for different system sizes N intersect at the critical temperature and thus provide a way to extrapolate to the thermodynamic limit using relatively small system sizes. The Binder cumulant needs to be computed for at least two different system sizes. A third data set is useful, however, as it makes it possible to calculate an error value for the results. Table 7.4 shows the largest dimension length L used to compute the Binder cumulant for each rewiring probability p, alongside other settings used for the individual simulations. The two smaller dimension lengths used to compute the intersecting curves are $L-32$ and $L-64$. The other settings remain unchanged. All three system sizes need to be large enough to ensure that a reasonable number of shortcuts are created for the given probability p. Smaller values of p therefore require larger system sizes.

The expected value of the system magnetisation, needed to compute U_N, is obtained by taking the mean value of $10,000$ measurements. As discussed before, the system configuration used for each measurement needs to be sufficiently independent from the configuration used for the previous measurement. The table lists the number of simulation steps performed to decorrelate the system state between each measurement. In the case of the Binder cumulant simulations for $p = 10^{-7}$, this means that a total of 2×10^6 Metropolis update steps are performed after the system is equilibrated. Each simulation step updates the entire system of $N = 512^3$, $N = 480^3$ or $N = 448^3$ spins for the three different system sizes used for this rewiring

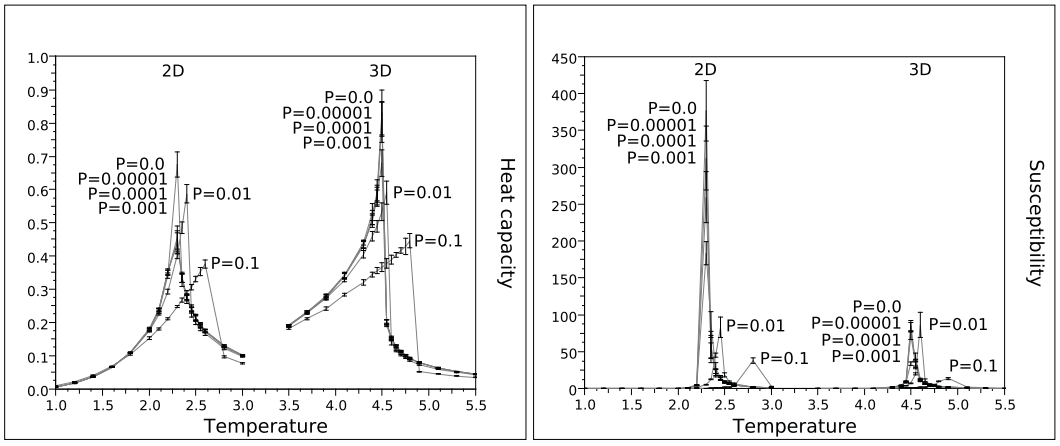


Figure 7.26.: The heat capacity (left) and magnetic susceptibility (right) can be used as a quick visual indicator of the location and shift of the critical temperature with rewiring probability p. The system size is $N = 1024^2$ and $N = 128^3$ for the 2D and 3D simulations respectively. The results show the mean and standard deviation for 10 simulation runs.

probability.

The table also shows that systems with smaller values of p tend to take longer to reduce the correlation between system configurations by the same amount, even where the system size remains the same. As discussed in Section 7.5.2, this observation can be explained with the lessening of the critical slow down effect caused by an increasing number of shortcuts. The number of simulation steps performed between measurements does not exactly reflect the point where ρ becomes smaller than 10^{-2}. It is always rounded up to the next multiple of ten and in some cases a slightly larger number is used to be on the safe side. It is important to remember that the measurements are affected by the value of T used to obtain them, which is merely the best approximation of T_c available at this stage. The values from the equilibration and decorrelation measurements should therefore be seen as a lower bound rather than the optimal settings.

The remaining question to answer is how this initial estimate for the critical temperature of a particular rewiring probability is obtained. One possible way is shown in Figure 7.17 on page 131. The bimodal behaviour of the cluster size distributions – where the two peaks move closer together with decreasing distance from T_c – can be used to get a relatively rough idea of the position of the critical temperature. Another way is to compute the heat capacity and magnetic susceptibility of the system.

7.7.1. Heat Capacity & Magnetic Susceptibility

The heat capacity and magnetic susceptibility can be used as a quick visual indicator of the location and shift of the critical temperature with rewiring probability p. This is illustrated in Figure 7.26. The sharpness of the phase transition is limited by the system size and its accuracy is limited by the granularity of the quench temperature increments used for the measurements.

The heat capacity can be computed using the fluctuation dissipation theorem from the energy variance, which is defined as [139]:

$$C_v = \frac{\beta}{T}\left[\langle E^2\rangle - \langle E\rangle^2\right] \tag{7.6}$$

where $\beta = J/k_BT$ is the reciprocal temperature in units of J. Likewise, the magnetic susceptibility is the spin fluctuation around the mean value and can be calculated in terms of the variance relationship [139]:

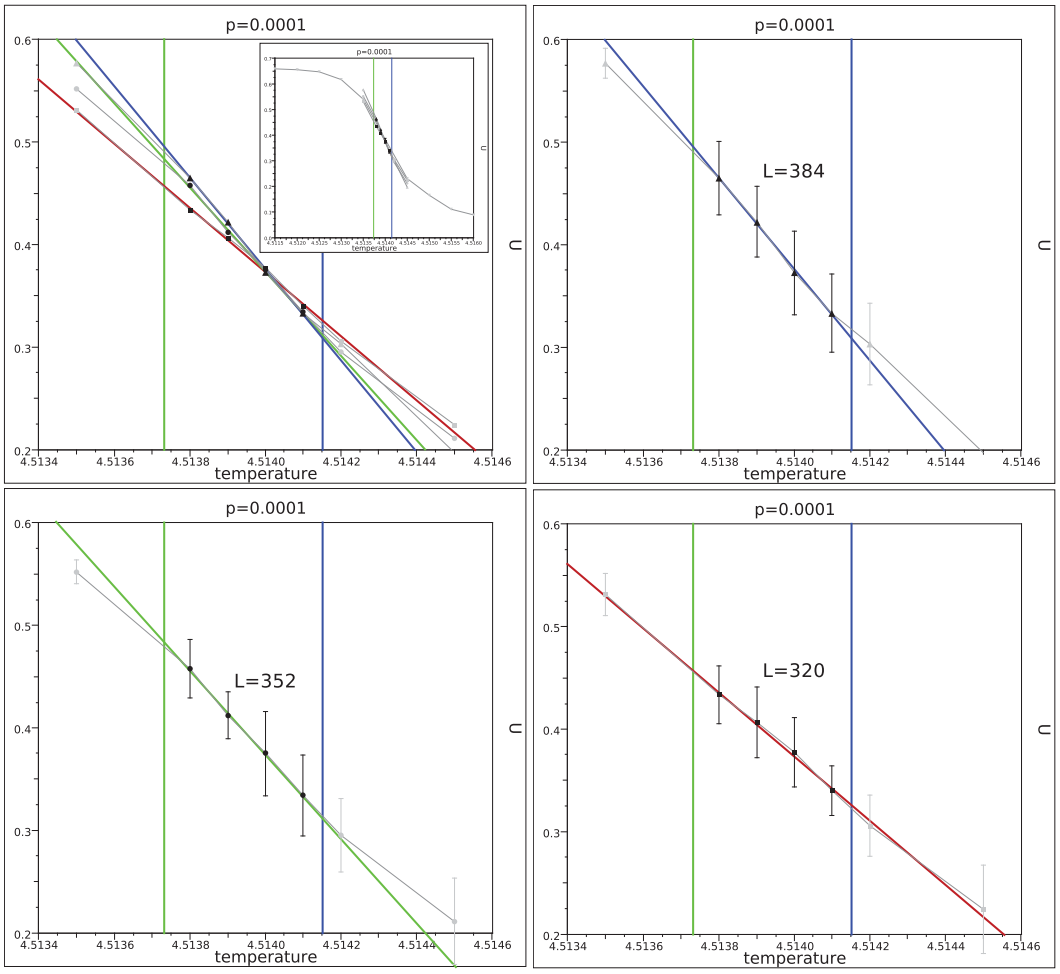


Figure 7.27.: Intersection at $T_c(p = 10^{-4}) = 4.51404 \pm 0.00004$.

$$\chi = \beta \left[\langle M^2 \rangle - \langle M \rangle^2 \right], \tag{7.7}$$

While these measures offer a good way to get a quick visual estimate of the critical temperature, they are not accurate enough to begin investing significant computational resources. Nevertheless, the results can be used to narrow the temperature region down and then do a scan with the Binder cumulant method, using the smallest of the three system sizes used for the simulations of a particular rewiring probability, to get a good estimate of the critical temperature region. This estimate is then used to obtain the equilibration and decorrelation settings given in Table 7.4 that are used for the fine-grained Binder cumulant measurements.

7.7.2. Binder Cumulant Results

The simulations are performed for a number of temperatures in a narrow region around the critical temperature estimate for each value of p. $10,000$ decorrelated system states are used to compute the magnetisation values needed to calculate the Binder cumulant result for a single simulation run. Between 20 and 80 simulation runs are performed for each data point actively used for the Binder cumulant calculation (i.e. data points within the filtered regions). Generally, additional runs are performed until further simulations do not significantly change the mean and error values anymore.

Table 7.5.: The critical temperatures $T_c(p)$ for rewiring probabilities $p = 10^{-2}$ to $p = 10^{-7}$.

p	$T_c(p)$	Error
10^{-7}	4.51175	$\pm$0.00041
10^{-6}	4.51159	$\pm$0.00004
10^{-5}	4.51186	$\pm$0.00008
10^{-4}	4.51404	$\pm$0.00004
10^{-3}	4.52674	$\pm$0.00004
10^{-2}	4.593754	$\pm$0.000007

Figure 7.27 illustrates the Binder cumulant results for $p = 10^{-4}$. The detailed results for all other values of p are given in Appendix C (see page 195). Table 7.5 lists the critical temperatures $T_c(p)$ obtained from the intersections of the Binder cumulant measurements. The results for $p = 10^{-2}$ should be taken with caution, because the data sets for $L = 320$ and $L = 288$ actually intersect twice. The author has not been able to improve these results, even though between 50 and 80 simulation runs have been performed for all data points in the region within the filter. Simulations with an additional system size are needed to assert that the chosen intersection is indeed the correct one. This is left for future work.

Only between 28 and 40 simulation runs have been performed for the results for $p = 10^{-7}$ so far. The least square linear fits to the data sets for $L = 480$ and $L = 512$ do not intersect within the filtered region and especially the results for $L = 512$ clearly require some more data to reduce the error.

Figure 7.28 illustrates the shift in the critical temperature $\Delta T_c = T_c(p) - T_c(p = 0)$ for three different estimates of $T_c(p = 0)$. The estimate for $T_c(p = 10^{-7})$ is not included in the calculation of the scaling, as it obviously is not accurate enough. The results agree with the hypothesis that the perturbation in the critical temperature goes as a power-law in p such that p^s. The best estimate for the exponent $s = 0.735 \pm 0.003$ is obtained from the slopes to all three data sets, which are given in the caption of Figure 7.28. There is still a significant discrepancy to the expected scaling [142] $\Delta T_c \sim p^{0.53}$ from Equation 7.3 on page 104, which may suggest that the values of p are still too large and the system sizes too small.

7.8. Discussion

This chapter describes how all steps of a scientific simulation can benefit from the use of GPUs as compute accelerators. It begins with the most central part, the simulation model and the update algorithms used to evolve the simulation. But the following steps, namely the validation of the model through visual and numerical analysis as well as the statistical methods used to extract certain properties from the model, are just as critical to the performance and eventual success of the experiment.

Several of the other chapters in this thesis discuss algorithms for complex graphs with data structures that are hard to predict and therefore difficult to process efficiently on the data parallel architecture of the GPU. While it is in many of these cases still possible to achieve performance improvements compared to the CPU, they tend to be rather moderate. The graph structure used for the rewired Ising model is not entirely regular either. But similar to the Watts-Strogatz β-model, it retains large parts of the underlying regular d-dimensional lattice structure as long as the rewiring probability is small. And with the knowledge that the rewiring probabilities of interest are very small, the algorithms have been optimised for this situation. This leads to significant performance improvements of almost 100 times for 2D simulations and up to about 45 times for 3D simulations when compared to a sequential CPU algorithm running on a high-end processor (Section 7.1.5).

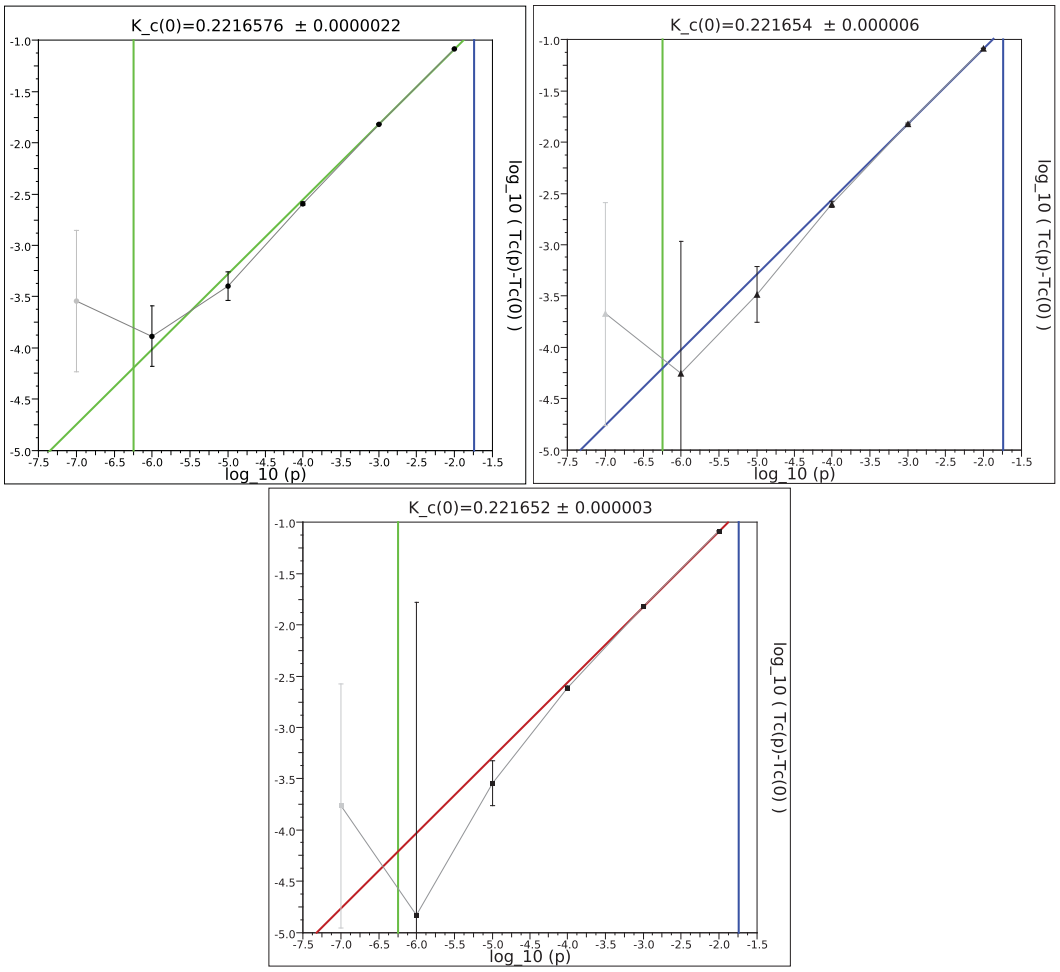


Figure 7.28.: The shift in the critical temperature ΔT_c with respect to the rewiring probability p when using the inverse critical temperature estimate K_c from Ferrenberg et. al. [149] (top left), from Pawley et. al. [146] (top right), or from Baillie et. al. [147] (bottom). The respective slopes of the least square linear fits to the data sets are 0.734 ± 0.002, 0.736 ± 0.004, and 0.737 ± 0.003.

From the two update algorithms discussed in Sections 7.1 and 7.2, the Metropolis update algorithm is much better suited to be executed on a data parallel architecture than the Wolff cluster update algorithm. With the Wolff algorithm, some of the work is performed in vain, like labelling all connections between vertices as either traversable or non-traversable based on some probability and the spin values of the neighbouring vertices, even though only a small fraction of these links is actually considered in the second step. However, the Metropolis algorithm suffers from its local update dynamics, which cause the critical slowing down effect, thus requiring a large number of update steps to decorrelate the system configuration.

Sections 7.5.1 and 7.5.2 investigate whether the "brute force" approach with a larger number of fast Metropolis updates or some combination with the slower Wolff cluster updates works best on the GPU. It turns out that Wolff cluster updates are not a good choice in the beginning of the equilibration phase, as the clusters are much too small. But even further into the simulation the raw speed of the Metropolis algorithm leads to faster overall execution times. The small-world long distance links found in the structure of the rewired Ising model help to decorrelate the system and by doing so reduce the critical slowing down, thus somewhat mitigating the major problem of the Metropolis algorithm.

The ability to visually monitor, analyse and interactively manipulate the behaviour of the system can be an

invaluable tool both to verify that the system works correctly and to improve the understanding of a complex system. But to observe the same emergent behaviour illustrated by numerical results, it is often necessary to be able to run the visual simulation using similar system sizes as those used for the numerical studies. This can be a computational challenge that takes powerful compute accelerators like the GPU to be mastered. With increasingly powerful and numerous processing cores, data movement is more and more often the limiting factor. Being able to keep the data on one device instead of having to send it over a relatively slow device interconnect, like the PCI express bus, can therefore significantly improve the performance. The visualisation of the simulation thus benefits from the fact that the data that is to be rendered already resides in device memory. The CUDA $\Leftrightarrow$ OpenGL interoperability is a powerful feature for all applications that can not only efficiently utilise the data parallel architecture of GPUs for their compute requirements, but also make use of the GPU in its more traditional function as a rendering device for computer graphics.

Statistical analysis of the system often condenses a large amount of data down to a smaller amount of data, in some cases only a single number. It therefore makes sense to analyse the data on the same device that is used to run the simulation. Then only the condensed data that is of actual interest to the scientist needs to be transferred back to the host system. The process of collecting the data that is required to compute the magnetisation and energy of the current Ising configuration, as discussed in Section 7.6, is an example for a case were a large amount of data can be condensed down to a single number for each of the metrics before it needs to be transferred over the PCI bus. Sometimes the architecture of the accelerator does not lend itself particularly well to condensing the data down to a single value, but it is possible to reduce it by several orders of magnitude, thus reducing the data that needs to be copied back to the host as well as the remaining amount of work.

Chapter 8

A Neural Network Model

This chapter describes the implementation of a neural network model that can be used to study the effects of general anaesthesia on the cerebral cortex. Such effects may include phase transitions in the electroencephalogram (EEG) spectral power [174–176], distinct spatial or temporal patterns of neural firing rates, "unconsciousness" when the anaesthetic effects produce functional disconnection of major regions in the network model [177] and anaesthetic hysteresis as the model transitions between states [178]. The aim of this chapter is not to analyse the model, but to lay the computational foundation for further studies by means of high performance, parallel GPU algorithms that can simulate networks with millions of individual cells and hundreds of millions of connections in near real-time on affordable commodity hardware. Such large-scale simulations are important if we are to improve our understanding of the way real brain networks work, because some macroscopic behavioural patterns only emerge for large systems with many interactions on the microscopic level or when the system is analysed over many length-scales of one of its properties.

A number of different models for cortical spiking neurons exist, ranging from simple but computationally cheap integrate and fire models to biophysically meaningful but computationally expensive Hodgkin-Huxley [179] type models. The computational demands of Hodgkin-Huxley type models limit simulations to relatively small neural networks, even when using multiple GPUs as parallel accelerators for the simulation [180]. In [181], Izhikevich gives a summary of the differences between several common models, both in terms of their complexity and their ability to reproduce the spiking and bursting behaviour found in cortical cells. He shows that the Izhikevich model of spiking neurons [182] is computationally much cheaper than the Hodgkin-Huxley model, but nevertheless capable of reproducing the firing patterns exhibited by real cortical neurons [181]. And just like the Ising model has not been studied so thoroughly by many scientists over the last decades because of its physical realism, but rather because it is one of the simplest models of interacting particles with some features of a physical system, the choice of the Izhikevich model is based on its ability to produce biologically plausible behaviour while being relatively fast to compute. It should be noted, however, that many of the algorithmic approaches proposed in this chapter are not limited to this particular model and can easily be transferred to other models of spiking neurons.

Nageswaran et. al. [183] have also proposed a CUDA implementation for a spiking neural network model using Izhikevich type neurons. However, they use very different algorithmic approaches and data structures to those discussed here and their model generally has different design goals. While it supports a more flexible definition of conduction delays between neurons and can adjust the strength of individual neural connections, it does not support the exponential washout of input currents over a period of time that is discussed in the following section. The ability to modify these washout values and the time frame of their effect is essential to the simulation of drug effects and therefore a fundamental concept of the model discussed here.

8.1. The Model

The model uses Izhikevich [182] type neurons, which can reproduce the spiking and bursting behaviour observed in cortical neurons. The behaviour of individual neurons is controlled by four parameters a, b, c, d. Depending on the values of these parameters, the model can simulate regular spiking, intrinsically bursting and chattering excitatory cells, as well as fast spiking and low-threshold spiking inhibitory cells. Inputs from excitatory cells decrease the distance to the spike threshold, whereas inhibitory inputs increase the distance to the threshold [184]. A cell is said to be depolarised when its potential is higher than the resting potential and it is hyperpolarised when its potential is lower than the resting potential. Appendix D illustrates the different spiking patterns produced by the Izhikevich model. See [182] for a more detailed explanation of the different types of cortical cells and [184] for an exhaustive discussion on neural excitability.

Following the pair of ordinary differential equations used for the spike initiation dynamics of the Izhikevich model [182]:

$$\begin{aligned} v' &\leftarrow 0.04v^2 + 5v + 140 - u + I \\ u' &\leftarrow a(bv - u) \end{aligned} \tag{8.1}$$

with the after-spike resetting

$$\text{if } v \geq 30mV, \text{ then } \begin{cases} v \leftarrow c \\ u \leftarrow u + d. \end{cases} \tag{8.2}$$

Variable v represents the neuron's membrane potential and u represents a membrane recovery variable. After an electrical impulse – or action potential – has been triggered, the neuron is reset according to Equation 8.2. The derivative v' of v with respect to time is chosen so that the membrane potential has millivolts (mV) scale and time has milliseconds (ms) scale [182]. The membrane potential is limited to $+30$ mV, which is the peak of a spike and should not be confused with the spike threshold. The actual threshold is dynamic and lies between -70 and -50 mV [181]. The model parameters are as follows:

- Parameter a affects the time scale of the recovery variable, such that larger values result in faster recovery and smaller values in slower recovery.
- Parameter b couples the recovery variable u to the fluctuations of the membrane potential v. Larger values represent a stronger coupling and can facilitate subthreshold oscillations [182].
- Parameters c and d specify the after-spike reset values of the membrane potential v and recovery variable u as defined in Equation 8.2.
- I is the input current from external sources, for example other parts of the brain like the thalamus.

As suggested by Izhikevich [182], excitatory neurons use $a = 0.02$, $b = 0.2$, $c = -65 + 15r^2$ and $d = 8 - 6r^2$, where r is a random variable uniformly distributed on the interval $[0, 1)$. This produces excitatory neurons of all three types, with a bias towards regular spiking cells. Inhibitory cells use $a = 0.02 + 0.08r$, $b = 0.25 - 0.05r$, $c = -65$ and $d = 2$. A 4:1 ratio of excitatory to inhibitory cells is used by default, although other ratios like 85:15 [176] are possible too.

I is calculated for every neuron as $I = I_{e,i} + I_{noise} + I_{boost}$, where $I_{e,i}$ is the base current for all excitatory (I_e) or inhibitory (I_i) cells. Different values can be set for excitatory and inhibitory neurons. $I_{noise} = n_{e,i} \times r_{norm}$ is random noise computed from a base noise value $n_{e,i}$, which is also defined independently for the two cell types, times a normally distributed random variable with mean $\mu = 0$ and standard deviation $\sigma = 1$. I_{boost} is an additional boost delivered to a fraction of all cells selected at random during each simulation step that the boost is active. Although the value for the random noise can be negative, the final input current I is

not allowed to fall below zero. While all three of these input currents can be modified between individual steps of an ongoing simulation run, the first two are generally meant to model a constantly active source of activity that drives the cortical model, whereas the boost is meant to simulate a brief external impulse or even to "jump start" the simulation if necessary.

When a cell generates a spike, then the excitatory or inhibitory action potential reaches all of its neighbours after a time delay δ that varies among the affected neurons. The delay models the time that a presynaptic action potential takes to pass through the axon (transmitting fibres) and dendrite (receiving fibres) and reach the postsynaptic neuron. The synapse is the axon-dendrite junction. The effects of an action potential on the receiving neuron – called the postsynaptic potential (PSP) – are not applied all at once, but rather diminish exponentially over a period of time. This is based on the model described in [176]. The values for the exponential washout are defined in separate washout tables $W_{e,i}$ for each type of neuron, such that:

$$W(t) = Ae^{-t/\tau}, \tag{8.3}$$

where t is time measured in simulation steps, the constant $A_{e,i}$ regulates the voltage amplitude and $\tau_{e,i}$ defines the rate of the exponential washout. The length of the washout table is implementation specific, but is generally chosen to be at least 3τ, thus allowing the action potential to diminish to less than 5% of its initial strength. The effects of action potentials are added directly to the membrane potentials v of the postsynaptic neurons. Values for A and τ are only experimental at this stage and are set to $A_e = 0.25$, $\tau_e = 10$, $A_i = -0.5$ and $\tau_i = 10$. Note the negative value used for A_i that causes the inhibiting effect on the membrane potentials of postsynaptic neurons.

The effects of anaesthetic drugs are modelled using values $\lambda_{A(e,i)}, \lambda_{\tau(e,i)}$ and $\lambda_{I(e,i)}$ that can be defined individually for excitatory and inhibitory neurons. The λ values modify the corresponding base values to get the effective excitatory values as follows:

$$\begin{aligned} A &= A_e/\lambda_{Ae} \\ \tau &= \tau_e/\lambda_{\tau e} \\ I &= I_e - \lambda_{Ie} + 1 \end{aligned} \tag{8.4}$$

and the effective inhibitory values:

$$\begin{aligned} A &= A_i\lambda_{Ai} \\ \tau &= \tau_i\lambda_{\tau i} \\ I &= I_i + \lambda_{Ii} - 1 \end{aligned} \tag{8.5}$$

Thus, values of $\lambda > 1$ simulate drugs that have a dampening effect when applied to excitatory neurons and a strengthening effect when applied to inhibitory neurons. By default no drug effects are applied, that is all $\lambda = 1$.

8.2. The Network & Data Structure

The current implementation of the neural network uses a 2D grid structure, such that every neuron can be addressed by its (x,y)-coordinates. Connections are one-way only and are generated randomly using a normal distribution centered on the neuron at the end of the arc. The radius r, which defines the maximum

distance between neighbouring neurons, is user defined but can be at most half the dimension length L of the system. The standard deviation of the normal distribution is set to $\sigma = r/3$. The fact that the neighbours are centered at the end of the arc and not at its source is an important one. The arcs are even stored in reverse direction. This is done because the simulation uses a pull model instead of a push model to simulate the transfer of action potentials from one neuron to its neighbours. Every neuron has the same number of incoming connections. The number of outgoing connections depends on how many other neurons pick a particular neuron as one of their neighbours. Figure 8.2 gives an example of the neighbour distribution for two neurons. The squares and circles represent the source of the connections to neurons $(0,0)$ and $(64,64)$ respectively.

The arcs are stored in the adjacency-lists of the receiving neurons. As every neuron has a fixed number of incoming connections, the main advantage of this implementation is immediately apparent. A major difficulty for many of the algorithms described in this thesis is that they have to iterate over adjacency-lists of variable length, which increasingly becomes a problem the more the degrees of nodes processed by sequential threads deviate from each other. Real cortical neurons have a very large number of connections, likely in the thousands. And even though the in-degree is set to 100 for the purpose of this simulation, unless specifically mentioned otherwise, warp divergence due to variable adjacency-list lengths would be significant. The use of the pull-model to query the state of incoming connections and the respective neurons at the source of the arcs together with a uniform in-degree is well suited for a data-parallel architecture. The chosen in-degree of 100 is not a fixed limit imposed by the algorithm, but seems appropriate considering the small number of neurons in the simulated systems compared to the cortex of even a small animal.

The way neighbours are chosen is not set in stone and can be changed without modifications to the simulation code as long as the data structure used to store the arcs in memory is not affected and the number of incoming connections remains the same for all neurons. For example, the author has experimented with neighbours chosen only from one side of each neuron, in a half-circle of radius r, thus introducing a directionality to the propagation of action potentials through the system. It has also been suggested in the literature [73] that a small-world graph structure may be a good candidate to more accurately model neural networks, as this structure tends to minimise the wiring costs with only a few long range connections among otherwise relatively localised short range links. A small-world structure similar to the Watts α-model [83] could, for example, be used to model tightly connected clusters of neurons – so called macrocolumns [176] – to more strongly enforce the difference between specialised and distributed information processing. The actual structure of a mammalian cortex still holds many mysteries even for experts in the field and whatever structure is chosen here is merely experimental. It is to be noted, however, that the performance of the simulation is affected by the distance between neighbours, for reasons discussed further along in this chapter. The multi-GPU and cluster implementations described in Section 8.4 are slightly less flexible, as they require neighbouring neurons to be either in the same sub-lattice or in one of the directly neighbouring sub-systems.

Neurons have to query the state of all their neighbours during each simulation step. The required information includes the type of the neighbour – excitatory or inhibitory – and whether it is firing an action potential. This information is packed into the two least significant bits of an `unsigned char (Uchar)` and queried using 2D texture fetches. 2D thread blocks are used to process the neurons and maximise the reuse of data in the texture cache. The assignment of threads to neurons is illustrated in Figure 8.1. The texture fetches make full use of the knowledge that most neighbours are located nearby and that the neighbours of other neurons processed by the same thread block are in the same area as well. Many neurons located within a short distance from each other even share some of the same neighbours. The bit-packed data of the surrounding neurons is therefore transferred into the texture cache and reused frequently, significantly improving the performance of the queries.

In order to fully utilise the memory bandwidth when reading the adjacency-lists of neurons processed

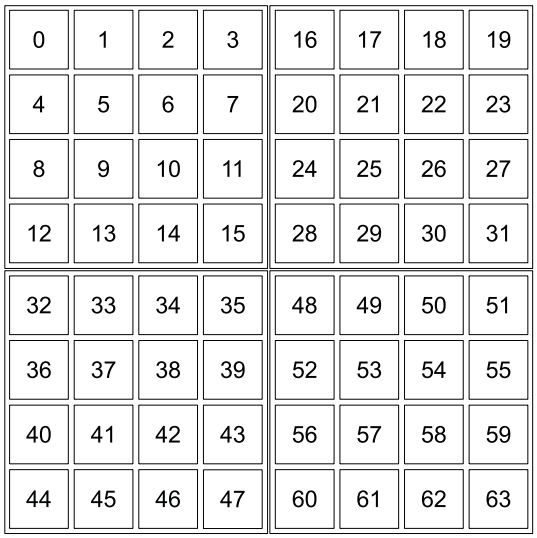


Figure 8.1.: The mapping of thread IDs to neurons in the 2D lattice structure. Threads are organised in blocks of size 4×4 in this example, the actual implementation uses blocks of size 16×16.

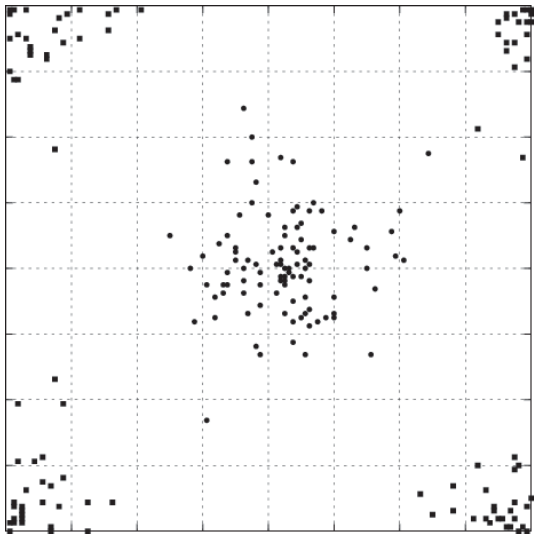


Figure 8.2.: Neighbours are chosen at random using a normal distribution centered at the source neuron. The plot shows a possible neighbour distribution for neurons $(0,0)$ and $(64,64)$ in a system of size 128×128 with maximum neighbour radius set to 48. The grid illustrates the thread blocks.

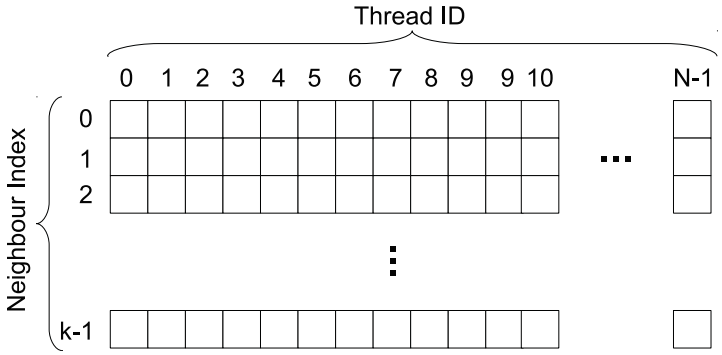


Figure 8.3.: The global neuron IDs of the source neurons of all arcs are stored in a 1D array of length $N \times k$ – here illustrated as a 2D array with N columns and k rows – where N is the system size and k is the in-degree of all neurons. The global neuron ID is the index in the system lattice when numbering the neurons in row-major order from top left to bottom right. But neurons are processed in the order of the thread IDs given in Figure 8.1. Therefore, the global neuron IDs of the source neurons are stored at index $n_{idx} \times N + tid$, where n_{idx} is the n's neighbour of the neuron that is at the end of the arc and gets processed by the thread with ID tid. This ensures fully coalesced memory transactions when accessing this array.

Algorithm 58 Pseudo-code for the initialisation steps performed by the host.

allocate device memory
generate the network structure
copy $A_d \leftarrow$ copy the adjacency-list structure to the device
copy seed values for the random number generator instances to the device
do in parallel on the device using T threads: initialise the RNGs
copy $W_c \leftarrow$ compute the washout tables using $A_{e,i}$ and $\tau_{e,i}$ and copy to constant device memory
copy the values for the current input $I_c \leftarrow I_{e,i}$ and noise $n_c \leftarrow n_{e,i}$ to constant device memory
copy $(a_c, b_c, c_c, d_c) \leftarrow (a, b, c, d)$ the values for the cell dynamics to constant device memory
do in parallel on the device using T threads: call InitKernel to initialise the simulation
bind the bit-packed data array D_d to the 2D texture reference D_{tex}

by sequential threads, the global memory accesses have to be coalesced. Once again, the decision that all neurons have the same number of incoming arcs and that neurons do not need to know about their own outgoing connections makes it possible to arrange the memory layout in a way that fully coalesced memory access can be achieved. Figure 8.3 describes how the arcs are stored. Depending on the situation, two different ways of addressing a neuron are used by the implementation. The first one is by the thread ID (tid) used to process the node, as illustrated in Figure 8.1. This is used as the offset into the adjacency-list structure, such that the first neighbour is stored at index $0 \times N + tid$, the second neighbour at index $1 \times N + tid$ and so on. The second way of addressing a neuron is by its global ID, which is its index in the system lattice when numbering the neurons in row-major order from top left to bottom right. This means that the neuron at coordinates (x, y) has global ID $y \times L + x$. The values of the elements stored in the adjacency-list are the global IDs of the neighbours. The global IDs can be used to quickly compute the coordinates of the respective nodes, in particular when the dimension length is a power of two, as it only takes a logical `AND` and a right shift (`>>`) to compute the coordinates in this case as shown in the following section. Since this improves the performance but does not impose a major limitation, the implementation always expects L to be a power of two. The coordinates of the neighbours are needed for the 2D texture fetches of the bit-packed data.

8.3. CUDA Implementation of the Cortical Model

This section describes the single-GPU CUDA implementation of the cortical model. It can be executed either in batch-mode using a configuration file that specifies the values of the system parameters and how they change throughout the simulation, or interactively using the OpenGL frontend discussed in Section 8.7. The steps described in Algorithm 58 are executed once to initialise the model before the two main CUDA kernels can be called to evolve the simulation.

The host first generates the network structure and stores it in the 1D arc array A_d described in Figure 8.3, which is then copied to the device. After this is done, memory for the following data structures is allocated:

- Memory for T instances of the CUDA implementation of the 64-bit random number generator `Ran` from Numerical Recipes [115]. This RNG is used as it does not have the requirement that all threads in a half-warp generate the exact same number of deviates for memory access to remain coalesced, which is imposed by the CUDA implementation of Marsaglia's lagged-Fibonacci generator (see Section 4.3). It is also more memory efficient than the implementation of Marsaglia's RNG, using only 24 Bytes of memory compared to 400 Bytes (lag-table length of 97). Section 4.5 shows that the performance of the two RNGs is comparable on Fermi-architecture based devices, which are the main target for this implementation.

- V_d and U_d record the current values of the Izhikevich [182] variables v (membrane potential) and u (membrane recovery) for all neurons.

- EV_d is the extra volts array, which records the additional input voltage from the action potentials of all neighbours for up to $EV_{max} = W_{max} + \delta_{max}$ simulation steps, for every neuron, where $W_{max} = 60$ is defined as the maximum washout table length and $\delta_{max} = 15$ as the maximum delay. These values are defined at compile time.

- Δ_d stores the synaptic delay δ for each neuron. A minimum delay of $\delta_{min} = 5$ and a maximum delay of $\delta_{max} = 15$ milliseconds are used in the current implementation. The delay is defined on the postsynaptic neuron and not on the link itself. This simplification reduces the memory requirements for the delay terms from $\mathcal{O}(Nk)$ to $\mathcal{O}(N)$.

- TV_d records the type values that define the dynamics of individual neurons. Because only parameters c,d for excitatory cells and a,b for inhibitory cells are modified by a random value and thus vary between cells of the same type, as defined in Section 8.1, only those two values are stored in global memory as $TV_d[tid][0]$ and $TV_d[tid][1]$. The base values are stored in constant device memory as a_c, b_c, c_c, d_c, such that $a_c[0]$ is the value for excitatory cells and $a_c[1]$ the value for inhibitory cells.

- D_d is the array of bit-packed data, where the second least significant bit (LSB) records the type of the cell and the LSB indicates whether the neuron is firing an action potential in the current simulation step.

- V_{sum} and F_{sum} record the partial sums for the voltage and firing rate of excitatory neurons as described in Algorithm 61.

Algorithm 59 describes `InitKernel` called by the host code in Algorithm 58. This kernel is called once to initialise the simulation on the GPU. It is executed by T CUDA threads. The first task performed by this kernel is to randomly choose the type for each neuron based on a given ratio. As suggested in [182], a 4:1 ratio of excitatory (type 0) to inhibitory (type 1) cells is used by default. The type is written to the second bit in array D_d, while the LSB is at the same time initialised to 0, which indicates that the neuron is not firing. This array is indexed using the global neuron ID, so that it can be bound to a texture reference and accessed using the (x,y)-coordinates when querying the state of all neighbours in kernel `Phase2`. When accessing D_d without the texture reference, as is necessary to update the values, it is important to use the correct memory row pitch as determined by `cudaMalloc3D`, which is also used to allocate 2D arrays. Memory transfers to D_d performed in this manner are partially coalesced, as the element size is 1 byte and only `BLOCK_DIM` consecutive threads access subsequent addresses. `BLOCK_DIM` is equal to 16 in the current implementation, thus 16 bytes can be transferred in a single transaction. This is only a small sacrifice compared to the advantages of accessing this array using texture fetches later on. And as the following 16 elements are needed by the next thread block, the unified L2 cache on Fermi-architecture devices further improves the memory bandwidth. All other arrays are indexed by tid, thus ensuring fully coalesced memory transfers. Each thread also uses a local counter to keep track of the number of excitatory neurons.

Based on the chosen type, the next step is to determine the individual dynamics of each neuron and to write the values to array TV_d. Then the membrane potentials V_d are initialised to some random value between the resting and firing states, which also determines the initial values of the recovery variables stored in U_d. All elements of the extra volts array EV_d are initialised to 0 using a stride of system size N between the indices for each individual neuron, thus ensuring coalesced access. Finally, the synaptic delay δ is randomly initialised to some value in the range $[\delta_{min}, \delta_{max}]$. When a thread has no more neurons to process, it adds its local counter of excitatory neurons to a counter in shared memory, which is eventually added to the global sum in device memory by the first thread in each block.

Algorithm 60 shows the host code that prepares each simulation step and calls kernels `Phase1` and `Phase2` to evolve the simulation. The current washout table length w is set to 3 times the larger of the

Algorithm 59 The CUDA kernel `InitKernel` called in Algorithm 58. `BLOCK_SIZE` $= (\texttt{BLOCK\_DIM})^2$ are the thread block size and block dimension length respectively. The values for the base 2 logarithms of the dimension length and block size are calculated once by the host and passed as parameters to the kernel.

Input parameters: fractionType0 specifies the fraction of excitatory neurons; $ntype0_d$ is a global memory counter for the number of excitatory neurons
$t \leftarrow$ global thread ID queried from the CUDA runtime
$btid \leftarrow$ thread block local thread ID queried from the CUDA runtime
load variables for the RNG instance used by this thread from global memory to registers
$ntype0 \leftarrow 0$ //local counter for excitatory neurons
$ntype0_s \leftarrow 0$ //shared memory counter for excitatory neurons
synchronise threads in block
for all $tid \in V_t := \{t, t+T, t+2T, \ldots : tid < N\}$ **do**
  $block\_ix \leftarrow (tid >> log2(\texttt{BLOCK\_SIZE})) * \texttt{BLOCK\_DIM}$
  $block\_iy \leftarrow (block\_ix >> log2(L)) * \texttt{BLOCK\_DIM}$
  $ix \leftarrow (block\_ix$ AND $(L-1)) +$ threadIdx.x
  $iy \leftarrow block\_iy +$ threadIdx.y
  $r \leftarrow$ generate a uniform random number on $[0,1)$
  type $\leftarrow r <$ fractionType0 ? 0 : 1 //ratio of type 0 (excitatory) to type 1 (inhibitory) neurons
  $D_d[iy * L + ix] \leftarrow$ type << 1 //shift the type to the second LSB; use the global neuron ID as index
  increment $ntype0$ by one if type $= 0$ //count the number of excitatory cells
  $r \leftarrow$ generate a uniform random number on $[0,1)$
  $TV_d[tid][0] \leftarrow$ type $= 0$? $(c_c[0] + 15r^2) : (a_c[1] + 0.08r)$ //initialise the cell dynamics
  $TV_d[tid][1] \leftarrow$ type $= 0$? $(d_c[0] - 6r^2) : (b_c[1] - 0.05r)$
  $b \leftarrow$ type $= 0$? $b_c[0] : TV_d[tid][1]$ //select the cell's value for b based on its type
  $r \leftarrow$ generate a uniform random number on $[0,1)$
  $V_d[tid] \leftarrow 30 - (r * 95)$ //initialise the voltage to a value in the range $[-65, 30]$
  $U_d[tid] \leftarrow b * V_d[tid]$ //initialise the recovery variable
  for $i \in \{0, 1, 2, \ldots, (EV_{max} - 1)\}$ **do**
    $EV_d[i * N + tid] \leftarrow 0$ //initialise the extra volts array to 0
  end for
  $r \leftarrow$ generate a uniform random number on $[0,1)$
  $\Delta_d[tid] \leftarrow$ use r to randomly initialise the cell's synaptic delay in the range $[\delta_{min}, \delta_{max}]$
end for
write the updated local variables for the RNG instance back to global memory
atomicAdd($ntype0_s$, $ntype0$) //add the local value to the shared memory counter
synchronise threads in block
if btid = 0 **then**
  atomicAdd($ntype0_d$, $ntype0_s$) //add the value from shared memory to the global memory counter
end if

effective $\tau_{e,i}$ values, after they are modified by the respective drug levels $\lambda_{\tau e}$ and $\lambda_{\tau i}$. The final value is limited to the maximum table length W_{max}. A warning is printed if the result would exceed W_{max}, notifying the user that the current washout constant is too large for the chosen table length and that the effects of action potentials are cut off sooner than intended. The factor of 3 is chosen as the cut off value after which the remaining effects of an action potential are $e^{-3\tau/\tau} < 0.05$ the initial value and thus become negligible.

Algorithm 61 describes kernel `Phase1`. The main tasks of this phase are to update the membrane potential and recovery variables for each neuron and to determine whether the cell is firing an action potential in the current simulation step. The kernel calculates the (x,y)-coordinates for the neuron and uses them to load its own bit-packed data from D_d in the same way as it is done in `InitKernel`. The type of the neuron is extracted from this data. Then the cell's input current I is computed.

As mentioned in Section 8.1, I is the input current from sources that are external to the model. These are simulated by the constant $I_{e,i}$ value, plus the normally distributed random noise $I_{noise} = n_{e,i} \times r_{norm}$ for excitatory and inhibitory neurons respectively, plus the optional boost I_{boost} to a random fraction I_{frac} of all neurons. The current values for $I_{e,i}$ and $n_{e,i}$ are stored in constant memory as I_c and n_c respectively. Both are of length two, with the first element holding the value for excitatory neurons and the second element holding the value for inhibitory neurons. Constant memory can be updated by the host code between simulation

Algorithm 60 This host function is called to evolve the simulation by `STEPS` simulation steps.

$w \leftarrow min(W_{max}, (3 * max(\tau_e/\lambda_{\tau e}, \tau_i * \lambda_{\tau i})))$ //the current washout table length
for $s \leftarrow 1$ to `STEPS` **do**
  do in parallel on the device using T threads: call kernel `Phase1`(EV_{idx})
  wait until `Phase1` is completed
  $EV_{idx} \leftarrow (EV_{idx} + 1) < EV_{max}\ ?\ EV_{idx} + 1 : 0$ //increment the index into the extra volts array EV_d
  copy V_{sum} from device memory to host memory //asynchronous, can overlap with kernel `Phase2`
  copy F_{sum} from device memory to host memory //asynchronous, can overlap with kernel `Phase2`
  do in parallel on the device using T threads: call kernel `Phase2`(EV_{idx}, w)
  wait until V_{sum} and F_{sum} have been copied to host memory
  $v_{avg} \leftarrow f_{avg} \leftarrow 0$ //the average voltage v and firing rate f of all excitatory neurons
  for $i \leftarrow 0$ to $(32 *$ [number of thread blocks]$)$ **do**
    $v_{avg} \leftarrow v_{avg} + V_{sum}[i]$
    $f_{avg} \leftarrow f_{avg} + F_{sum}[i]$
  end for
  $v_{avg} \leftarrow v_{avg}/ntype0$ //$ntype0$ is the number of excitatory neurons
  $f_{avg} \leftarrow f_{avg}/ntype0$
end for

steps. I_{frac} is specified as a parameter to the kernel, while the magnitude of the boost itself is set to a fixed value.

To compute the noisy input I_{noise}, a standard normally distributed random deviate has to be generated first. The Box-Muller transformation is used to convert two uniform random deviates into two independent Gaussian deviates. The implementation for this conversion is essentially the same as the one given in Numerical Recipes [115], only that it converts the 64-bit random deviates to 32-bit single precision and uses intrinsic functions for the floating point division and logarithm. Note that the uniform deviates are generated with full 64-bit precision, only the conversion process is performed in lower precision. Doing the conversion in full double precision adds approximately $4-5\%$ to the execution time of the entire simulation (tested on a GTX580 with $N = 1024^2$ and $k = 100$). Since the Box-Muller method generates two deviates at a time, it is only called for every second iteration of the main loop. The spare value is stored in a register and used in the following iteration. While the authors of Numerical Recipes suggest to use the Ratio-of-Uniforms method to generate normally distributed deviates, which is faster on the CPU, it causes more warp divergence than the Box-Muller transformation and is therefore slower on the GPU.

Next the values a, b, c, d for the Izhikevich neurons are loaded. Depending on the type, either a, b or c, d are loaded from constant memory, which means that the values are the same for all neurons of that type. The pair of individually varying values, previously determined in `InitKernel`, is loaded from TV_d. Then the cell's membrane potential v and recovery variable u are loaded from global memory. They are reset to the after-spike defaults defined in Equation 8.2 if the neuron has fired an action potential during the previous simulation step. Then the change in the voltage and recovery variables is computed using Equation 8.1. Finally, the neuron's individual synaptic delay δ is loaded from Δ_d and used to compute the index into the extra volts array EV_d. The value stored at this index represents the accumulated effects from action potentials fired by neighbouring cells that reach the neuron at this time step. This includes all action potentials fired at time steps in the range $[-(1+\delta), -(\delta + w)]$ from the current time. These inputs change the neuron's membrane potential and are thus added to the new value of v.

The value of v is limited to a maximum of 30mV, at which point the neuron is firing. The new values are written to global memory and the current element in the extra volts array is reset to zero. Finally, every thread accumulates the membrane potentials of excitatory neurons and increments a counter for the number of firing excitatory cells in shared memory. Once all threads in a thread block are done processing neurons, they perform a parallel prefix-sum on these values in shared memory. The process is stopped when the values from all indices ≥ 32 have been accumulated into the first 32 values. Any further steps would cause warp divergence and are avoided. Instead, the first warp writes these partial sums to global memory.

Algorithm 61 The CUDA kernel `Phase1` called in Algorithm 60. The values for the base 2 logarithms of the dimension length and block size are calculated once by the host and passed as parameters to the kernel. Note that only the Box-Muller transformation used to generate normally distributed deviates can cause a small amount of warp divergence in this kernel. All other conditionals always evaluate to the same value for all threads in the same warp.

Input parameters: EV_{idx} is the current index into the extra volts array; I_{frac} specifies the fraction of neurons to receive a boost to I (default 0.0)
$t \leftarrow$ global thread ID queried from the CUDA runtime
$btid \leftarrow$ thread block local thread ID queried from the CUDA runtime
load variables for the RNG instance used by this thread from global memory to registers
$v_s[\texttt{BLOCK\_SIZE}] \leftarrow 0$ //shared memory for partial voltage sums initialised to 0
$f_s[\texttt{BLOCK\_SIZE}] \leftarrow 0$ //shared memory for partial firing rate sums initialised to 0
$r_{norm}[2]$ //two 32-bit registers for normally distributed random numbers
$r_{flag} \leftarrow 1$ //flag to indicate that r_{norm} needs to be updated
for all $tid \in V_t := \{t, t+T, t+2T, \ldots : tid < N\}$ **do**
  $block\_ix \leftarrow (tid >> log2(\texttt{BLOCK\_SIZE})) * \texttt{BLOCK\_DIM}$
  $block\_iy \leftarrow (block\_ix >> log2(L)) * \texttt{BLOCK\_DIM}$
  $ix \leftarrow (block\_ix$ AND $(L-1)) +$ threadIdx.x
  $iy \leftarrow block\_iy +$ threadIdx.y
  $dat \leftarrow D_d[iy * L + ix]$ //load the cell's own data from D_d
  type $\leftarrow ((dat$ AND $(1 << 1)) >> 1)$ //extract the type
  if $I_{frac} > 0$ **then**
    $r \leftarrow$ generate a uniform random number on $[0, 1)$
    $I_{boost} \leftarrow r < I_{frac}$? 10 : 0 //add an extra boost to a fraction of cells
  end if
  if $r_{flag} = 1$ **then**
    $r_{norm} \leftarrow$ generate two normally distributed random numbers for the noisy input to I
    $r_{flag} \leftarrow 0$ //use the second deviate in the next iteration
  else
    $r_{norm}[0] \leftarrow r_{norm}[1]$ //use the second deviate from the previous iteration
    $r_{flag} \leftarrow 1$
  end if
  $I \leftarrow I_c[\text{type}] + r_{norm}[0] * n_c[\text{type}] + I_{boost}$ //calculate the cells' input current I
  $I \leftarrow I > 0$? I : 0 //no negative current
  $a \leftarrow \text{type} = 0$? $a_c[0] : TV[tid][0]$ //constant value for excitatory cells, individual value for inhibitory cells
  $b \leftarrow \text{type} = 0$? $b_c[0] : TV[tid][1]$ //constant value for excitatory cells, individual value for inhibitory cells
  $c \leftarrow \text{type} = 0$? $TV[tid][0] : c_c[1]$ //individual value for excitatory cells, constant value for inhibitory cells
  $d \leftarrow \text{type} = 0$? $TV[tid][1] : d_c[1]$ //individual value for excitatory cells, constant value for inhibitory cells
  $v \leftarrow V_d[tid]$, $u \leftarrow U_d[tid]$ //load the values for the cell's membrane potential and recovery variables
  $u \leftarrow v < 30$? $u : u + d$ //reset the recovery variable if the cell fired during the previous simulation step
  $v \leftarrow v < 30$? $v : c$ //reset the membrane potential if the cell fired during the previous simulation step
  $dv \leftarrow 0.04v^2 + 5v + 140 - u + I$ //Izhikevich's formula for the membrane potential
  $du \leftarrow a(bv - u)$ //Izhikevich's formula for the recovery variable
  $ev_{idx} \leftarrow EV_{idx} - \Delta_d[tid]$ //subtract the cell's delay δ from the current EV_{idx}, wrap around if necessary
  $new\_v \leftarrow v + dv + EV_d[ev_{idx} * N + tid]$ //the updated v including the input current from neighbouring cells
  $new\_v \leftarrow new\_v > 30$? 30 : new_v //limit the voltage to 30mV $\Rightarrow$ cell fires an action potential
  $V_d[tid] \leftarrow new\_v$
  $U_d[tid] \leftarrow u + du$
  $EV_d[ev_{idx} * N + tid] \leftarrow 0$ //reset the value for the input current
  $ap \leftarrow new\_v < 30$? 0 : 1 //0 = no action potential, 1 = the cell is firing
  $D_d[iy * L + ix] \leftarrow ((\text{type} << 1)$ OR $ap)$ //bit-pack the type and 'fired' bits and store to global memory
  $v_s[btid] \leftarrow \text{type} = 0$? $(v_s[btid] + new\_v) : v_s[btid]$ //sum the voltages for excitatory neurons
  $f_s[btid] \leftarrow (\text{type} = 0)$ and $(ap = 1)$? $(f_s[btid] + 1) : f_s[btid]$ //number of firing excitatory cells
end for
write the updated local variables for the RNG instance back to global memory
do a parallel prefix-sum on v_s, stop when the first 32 elements contain the partial sums for all higher indices
do a parallel prefix-sum on f_s, stop when the first 32 elements contain the partial sums for all higher indices
if $btid < 32$ **then**
  $V_{sum}[\texttt{BLOCK\_INDEX} * 32 + btid] \leftarrow v_s[btid]$ //write 32 partial sums to global memory
  $F_{sum}[\texttt{BLOCK\_INDEX} * 32 + btid] \leftarrow f_s[btid]$
end if

Algorithm 62 The CUDA kernel `Phase2` called in Algorithm 60. The value for the base 2 logarithm of the dimension length is calculated once by the host and passed as parameters to the kernel.

Input parameters: w is the current washout table length and EV_{idx} is the current index into the extra volts array
$t \leftarrow$ global thread ID queried from the CUDA runtime
$btid \leftarrow$ thread block local thread ID queried from the CUDA runtime
ap_s[BLOCK_SIZE∗2] //shared memory array to count excitatory and inhibitory action potentials
for all $tid \in V_t := \{t, t+T, t+2T, \ldots : tid < N\}$ **do**
  $ap_s[btid] \leftarrow ap_s$[BLOCK_SIZE+$btid$] $\leftarrow 0$ //initialise both counters
  for $n_i \leftarrow 0, 1, 2, \ldots, (k-1)$ **do**
    //this loop is unrolled 20× for improved performance
    $n_{gid} \leftarrow A[n_i * N + tid]$ //load the global ID for the n_i's neighbour from the arc array
    $n\_ix \leftarrow n_{gid}$ AND $(L-1)$ //the neighbour's x-coordinate
    $n\_iy \leftarrow n_{gid} >> log2(L)$ //the neighbour's y-coordinate
    $dat \leftarrow D_{tex}(n\_ix, n\_iy)$ //load the neighbour data using a texture fetch
    type $\leftarrow ((dat$ AND $(1 << 1)) >> 1)$ //extract the type
    $ap \leftarrow (dat$ AND $1)$ //extract the 'fired' flag
    ap_s[type∗BLOCK_SIZE+$btid$] $\leftarrow ap_s$[type∗BLOCK_SIZE+$btid$] $+ ap$ //increment the counter for this cell type
  end for
  $ev_{idx} \leftarrow EV_{idx}$
  for $w_i \leftarrow 0, 1, 2, \ldots, (w-1)$ **do**
    $ev_{idx} < EV_{max}$? $ev_{idx} : (ev_{idx} - EV_{max})$ //circular buffer
    $my_{idx} \leftarrow ev_{idx} * N + tid$ //the ev_{idx}'s element in the neuron's private extra volts array
    $ev_{val} \leftarrow EV_d[my_{idx}]$ //load the current value from the extra volts array
    $ev_{val} \leftarrow ev_{val} + ap_s[btid] * W_c[w_i]$ //add the effects of all excitatory action potentials
    $ev_{val} \leftarrow ev_{val} + ap_s$[BLOCK_SIZE+$btid$] $* W_c[W_{max} + w_i]$ //add the effects of all inhibitory action potentials
    $EV_d[my_{idx}] \leftarrow ev_{val}$ //write the updated value back to global memory
    $ev_{idx} \leftarrow ev_{idx} + 1$
  end for
end for

Algorithm 60 describes how the host copies these arrays to system memory after it has started execution of kernel `Phase2` on the GPU. The host adds up the partial sums to obtain the final sums for the entire system. From these, it computes the average membrane potential of all excitatory neurons as well as the fraction of currently firing excitatory neurons. The average membrane potential is used to generate the raw pseudo-EEG signal, which is then further processed as discussed in Section 8.6. Only the membrane potential from excitatory neurons is used to generate the pseudo-EEG, as it is generally accepted that, due to the arrangement of their dendrites and axons, the scalp-measured EEG voltage signal is dominated by the fluctuations in excitatory cells [176]. The author has experimented with a different way of computing the pseudo-EEG signal based on the method discussed by Talavera et. al. [185], but this approach is not actively used in the current implementation.

Algorithm 62 describes kernel `Phase2`. In this phase, every neuron queries the state of all its neighbours to determine which ones are firing an action potential. The neighbour IDs are looked-up from the arcs array A_d with fully coalesced reads as visualised in Figure 8.3. The arc references are stored using the global IDs of the neighbouring cells, which can be quickly converted into the coordinates that are needed to access the data array D_d using the 2D texture reference D_{tex}. The information found in the values loaded from D_d includes the neighbour's type and firing state. Nothing further is done if the neighbour is not firing an action potential. But if it is, then the counter for the respective type of the neighbouring cell is incremented in shared memory. Once the data from all neighbours has been queried, the two counters state the number of incoming excitatory and inhibitory action potentials respectively.

Note how the indexing of the shared memory array ap_s avoids shared memory bank conflicts. Successive threads in the same thread block either access consecutive 4-byte words, which maps to different 32-bit memory banks for each thread in a half-warp (16 memory banks per multiprocessor for CC 1.x [90]) or warp (32 memory banks for CC 2.x), or they use an offset of `BLOCK_SIZE`, which is always chosen to be a multiple of the number of shared memory banks available on the multiprocessor.

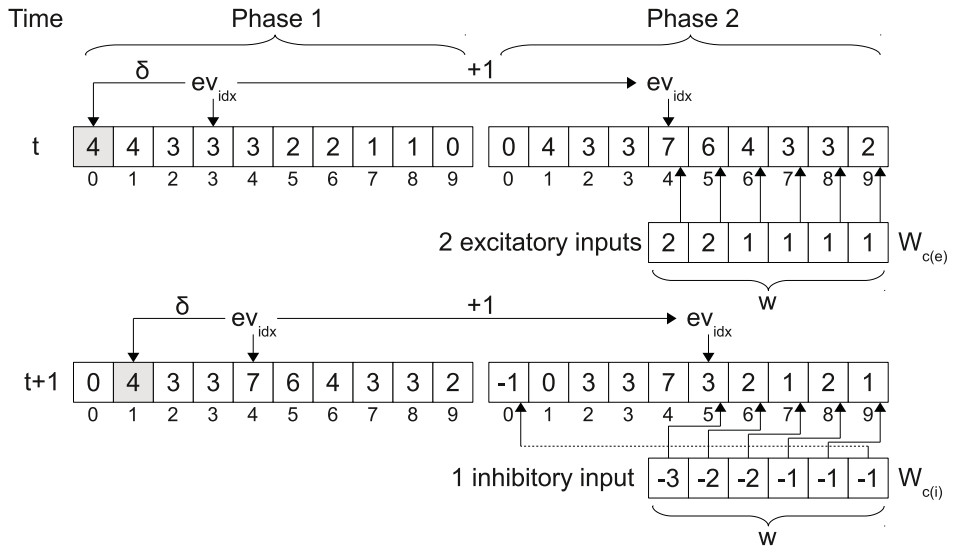


Figure 8.4.: This diagram illustrates how the extra volts array EV_d for one neuron is indexed and modified over the two phases of the simulation. Note that the actual implementation uses a stride of N between indices into this array, as the array elements for all neurons are interleaved to facilitate coalesced memory transactions. The neuron's delay is $\delta = 3$ and the current washout table length is $w = 6$ in this example. At time step t, kernel `Phase1` reads the input value from index $ev_{idx} - \delta = 3 - 3 = 0$ and resets this value to 0 when it is finished. Index ev_{idx} is incremented before kernel `Phase2` is called. The kernel first queries all neighbours and finds that two of them are firing an excitatory action potential. It then proceeds to add $2\times$ the values from the excitatory washout table $W_{c(e)}$ to the corresponding w elements of the extra volts array, beginning with index $ev_{idx} = 4$. In the first phase of the next simulation step $t+1$, ev_{idx} is still equal to 4, thus the element at index 1 is read after the delay is subtracted. Then ev_{idx} is incremented again before `Phase2` is executed. This time a single inhibitory action potential is being fired by one of the neighbours and, thus, $1\times$ the values from the inhibitory washout table $W_{c(i)}$ are added to the w elements of the extra volts array, this time beginning with index $ev_{idx} = 5$. The extra volts array is a circular buffer, which wraps around to connect the first and last elements seamlessly.

The entire loop is unrolled 20 times, which tests have shown to be a good number for the GeForce GTX480 and GTX580. On the GTX580, unrolling the loop 20 times reduces the execution time by about 13% compared to an implementation with no loop unrolling (tested for $N = 1024^2$ and $k = 100$). The optimal value mainly depends on the number of registers available per multiprocessor and the actual number of registers used by the kernel. Since the former can differ between devices and the latter can change between compiler versions, the code may perform slightly better with a different value on another system. But generally speaking, it pays off to unroll a few iterations of this loop. As no additional checks are performed between unrolled iterations, it is important to ensure that the in-degree is always a multiple of the chosen value.

The effects of the received action potentials, which decay over time as defined in the washout table, are added to the values stored at the next w indices of the neuron's extra volts array EV_d. All neurons index EV_d with the same major offset $EV_{idx} \times N$, which is increased with stride N for every index in the current washout table, plus a minor offset that is equal to the thread ID. With this memory access strategy, all w memory reads and w memory writes to EV_d performed in this phase can be fully coalesced. It now becomes obvious why it is much preferable to add the individual delay δ, which each neuron applies to its incoming action potentials, to the single read and write from and to the extra volts array during `Phase1`.

Table 8.1.: This table lists the device memory requirements in bytes per neuron.

V_d	U_d	A_d	EV_d	Δ_d	TV_d	D_d	V_{sum}	F_{sum}
4	4	$4k$	$4EV_{max}$	1	8	1	0.5	0.5

This sacrifices a little bit of performance in `Phase1` for a huge performance gain in `Phase2`. During each of the w iterations, all threads in a half-warp concurrently access one of two addresses in the washout table – one for the current excitatory value and the other one for the current inhibitory value. And since the washout table resides in constant memory, most reads result in a cache hit and can therefore be served very quickly. The entire process involving the extra volts array is visualised in Figure 8.4.

The device memory requirements per neuron for this implementation are listed in Table 8.1. They are dominated by the chosen in-degree k and by the maximum extra volts array length EV_{max}, which is mostly determined by the maximum washout table length W_{max}. In addition to the memory requirements per neuron, $24T$ bytes of device memory are used for the RNG instances regardless of the system size. Since the system size is always a power of two, a value of T that is also a power of two gives the best performance on all tested devices. The value should be relatively large, in the hundred thousands, as a small value has a significantly larger effect on the execution times than the exact choice of the value. It should not be excessively large either, however, considering the related memory requirements for the RNG instances. A value of $T = 2^{19} = 524,288$ is used for the performance measurements in Section 8.5, unless $N < 2^{19}$, in which case $T = N$.

As mentioned in the beginning of the section, the simulation can be executed in batch-mode using a configuration file to define the settings used for any given simulation step. This currently includes the values used for $A_{e,i}, I_{e,i}, n_{e,i}, \lambda_{\tau(e,i)}, \lambda_{A(e,i)}, \lambda_{I(e,i)}$ for both excitatory and inhibitory neurons each, as well as the generally applicable I_{frac}, but can easily be extended to additional settings. The settings are specified in the following format:

```
0     SET_E_BASE_I      3.0
0     SET_I_BASE_I      2.0
0     SET_E_NOISE       4.5
0     SET_I_NOISE       3.0
50    SET_BOOST_I       0.1
100   SET_BOOST_I       0.0
4000  SET_E_LAMBDA_TAU  3.0
5000  SET_E_LAMBDA_TAU  1.0
```

The value in the first column specifies the simulation step when the setting becomes active. The second column specifies the property to be changed and the last column defines the new value. Commands starting with SET_E change the specified excitatory value and commands starting with SET_I change the corresponding inhibitory value. Values never changed explicitly remain at their default values. The sample configuration shown above sets values $I_e = 3.0$, $I_i = 2.0$, $n_e = 4.5$ and $n_i = 3.0$ before the first simulation step is performed. After 50 time steps, a boost to I is activated to 10% of the neurons, lasting another 50 steps. Before simulation step 4000 is executed, $\lambda_{\tau e}$ is increased to 3.0, which increases the rate of the exponential washout of excitatory action potentials, thus reducing excitatory effects and simulating the administration of an anaesthetic. At time step 5000, $\lambda_{\tau e}$ is returned to its default value, allowing the drug effects to wear off again. The average excitatory action potentials and firing rates are recorded throughout the simulation run and written to files to be analysed later on.

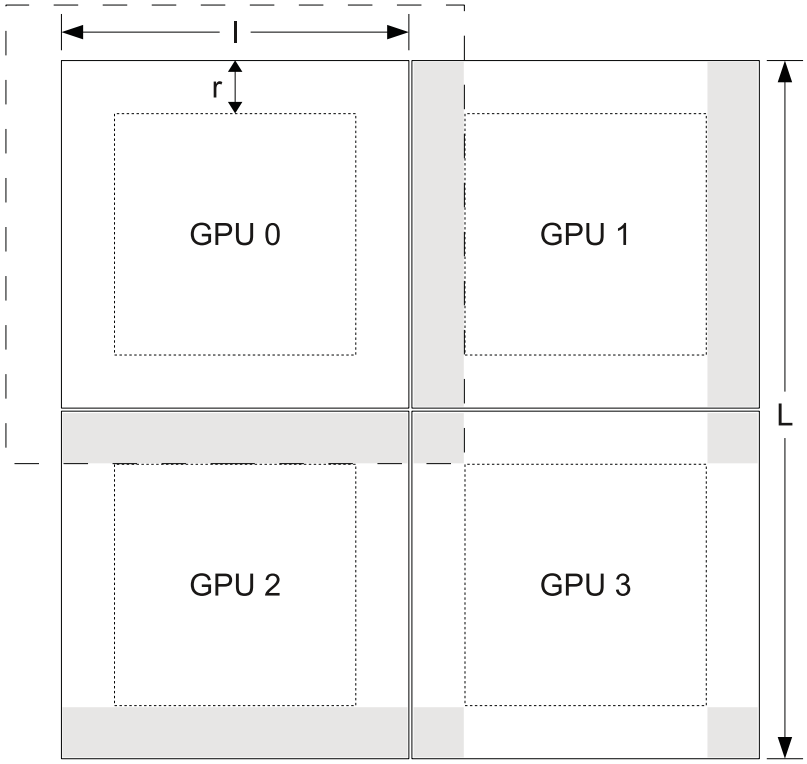


Figure 8.5.: The distributed model for multi-GPU processing. The entire system of size $L \times L$ is divided into multiple sub-systems with local dimension length $l = L/2$ when using 4 GPUs, $l = L/4$ when using 16 GPUs, and so forth. The local system processed by a single device is further divided into the inner core – the neurons within the finely dashed area – and the local border – which comprises all cells in the area between the inner core and the outer boundaries of the local system (continuous line). Thus, the inner core has size $(l-2r)^2$ and the local border has size $l^2-(l-2r)^2$, where r is the maximum neighbour distance. Kernel `Phase2` requires the bit-packed data from all presynaptic neurons connected to the neuron that is being processed. To update the extra volts arrays of neurons located in the local border, it is therefore necessary to obtain some data from the local borders of the neighbouring sub-systems. This outer border is illustrated for GPU 0 by the coarsely dashed line, as well as by the highlighted areas, which take periodic boundary conditions into account. It includes all neurons surrounding the local system up to distance r.

8.4. Multi-GPU and Cluster Implementations

This section describes how the cortical model can be distributed across multiple-GPUs to either reduce the execution times or to increase the system size beyond the memory limit of a single GPU. This has been implemented both for multi-GPU, shared memory host systems and for distributed systems utilising the Open MPI [186] implementation of the message passing interface (MPI). Apart from the data exchange mechanisms, both implementations are identical.

Figure 8.5 illustrates the sub-division of the cortical model for multi-device processing. While kernel `Phase1` only requires local information and, therefore, does not require any data exchange between devices, kernel `Phase2` has to look-up the status of all presynaptic neurons that is recorded in array D_d. As discussed in Section 8.2, the maximum distance between neighbouring neurons is equal to the radius r. Kernel Phase2 thus requires data from the local borders of neighbouring sub-systems that are processed by different devices. However, the cells within the inner core of the local system do not require any data from cells outside the local system. These cells can therefore be updated by the local GPU while the data required for

Algorithm 63 The major tasks performed by the host threads of the multi-GPU and cluster implementations. The host threads coordinate the simulation progress and data exchange among each other.

```
1: if root node then
2:     use the given seed to initialise the local RNG
3:     generate random seeds for all other host threads and send them to those hosts
4: else
5:     wait for the seed value and initialise the local RNG once it has been received
6: end if
7: generate the local network structure
8: initialise the GPU as it is done in the single-GPU implementation (Algorithm 58)
9: create local buffers for data exchange with other host threads
10: for s ← 1 to STEPS do
11:     read the batch configuration file and make the required changes
12:     do in parallel on the device using T threads: run kernel Phase1 for the entire local system
13:     wait until Phase1 is completed
14:     copy V_sum from device memory to host memory //asynchronous, can overlap with kernel Phase2
15:     copy F_sum from device memory to host memory //asynchronous, can overlap with kernel Phase2
16:     do in parallel on the device using T threads: run kernel Phase2 for the inner core of the local system
17:     copy data D_d for neurons located within the local border from device memory to host memory
18:     send/receive required border data to/from host threads that are processing neighbouring sub-systems
19:     wait until all required data has been received
20:     copy data D_d for neurons located within the outer border from host memory to device memory
21:     do in parallel on the device using T threads: run kernel Phase2 for the local border
22:     wait until V_sum and F_sum have been copied to host memory
23:     compute the local sums V_local and F_local from all elements in V_sum and F_sum
24:     wait for both calls of kernel Phase2 to complete
25:     perform a parallel reduce operation to compute the system wide sums for the values in V_local and F_local
26:     if root node then
27:         compute the average excitatory membrane potential and firing rate from the results of the reduce operation
28:     end if
29: end for
```

the cells in the local border is being transferred from other GPUs using asynchronous memory copies and multiple CUDA streams. Depending on the radius r and local dimension length l, a more or less significant part of the copy times can be hidden in this way. The entire process is described in Algorithm 63.

Here, the term host thread is used to refer to an execution stream that is in charge of a single GPU and the associated sub-system of the cortical network. If the application is running in single host, multiple GPUs mode, then host threads are created using the PThreads library, the `pthread_barrier_wait` instruction is used to synchronise between multiple host threads and shared memory is used to exchange data. If the application is running in MPI mode, then the term host thread refers to a single MPI process. The function `MPI::Request::Waitall` is used to wait for data from other MPI processes and to synchronise the simulation progress at the same time. Data is exchanged using standard MPI message send and receive functions. Every MPI process has a unique global MPI world rank as well as a local rank that is unique among the MPI processes running on the same host system. The global MPI rank is used to determine the position of the local sub-system within the entire neural network, whereas the local MPI rank is used to determine which GPU in the local host system, assuming there is more than one, the MPI process uses.

If bi-directional connections are allowed, then each host thread can generate the network structure for its own sub-system. There is no need to communicate any details about the local structure to other host threads, as arcs are only stored in the adjacency-list of the receiving neuron. And since each process knows where its local system is located within the entire network structure, it can select presynaptic neurons from its outer border as if they were part of the local sub-system. By doing so, the distributed sub-systems become seamlessly connected. However, if bi-directional connections are not allowed, then it would be necessary to negotiate which neuron gets to establish a particular link. This would be time consuming, in particular in the case of the MPI implementation running over a network, which is why in this situation the graph

structure is simply generated beforehand on one system and then copied to all participating hosts.

When the network has been generated and the local GPU has been initialised, kernel `Phase1` can be executed for all neurons in the local system, as all required information is available locally. Once `Phase1` is completed, `Phase2` can be executed for all neurons in the inner core of the local system, as they also only require local data. Concurrently with this phase, the local border is copied to the host system and different parts of it are transferred to the devices that are processing the surrounding sub-systems as illustrated in Figure 8.5. Once the data for the outer border has been received, kernel `Phase2` can be called once more, this time to process the neurons located in the local border. It is worth noting that all steps from line 14 to line 18 of Algorithm 63 can be performed in parallel. And if the first call to `Phase2` has not completed by the time the data for the second call has been copied to the device, then both parts of the system can be processed concurrently on devices that support this feature (requires compute capability 2.x).

The CUDA kernel implementations are mostly unchanged from those described in the previous section. The only modifications needed to `InitKernel` and `Phase1` are related to the addressing of the bit-packed data array D_d. Because `Phase2` requires the data from the outer border, this array is of size $(l+2r)^2$ instead of size l^2 ($L = l$ in the single-GPU implementation). To address the data for the neurons in the local system, it is therefore necessary to add a padding of length r to both the x and y-coordinates of every cell. In `Phase2`, the kernel needs to distinguish between calls to process the the neurons located in the inner core and calls to process the neurons located in the local border. Algorithm 64 shows how the correct thread ID tid is computed in the two different cases and how the kernel has been turned into a C++-style template function to avoid having to duplicate the code shared by the different scenarios. With this approach, the compiler generates two specialised versions of the kernel code that do not include any of the instructions specific to the respective other version. For the calculation of the correct thread IDs, it is important to remember how threads are mapped to neurons (see Figure 8.1 on page 151).

8.5. Performance Results

This section describes the performance of the single and multi-GPU implementations. The performance is measured both in terms of the absolute execution times for a particular configuration as well as by a comparison of the maximum system sizes that can be achieved on a specific device. A number of different devices are used as described in Table 8.2. All results are averaged over 10 independent simulation runs. Error bars representing the standard deviations are smaller than the symbol size.

The tests use a batch configuration that sets the following values for the entire simulation: $I_e = 3.0$, $n_e = 4.5$, $I_i = 2.0$ and $n_i = 3.0$. Additionally, it sets $\lambda_{\tau e} = 3$ before simulation step number 4000 is executed and returns it to its default $\lambda_{\tau e} = 1$ before step 5000. All other values are set to the defaults mentioned in Sections 8.1 and 8.3. Figure 8.6 shows an example of the pseudo-EEG produced with this configuration when using a system size of $N = 1024^2$ neurons and an in-degree of either $k = 100$ or $k = 200$.

Figure 8.7 shows the execution times for a range of system sizes with a fixed in-degree per neuron of $k = 100$ and a maximum neighbour distance $r = 64$. Not every system size is supported on all system configurations. The shared memory and MPI multi-GPU implementations require a system size of at least $N = 256 \times 256$ with a local dimension length of $l = 128$, as l is required to be at least twice the maximum neighbour distance. On the other hand, not all devices have enough global memory to simulate a neural network larger than $N = 1024 \times 1024$. The relative results of the single-GPU implementations offer no surprises. The GTX580 is the fastest GPU out of the tested devices. It can compute one second of simulated time in the cortical model with $N = 262,144$ neurons and a total of over 26 million neural connections in about 1.85 seconds of real time. With a resolution of one millisecond, one thousand simulation steps are performed for each simulated second. The GTX580 is closely followed by the GTX480 and with a larger distance by the M2070 and GTX260. Although the M2070 has only 32 CUDA cores less than the GTX480,

Algorithm 64 The modifications to kernel `Phase2` that are needed to distinguish between computing the inner core and the local border of the system. The compiler generates two specialised versions for the 'true' and 'false' case of the template parameter `isInner`. The maximum neighbour distance r must be a multiple of the thread block dimension length BLOCK_DIM_Y. All of the if-conditions can either be removed by the compiler or are implemented as ternary expressions to avoid warp divergence.

template<bool isInner>
  $t \leftarrow$ global thread ID queried from the CUDA runtime
  ... (same as Algorithm 62)
  for all $tid_{tmp} \in V_t := \{t, t+T, t+2T, \ldots : tid < N\}$ **do**
    if isInner = true **then**
      //compute tid for cells located in the inner core
      $tid \leftarrow tid_{tmp} + (l * r)$ //top padding
      //add padding for the left and right borders of every row of thread blocks
      $tid \leftarrow tid + ((tid_{tmp} / (((l * \text{BLOCK\_DIM\_Y}) - (2 * r * \text{BLOCK\_DIM\_Y}))) * (2 * r * \text{BLOCK\_DIM\_Y}))$
      $tid \leftarrow tid + (r * \text{BLOCK\_DIM\_Y})$ //add padding for the left border of the final row of blocks
    else
      //compute tid for cells located in the local border
      $idx0 \leftarrow (l * r) + (r * \text{BLOCK\_DIM\_Y})$ //the first thread to require any padding
      $idx1 \leftarrow (l * (l - r)) - (l - (2 * r))^2$ //the first thread to process a neuron from the lower local border
      if $tid_{tmp} < idx1$ **then**
        //the thread is processing a neuron that is located outside of the lower local border
        if $tid_{tmp} < idx0$ **then**
          $tid \leftarrow tid_{tmp}$ //before the inner core begins, no padding required
        else
          //Every row of thread blocks consists of a left and a right border of size r*BLOCK_DIM_Y and the inner
          //core of size $(l-(2*r))$*BLOCK_DIM_Y. Compute the current row, add one for the first row of the inner
          //core, and multiply by the area per row of blocks taken up by the inner core.
          $tid \leftarrow tid_{tmp} + ((1 + ((tid_{tmp} - idx0) / (2 * r * \text{BLOCK\_DIM\_Y}))) * ((l - (2 * r)) * \text{BLOCK\_DIM\_Y}))$
        end if
      else
        //the thread is processing a neuron that is located in the lower local border
        $tid \leftarrow tid_{tmp} + (l * (l - r)) - idx1$
      end if
    end if
    ... (same as Algorithm 62)
    for $n_i \leftarrow 0, 1, 2, \ldots, (k-1)$ **do**
      $n_{gid} \leftarrow A[n_i * l^2 + tid]$ //load the global ID for the n_i's neighbour from the arc array
      $n\_ix \leftarrow n_{gid} \% (l + (2 * r))$ //the neighbour's x-coordinate (takes padding for the outer border into account)
      $n\_iy \leftarrow n_{gid} / (l + (2 * r))$ //the neighbour's y-coordinate (takes padding for the outer border into account)
      ... (same as Algorithm 62)
    end for
    ... (same as Algorithm 62)
  end for
  ... (same as Algorithm 62)

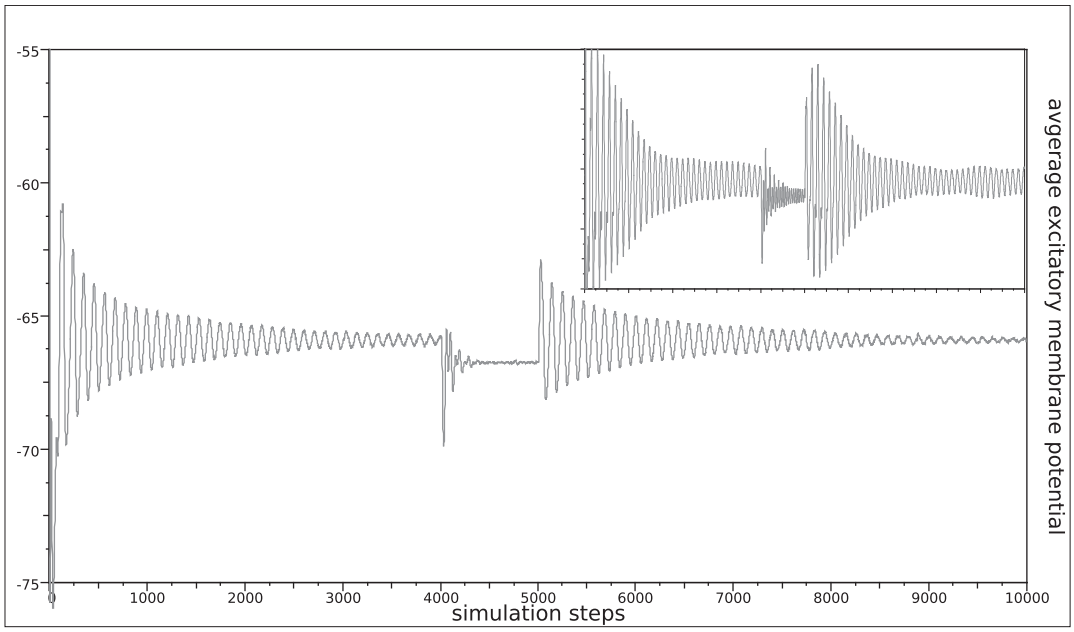


Figure 8.6.: An example for the pseudo-EEG generated with the test configuration that is used for the performance measurements. System size $N = 1024^2$ and in-degree $k = 100$ for the main plot and $k = 200$ for the inset. The inset is plotted on the same scale as the main graph and illustrates how the magnitude of the membrane potential changes when the degree is increased but all other parameters are kept the same.

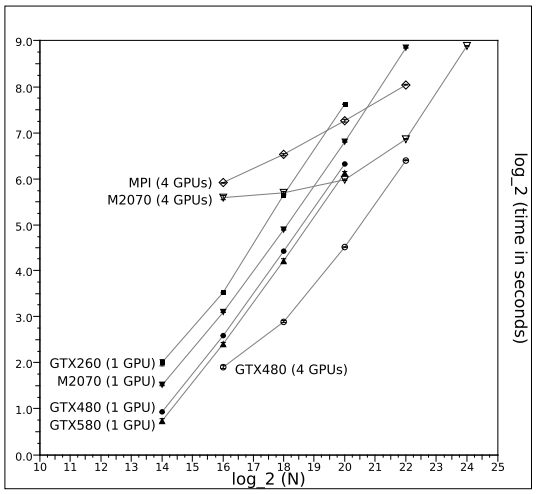


Figure 8.7.: The execution times for 10,000 simulation steps with system sizes ranging from $N = 2^{14}$ to $N = 2^{24}$. The in-degree is set to $k = 100$, which adds up to a maximum of $\approx 1.68 \times 10^9$ neural connections in the largest system. The the maximum neighbour distance $r = 64$.

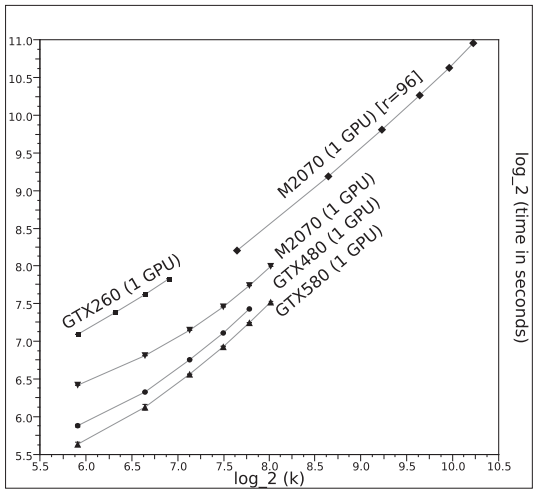


Figure 8.8.: The execution times for 10,000 simulation steps with in-degrees ranging from $k = 60$ to $k = 1200$. The system size is $N = 2^{20}$ and the maximum neighbour distance is $r = 64$, except for the second result set for the M2070 ($k = 200 - 1200$), which uses $r = 96$.

Table 8.2.: The test systems used for the performance measurements.

GTX260	The only pre-Fermi architecture device used for these tests. The GPU has 896 MB of global memory. The host uses an Intel Core 2 Quad Q8200 CPU at 2.33 GHz and 8 GB of main memory.
GTX480	The system hosts four NVIDIA GTX480 GPUs, each installed in a separate PCI Express $\times 16$ slot. The GPUs have 1536 MB of global memory each. The host uses an Intel Core i7 970 CPU at 3.2 GHz and 24 GB of main memory.
GTX580	The system hosts three NVIDIA GTX580 GPUs, each installed in a separate PCI Express $\times 16$ slot. The GPUs have 1536 MB of global memory each. The host uses an Intel Core i7 970 CPU at 3.2 GHz and 24 GB of main memory.
M2070	The system is connected to four NVIDIA M2070 GPUs through a single PCI Express $\times 16$ bridge. The GPUs have 6 GB of global memory each. The host uses an Intel Xeon X5675 at 3.07 GHz and 24 GB of main memory.
MPI	The measurements of the execution times for the MPI implementation are performed with two GPUs from each of the GTX480 and GTX580 systems. The systems are connected to the same 100 Mbit switch.

it operates at a lower clock speed, which explains the performance difference. On the up side, the larger memory capacity of the M2070 makes it possible to process systems of up to about 4.2 million neurons on a single device.

This system size can also be achieved with four GTX480 GPUs in a shared memory system. But not only can four GPUs process larger systems, they also scale well with a speed-up of 3.5 compared to the single GTX480 results for the largest system supported in both scenarios. Four M2070 GPUs even achieve a speed-up of 3.9 times compared to one of them by itself when processing the largest system supported by a single GPU. This is notable because the M2070s have to share a single PCI Express bridge to the host to transfer their border information, whereas the GTX480s each have their own PCI slot. However, this limitation does not go unnoticed, as four M2070s are significantly slower than a single device when processing smaller systems, where the border width of $r = 64$ makes up a larger fraction of the entire system and not much of the memory copy times can be hidden by overlapping them with the processing time of the inner core. This affects the M2070 much more dramatically than the GTX480. Thanks to the 6 GB of global memory, four M2070s are capable of processing system of over 16 million neurons with a total of over 1.6 billion neural connections.

The MPI implementation shows a significantly larger footprint than the shared memory algorithms, but it scales very well with the system size. It would likely perform better in an environment with distributed M2070s, which can process larger sub-systems on each individual device and therefore are better at hiding the time required to perform the border data exchanges over the network. The slopes for the least square linear fits to the data sets are given in Table 8.3.

Figure 8.8 shows the performance results when scaling the in-degree k while keeping the system size constant at $N = 2^{20}$. The maximum neighbour distance is again set to $r = 64$ unless explicitly marked otherwise. No results are given for the multi-GPU implementations, as they can only increase the degree at the expense of the sub-system size processed by individual devices. The results for the single GPU tests largely scale as expected. Only the GTX260 scales surprisingly better than the other devices, at least in the range of degrees that can be tested, as it is restricted by its relatively small amount of global memory. Even though the GTX580 and GTX480 are supposed to have the same amount of global memory, the GTX480 runs out of memory when $k = 260$, whereas the GTX580 completes the simulation successfully. This has been tested on multiple GTX480 and GTX580 devices with the same results. Simulations with a degree of $k = 240$ are successful on both devices. For the very large degrees of up to $k = 1200$ that are possible with

Table 8.3.: The slopes of the least square linear fits to the data sets on a logarithmic scale. No slope is given for the multi-GPU results of the M2070, as no good linear fit can be calculated for this data set. Only results for degrees equal to or larger than 140 are included in the calculation of the slopes for the results of scaling the degree to improve the linear fits, with the exception of the slope for the GTX260, which is computed for all data points. The first value for the M2070 states the slope for $r = 64$ and the second value the slope for $r = 96$.

Test	**GTX260**	**GTX480 x1**	**GTX480 x4**	**GTX580**	**M2070**	**MPI**
Scale N	1.00	0.94	0.93	0.90	0.98	0.35
Scale k	0.72	1.02	N/A	1.05	0.94/1.01	N/A

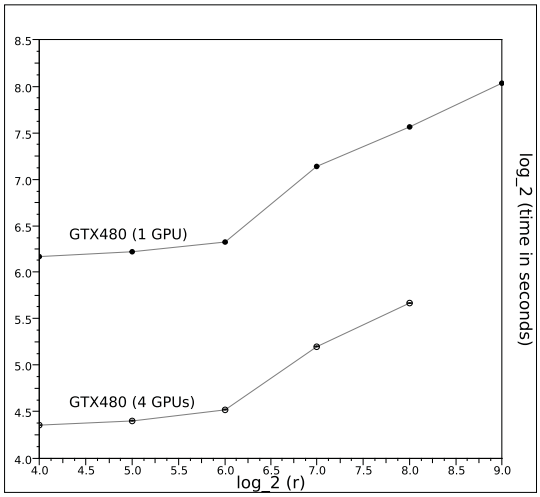


Figure 8.9.: The execution times for 10,000 simulation steps with the maximum neighbour distance ranging from $r = 16$ to $r = 512$. The system size is $N = 2^{20}$ and the in-degree $k = 100$. The multi-GPU implementation does not support $r = 512$, as each device only processes a sub-system with local dimension length $l = 512$ and r can be at most half the dimension length.

the M2070, the maximum neighbour distance is set to $r = 96$ to increase the size of the pool of neighbours to choose from. The gap between the two data sets for the M2070 indicates a significant performance impact when using large neighbour distances.

This is investigated in more detail in Figure 8.9, which shows exactly how much the chosen neighbour radius r affects the execution times in single and multi-GPU environments using the GTX480 system to demonstrate the effect. In both cases, there is a significant performance loss when r exceeds 64. This is most likely related to the size of the texture cache, which can not keep up with the requirements of the increased width of the neighbour distribution and therefore needs to repeatedly load certain memory address regions as requests for the entire area are processed. The cache usage can be improved by a factor of up to four by packing the data for 16 sequential neurons into a single 32-bit integer, two bits of data for every neuron. During `Phase1`, this integer value can then be updated in shared memory using `atomicOr` instructions before finally writing it to global memory. The atomic instruction works on 32-bit words, which is why it would not be enough to pack the data of four neurons into a 1-byte char. The implementation does not perform this optimisation, as the currently remaining 6 bits per neuron have been reserved for future extensions to the data shared between neighbours.

8.6. Processing the Pseudo-EEG Signal

The simulation generated pseudo-EEG signal provides some useful information about the state of the model. A quick glance is often enough to tell whether the model is producing some activity or if it is quiescent. For example, if not enough excitatory neurons are firing and the pseudo-EEG is consequently flat, then it may be necessary to tune the parameters to either increase excitatory inputs or reduce inhibitory inputs. In order to extract more information from the signal it is typically further processed. A number of common signal processing techniques typically applied to such a signal are described in this section.

8.6.1. Sampling an Analog EEG Signal

Firstly, it is necessary to put the pseudo-EEG signal produced by the simulation into context with the digitised EEG signal produced by digital EEG systems. Digital EEG systems sample or digitise an analog signal at regular intervals, called the sampling interval Δ. The sampling rate f_s is the reciprocal of the sampling interval and is usually expressed in hertz (Hz). The incoming signal should always be sampled at more than twice the highest expected frequency, as it takes at least two points within a single cycle to identify a sinusoid. If the sampling rate is too low, then the fastest sine waves in the measured time segment – or epoch – can not be identified correctly. The Nyquist critical frequency [187] $f_c = 1/(2\Delta)$ defines the peak frequency that can be sampled with a given sampling rate without the loss of any information. This means that if all frequencies in the signal are smaller than f_c, then the continuous function of the signal is completely determined by its samples. Frequencies outside the Nyquist frequency range cause aliasing. Aliasing distorts the resulting digital data, as the power spectral density (PSD) that lies outside of the frequency range $-f_c < f < f_c$ is moved into that range.

To reduce aliasing to a minimum, analogue low-pass and high-pass filters are often used before sampling to remove frequency components from the incoming signal that are outside the range $-f_c < f < f_c$ or more strictly outside the typical EEG signal range between 0.5 Hz at the low end to about $30-70$ Hz at the high end [188, 189]. Interference from other sources, for example power line interference at 50 or 60 Hz, is removed with a notch filter. Digital EEG systems often sample at a rate above 250 Hz to prevent aliasing distortion in the EEG [188]. As long as the signal is not corrupted by aliasing, further filters can easily be applied once the digital signal is transformed into the frequency domain.

The neural network simulation of the cortex used in this study runs at a resolution of one millisecond and thus produces a discrete pseudo-EEG signal with 1000 uniformly spaced samples per second of simulation time. In other words, the sampling interval $\Delta = 0.001$ and the sampling rate $f_s = 1000$Hz. Note that the simulation time is not directly related to real time, as it depends on the execution speed of the simulation.

8.6.2. Digital Signal Processing

A finite stretch of the signal can now be used to compute the discrete Fast Fourier Transform (FFT) and from this the power spectral density (PSD) estimate. The PSD estimate, which is also called the *periodogram*, describes how much power is contained in the frequency interval between f and $f+df$ [187]. Using the Fourier transform C_k, where $k = 0,\ldots,N-1$, of an N-point sample of the signal gives a periodogram estimate $P(f)$ of the power spectrum that is defined at $N/2+1$ frequencies in the range 0 to f_c, such that [190]:

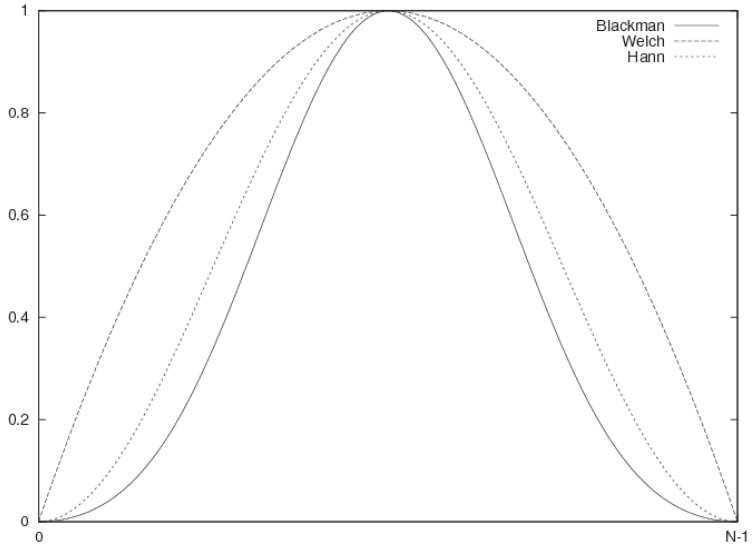


Figure 8.10.: The Blackman, Welch and Hann data windowing functions.

$$
\begin{aligned}
P(0) &= P(f_0) = \frac{1}{N^2}|C_0|^2 \\
P(f_k) &= \frac{1}{N^2}\left[|C_k|^2 + |C_{N-k}|^2\right] \qquad\qquad k = 1,2,\ldots,\left(\frac{N}{2}-1\right) \\
P(f_c) &= P(f_{N/2}) = \frac{1}{N^2}|C_{N/2}|^2
\end{aligned}
\tag{8.6}
$$

where f_k is only defined for the zero and positive frequencies

$$
f_k \equiv \frac{k}{N\Delta} = 2f_c\frac{k}{N} \qquad k = 0,1,\ldots,\frac{N}{2} \tag{8.7}
$$

The periodogram is normalised so that the sum of the $N/2+1$ values of P is equal to the mean squared amplitude of the original function $c(j)$ in the range of time $T = (N-1)\Delta$, as defined by [190]:

$$
\frac{1}{T}\int_0^T |c(t)|^2 dt \approx \frac{1}{N}\sum_{j=0}^{N-1}|c_j|^2 \tag{8.8}
$$

The Fourier transform assumes that the epoch represented by the N sampled data points represents what the continuous signal typically looks like at all other times. However, in reality, the EEG signal is sliced into arbitrary epochs. The signal is basically multiplied by a square window function, which has all values outside the epoch set to zero and all values within the epoch set to one. This causes distortions when the epoch of the time domain signal does not start and end with zero valued samples, because the Fourier transform adds artificial high frequency components to the frequency domain to match the abrupt, step-like transitions in the input signal [187].

A window function of N values that gradually tend towards zero at either end of the epoch, as opposed to the abrupt transition caused by the square window, can be used to minimise this type of distortion. Figure 8.10 illustrates three such window functions, although many more exist [190].

The N samples c_k in the epoch of the signal are multiplied with the same number of samples w_k of the window function of choice before the Fourier transform is performed, thus causing the epoch to begin and

end with samples that are close to zero-valued. For example, the Blackman windowing function is used in the calculation of the widely used bispectral index (BIS) [188], a quantitative EEG parameter (qEEG) that is based on a proprietary mixture of sub-parameters. The BIS is widely used to monitor anaesthetic effects on the brain.

However, data windowing creates another problem. It effectively throws away some valid data by deweighting the endpoints of the epoch. The standard deviation of the resulting PSD estimates is about 100% of the power value in the frequency bin [189, 190]. The accuracy can be increased by averaging the individual frequency bins of the PSDs calculated for multiple data segments of the same signal. Press et. al. [190] suggest that overlapping the data segments by one-half of their length gives the best results if one is limited by the number of data samples.

While the variance of the estimates gets smaller the more periodograms calculated for different data segments are used to compute the average, this only works as long as the input signal does not change. However, the signal generated by the cortical model is expected to change over time as the user modifies parameters to simulate drug effects. A balance between estimation quality and timeliness of the data is therefore needed. The current pseudo-EEG analysis tools that implement the signal processing methods discussed in this section average the ten most up to date PSDs to obtain the final periodogram. Each PSD is computed from $N = 1024$ discrete values generated by the simulation, although other powers of two work just as well. Because the overlap of the data segments is $N/2$, it takes 5120 simulation steps to completely remove the effects from a particular epoch from the averaged periodogram.

The following list gives a concise summary of the signal processing steps applied to the pseudo-EEG to obtain the periodogram estimate:

1. Compute the straight line fit to the current epoch of the input signal and subtract it from itself to center the data around zero. The data segment is of length $N/2 = 512$.
2. Overlap the new data segment with the second half of the previous segment to obtain a segment of length N.
3. Multiply the data segment with the chosen window function.
4. Compute the Fast Fourier Transform.
5. Normalise the output of the FFT to obtain the PSD estimate.
6. Average the new PSD estimate with the nine previous estimates.
7. Optionally apply a Wiener filter [190] to reduce high frequency noise. A straight line fit to the data set is used for the noise function.

8.7. Visual Analysis & Interactive Simulation Control

An interactive graphical frontend can be used as an alternative to the configuration file based batch processing. This is particularly useful when the user wants to get a feeling for the general behaviour of the model and how it reacts to certain changes. The direct feedback from such an interactive graphical tool is invaluable, especially in the early phases when the user has to tune the model parameters. The batch processing method, on the other hand, makes it easy to obtain results from a large number of simulation runs that may be necessary to improve the statistical accuracy of a qEEG or some other measured quantity.

Figure 8.11 gives an example of the output produced by the visual tool. The plot on the top illustrates the continuously updated pseudo-EEG generated during the previous second of simulated time. It can be expanded to show up to ten seconds of simulation time at once. While a more detailed view makes it

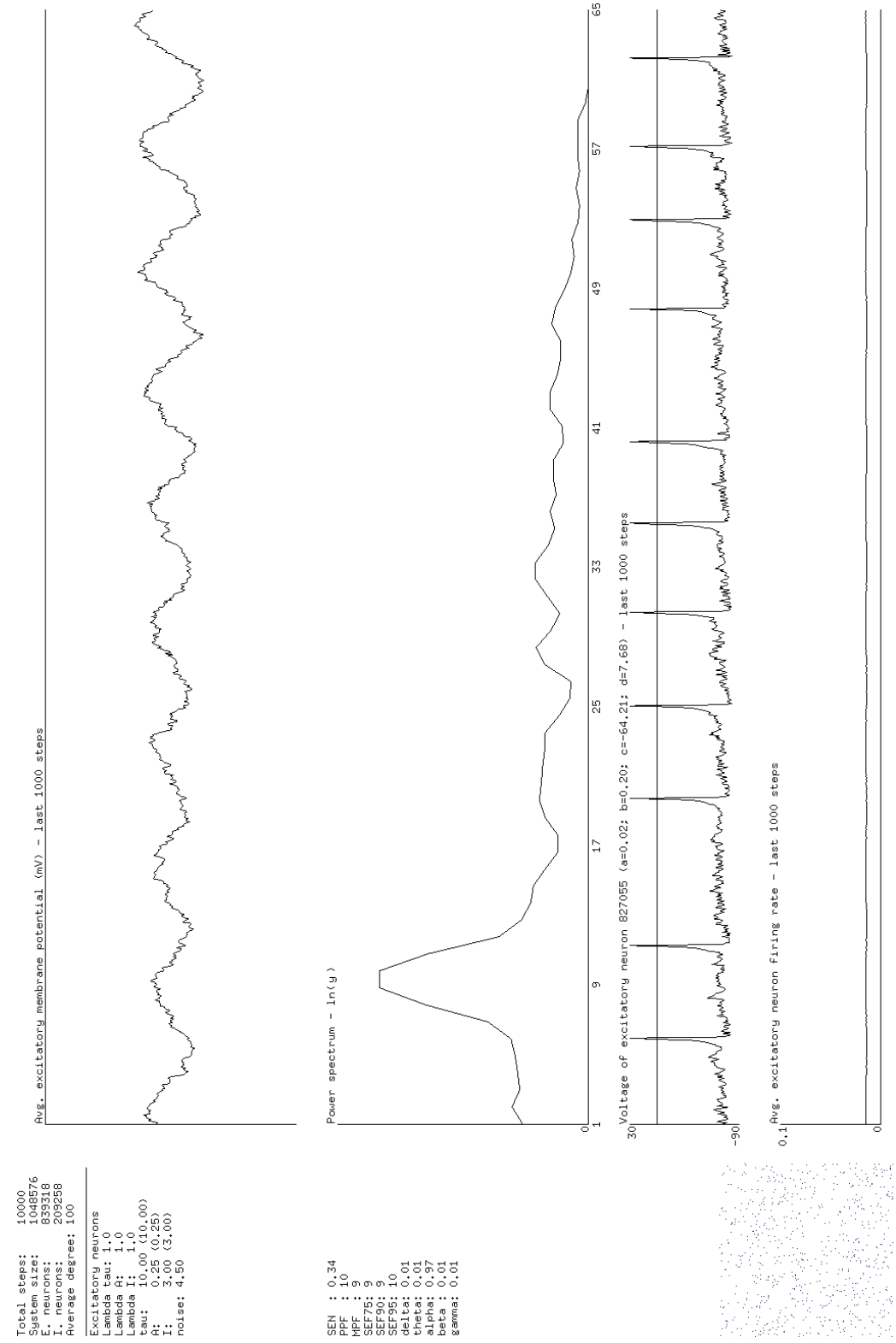


Figure 8.11.: The interactive visual analysis tool. The example shows a simulation for $N = 2^{20}$ neurons with in-degree $k = 100$ after $10,000$ simulation steps.

Table 8.4.: The frequency bands used by the analysis tool. The values are based on those listed by Rampil [188], except that the lower end of the range for δ is 0.0 instead of 0.5 and the upper end of the range for γ (called β_2 in [188]) is chosen to be the cutoff value for the periodogram, which is set to 65.0 Hz.

Band designation	Frequency range (Hz)
δ	< 3.5
θ	$3.5 - 7.0$
α	$7.0 - 13.0$
β	$13.0 - 30.0$
γ	30.0−cutoff

easier to see small changes in the signal, displaying several seconds of simulated time at once is often advantageous as it makes it possible to spot low frequency wave functions. The first plot in Figure 8.12 illustrates this by showing the ten seconds of pseudo-EEG generated during simulation steps 7000 to 17000. This includes the one second epoch visualised in Figure 8.11.

Below the pseudo-EEG is the periodogram on a semi-logarithmic scale. As discussed in the previous section, the ten most recent PSDs are averaged to obtain the results visualised here. The periodogram is updated every 512 simulation steps, using the new signal data to compute the current average. The user can switch between various data windowing functions, including the three functions illustrated in Figure 8.10. The Blackman windowing function is used in the example.

The third plot from the top illustrates the membrane potential for an individual, randomly chosen neuron. The neuron illustrated in the image is a regular spiking neuron, the most common type of excitatory cells used by the model. The global neuron ID and the values of the model parameters used for the selected cell are displayed above the plot. The user can switch to a different neuron to monitor its behaviour. Figure 8.12 gives examples for the firing rates of a fast spiking inhibitory neuron as well as those generated by chattering and intrinsically bursting excitatory neurons. The final plot on the bottom of Figure 8.11 illustrates the average excitatory firing rate over the time segment that is used for the pseudo-EEG.

The text in the top left lists some information about the configuration of the model. This includes the number of simulation steps performed so far, the total system size as well as the division into excitatory and inhibitory neurons and the in-degree per neuron. Below are the parameter values used for the excitatory or inhibitory neurons. The user can switch between displaying either of the two types. It lists the three λ values as well as parameters τ, A, I and noise n. For τ, A and I, the first value displayed is the base value, which is followed by the effective value with the corresponding drug effect λ taken into account in parenthesis. Both the base values and the values for the drug effects can be changed at runtime. This enables the user to test certain configurations and, by monitoring the plots in the main area of the interface, get an immediate feedback of the effects on the model.

To the left of the power spectrum are a number of frequency domain qEEG parameters. The first one reports the spectral entropy (SEN) [189]. It is computed for the frequency range $[f_s = 1, f_e = 47]$ of the power spectrum P, normalised to unity, such that $SEN = H(P)/\log(f_e - f_s + 1)$, where H is the Shannon entropy defined in Equation 2.3 on page 25. The peak power frequency (PPF), as the name suggests, reports the frequency with the highest power in that epoch. Similarly, the median power frequency (MPF) reports the frequency that bisects the spectrum, such that half the power is above and half the power is below. Likewise, the spectral edge frequency (SEF) reports the frequency below which a certain percentage of the power resides. The tool provides values for 75%, 90% and 95%. The SEF values thus provide some information about the width of the spectral distribution [188].

The final statistics report the relative power of the PSD estimates in different frequency bands. The fre-

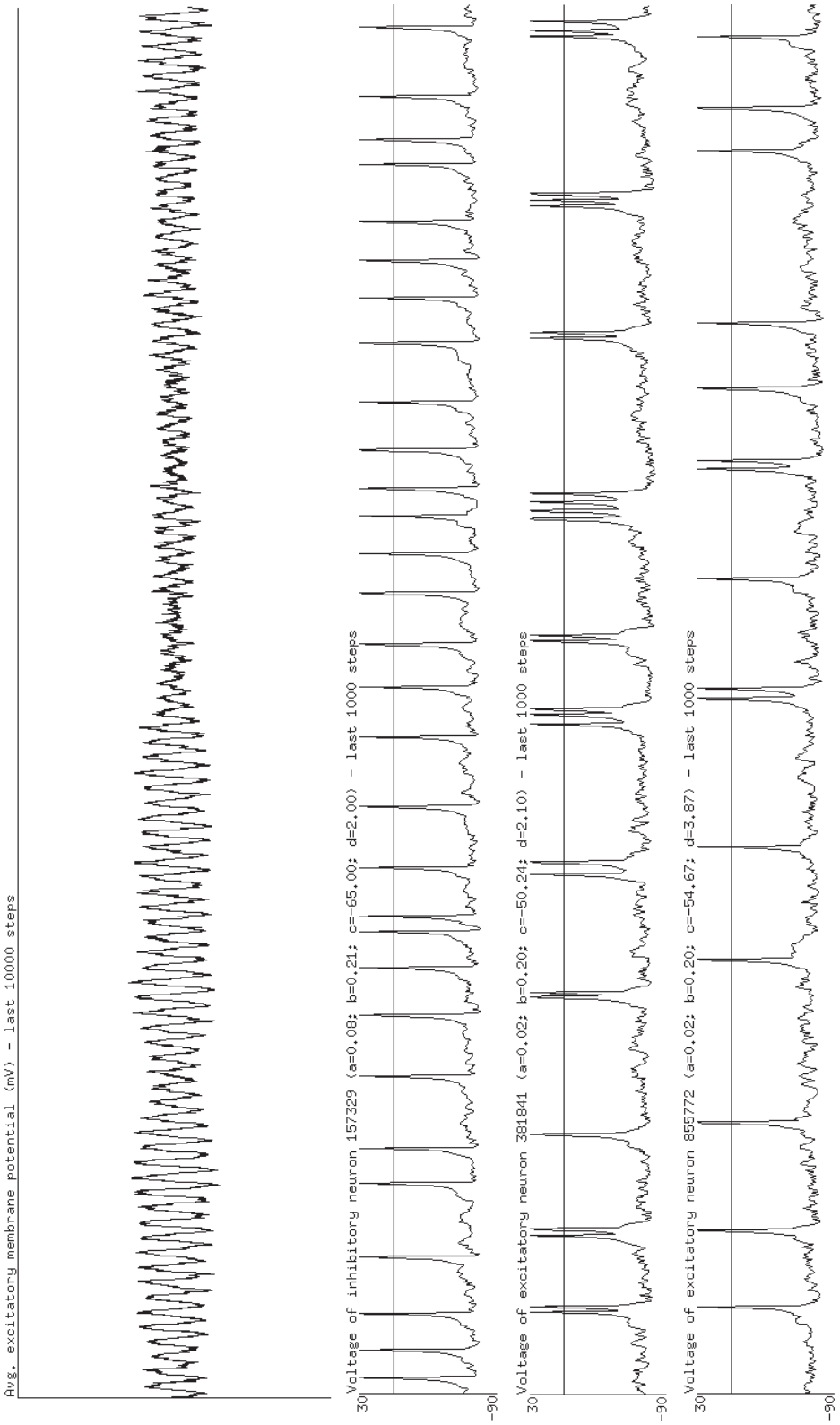


Figure 8.12.: The leftmost plot shows a ten second epoch of the pseudo-EEG. The other plots illustrate the firing rates of a fast spiking inhibitory neuron as well as those generated by chattering and intrinsically bursting excitatory neurons (left to right).

quency bands are typically called $\delta, \theta, \alpha, \beta$ and γ. Different definitions of the boundaries of each frequency band are used in the literature [188, 189]. Table 8.4 specifies the definition used in this implementation. These frequencies are rounded to the closest frequency bin. This is necessary because the PSD bins are not exactly equal to 1 Hz (or some other common denominator of the frequencies listed in the table), due to the fact that the FFT is computed for a data segment of length $N = 1024$. According to Equation 8.7, each bin represents a frequency range of ≈ 0.977 Hz. Due to the rounding, the sum of the reported relative powers can be slightly more or less than unity. There are a number of other qEEG metrics [188, 189], for example the burst suppression ratio and the bispectrum, that may be worth adding in the future.

The area in the bottom left corner visually indicates which neurons are firing in any given simulation step for a region that comprises 200×200 cells. The user can switch to a view of the entire system, which hides the plots and takes up most of the interface area. While it is difficult to see anything else than noise in a static image, this view is very useful to detect waves of activity in the model at runtime. The last feature, which is not shown in the image as it also requires most of the interface area, allows the user to compute and visualise the 2D FFT of either the membrane potentials or the firing state of the entire grid of individual cells in the system. This can be useful to detect certain patterns of neural activity.

8.8. Discussion

This chapter describes the CUDA implementation of a spiking neural network model based on Izhikevich type neurons. The model is designed with the intention to simulate and analyse the neural processes involved in anaesthesia. The algorithmic approaches proposed here make it possible to perform large scale simulations with neural networks consisting of over one million neurons and hundreds of millions of neural connections on a single commodity graphics accelerator. Professional workstation or data center products, like the Tesla M2070, can even cope with system sizes of over four million neurons or more than 1.2 billion neural connections on a single device. Even larger models can be simulated using the shared memory or message passing based multi-GPU algorithms discussed in Section 8.4.

Although the computational foundations are in place, the model still needs to be fine-tuned to produce more realistic pseudo-EEG patterns. Bennett et. al. [191] describe the wave forms and patterns observed in real EEG data during different phases of wakefulness and anaesthesia. Getting the model to reproduce these activity patterns has proven to be complicated and computationally expensive, as the parameter space is relatively large and some of the parameters are very sensitive to even small changes, producing unexpected and often unwanted effects. Many of the system parameters are also correlated with each other, making it more difficult to find the right values for one particular parameter. The interactive analysis tool described in the previous section can be used to test promising combinations more thoroughly or to quickly narrow the parameter space down to a smaller range of values. The batch mode can be used to automatically test a large number of combinations of various parameters and value ranges, making very fine grained changes to individual values. It is usually used to extract certain information in form of quantitative EEG metrics that can then be plotted or otherwise visualised by the user.

The implementation of the model demonstrates how certain design decisions can help to optimise an inherently irregular system for the data-parallel architecture of today's graphics accelerators. In particular, the use of a pull model instead of a push model for the transfer of action potentials along neural pathways, coupled with a uniform in-degree and the specification of the conduction delays on the receiving cells, makes it possible to fully coalesce the memory transfers in one of the most critical sections of the algorithm, thereby dramatically improving the performance of the simulation on the GPU. The data layouts used by the implementation are designed to achieve high bandwidth memory access, reducing the number of times sequential threads do not access subsequent addresses in global memory to the absolute minimum. The algorithms make use of all types of memory available on the latest generation of NVIDIA devices, most

importantly the fast on-chip constant and texture caches, the automatic L1/L2 caches as well as the shared memory space. It is demonstrated how the different caches can best be utilised for a given task in order to reduce transfers to and from global memory and thus improve the performance of the simulation.

The model is an ongoing project and many additions are possible to further improve its realism or simply to test how they affect its dynamics. It would, for example, be interesting to see how a move to a three dimensional layout of neurons changes the pseudo-EEG and related measures. Other data structures, in particular a more small-world like structure with densely connected local clusters that are only weakly connected to other clusters, may also be able to produce some interesting firing patterns and propagation behaviour. The delay, which is currently assigned at random value in the range from 5 to 15ms, could instead be based on the distance on the grid structure that separates the two connected neurons. And parameter I, which is used to model currents from external sources, could be replaced by inputs from a simulated model of the thalamus that uses cells with different spike patterns, which would then have to be driven by some sort of external, random input similar to the way it is currently done for the cortex. While many such additions are possible, they tend to make the model more complicated and in turn more difficult to analyse. The more parameters are introduced, the more possible combinations need to be tested in order to produce a system that behaves realistically enough to be of any value. An increase in the complexity of a system also tends to come at an increased computational cost, which in turn means the system size has to be reduced to retain the same level of performance or to be able to fit all required data into memory. It is therefore important to find a balance between complexity and computational cost and the author believes that the current implementation does exactly that.

Part IV.

Discussion & Conclusions

Chapter 9

Discussion

The single instruction, multiple data processor architecture is becoming increasingly important for general purpose computing on commodity hardware. Historically mostly used in supercomputers, the growing demand for highly data parallel tasks like multimedia and graphics processing saw to it that CPU manufacturers began to integrate vector extensions into their consumer products. Dedicated graphics accelerators soon appeared to keep up with the constantly increasing requirements of the video game and computer graphics industries. With the slow down of the race to higher processor clocks rates in recent years, CPU manufacturers have started to move towards more parallel architectures. While they have been busy increasing both the number of independent cores as well as the data parallel processing capabilities of their chips, graphics accelerators have received a continuously growing amount of attention from the high performance computing community. With their wide vector units, GPUs achieve a much higher peak floating point performance than traditional x86 CPU designs. And thanks to the notable efforts that GPU manufacturers have put into making these devices more flexible and their capabilities more accessible to developers, graphics accelerators have become a force to be reckoned with for general purpose computing tasks.

9.1. Parallel Processing Architectures

CPU manufacturers are not only increasing the number of scalar cores in their processors, but they are also extending the vector processing capabilities of their chips. The latest generation of Intel processors, codenamed *Sandy Bridge*, integrates the advanced vector extensions (AVX), which extend the streaming SIMD extensions (SSE) instruction set and offer a 256 bits wide SIMD register file. AMD is also expected to introduce AVX into its upcoming processor generation. While these vector units can be used to improve the performance of some floating point intensive calculations, the manufacturers are already taking the next step to integrate the processing architectures of CPUs and GPUs more tightly.

Both Intel and AMD have integrated extended graphics capabilities into the silicon of their latest generation CPU designs. Intel's Sandy Bridge architecture processors share the last-level cache (LLC) between the scalar CPU cores and the GPU vector arrays [192]. In the first generation of AMD's *Fusion* processors, the GPU has to flush its cache to memory before the CPU can access shared data. In the other direction, however, the GPU can either read uncacheable data from system memory using a non-coherent high throughput bus, or it can use a coherent bus to snoop the CPU cache [193]. It can be expected that the trend to combine scalar and vector processing capabilities for both graphics and general purpose computing tasks into a single chip will continue and that the integration will become tighter so that context switches can be performed more efficiently.

The growing momentum of GPUs as general purpose compute accelerators is reflected in the most recent TOP500 supercomputer list from June 2011 [22], in which three of the top 10 systems alone integrate NVIDIA Tesla modules. The ranking is based on the performance in the LINPACK benchmark [194], which uses a dense system of linear equations to measure the 64-bit floating point computing power. Based on the TOP500, the Green500 [195] ranks systems by how energy efficient they are and tries to raise awareness about power consumption, which is another important aspect of supercomputers. The performance is therefore measured in FLOPS per watt. The latest Green500 list from June 2011 places two systems with ATI/AMD Radeon GPUs and two systems with NVIDIA Tesla GPUs in the top 10. Another five Tesla based systems can be found on ranks 11 to 20. The Graph500 [27] supercomputer list, on the other hand, focuses on data intensive applications and uses a set of graph algorithms to compare the performance of different systems. Although this is a relatively new approach to complementing the TOP500 list and still under development, it emphasises the importance of graph algorithms for high performance applications. The processing power is measured in traversed edges per second (TEPS). Unfortunately, the Graph500 list does not provide detailed information about the configuration of individual systems. Nevertheless, the author is aware that the system ranked number three on the June 2011 list, which achieved the highest absolute performance of about 43.5 billion TEPS, is configured using a combination of Intel Xeon CPUs and NVIDIA Tesla GPUs. The same system is placed on rank 13 in the latest TOP500 list.

But not all forthcoming processor designs merge CPU and GPU architectures. The first product from Intel's many integrated core architecture (MIC) [192], codenamed *Knights Corner*, is said to scale to more than 50 single threaded, low power processing cores with 512 bits wide vector units on a single chip. While the press release for the MIC co-processor emphasises the ease of migrating existing code for Xeon processors to the accelerator and vice versa, thanks to the use of the same programming models and tools, it remains to be seen if it will really take less time to fully optimise code for the MIC architecture than it does for the GPU. Although both architectures rely on wide vector units for a significant part of their compute capabilities, the MIC co-processor can fall back to the 50 low power scalar processing cores for tasks that are not data parallel. Another advantage for Knights Corner may be that it is not burdened with any features that are not directly aimed at raw general purpose compute throughput. It may also have an edge due to the use of the 22nm manufacturing process, assuming that Intel can bring it to market at a competitive price before other manufacturers are able to switch to the same process. However, without details about the final clock rates and memory bandwidth used by Knights Corner and scarcely any information about the next product generations from NVIDIA and AMD, the only thing that seems certain is that data parallel processing is bound to become even more important.

9.2. Data Parallel Programming Models

Even though CPU manufacturers have integrated vector extensions into their chips for many years, most applications do not explicitly take advantage of these compute capabilities. Compilers can only do so much to parallelise and vectorise code and many opportunities for parallel processing already go unused. Only a relatively small number of applications, in particular multimedia software or specially optimised code for scientific high performance computing tasks, really utilise modern processors. This needs to change if applications are to keep up with the way hardware is evolving. Programming models and tools that enable the developer to specify parallelism at a very fine grained level are needed.

9.2.1. The Compute Unified Device Architecture

CUDA is used to refer to both the architecture and the programming model, but it is worth looking at them separately. While a group of threads begins execution at the same program address in the usual SIMD

fashion, the hardware can disable individual threads if data-dependent branches force them to take different paths. All execution paths taken by the threads in the same warp are serialised and threads are activated and deactivated as necessary. NVIDIA refers to this architecture as single instruction, multiple thread (SIMT), because it makes it possible to express the code in terms of scalar threads instead of the vector instructions used by traditional SIMD architectures.

The developer is not forced to use intrinsic vector instructions for a particular SIMD device, but instead the code is expressed in terms of the execution paths that each individual thread may take under certain conditions. This does not mean that the developer can disregard the underlying architecture, as substantial performance improvements can be achieved when the execution path of a warp does not diverge, but the correctness of the outcome is guaranteed. The SIMT model makes it easier to focus on the actual algorithmic challenges and to avoid getting bogged down trying to express a simple conditional statement with multiple intrinsic instructions. This case is demonstrated in the CellBE implementation of the clustering algorithm in Section 6.2.3, where three intrinsic instructions are needed just to conditionally increment a counter on a vector of four elements. The way memory latency hiding is done is another example for the simplicity of the CUDA model, although it is more of a hardware feature that makes the difference rather than a software functionality. When developing for the CellBE, it is up to the developer to explicitly manage the asynchronous data transfers between system memory and local memory. CUDA devices, on the other hand, automatically switch between warps that are waiting for data to warps that are ready to run.

With the fine grained level of parallelism found in CUDA, the developer is forced to break the problem down to its most basic elements. While this requires a different way of thinking than the development of serial code, it is fundamental to the idea of spreading out work over any highly parallel architecture. The ability to manage thousands of threads concurrently without any significant overhead is particularly useful for memory bound kernels, because it allows the GPU to switch between threads that are waiting for some memory and threads that are ready to proceed, thus very effectively hiding memory latency. While the PCI Express bandwidth can be a bottleneck, this is only really a problem when the computational intensity, that is the number of operations performed on each data element, is very low. The ability to overlap memory transfers with computation can at least partially hide those transfers in many situations. And once the data is on the device, the GPU has a significantly higher bandwidth to device memory than the CPU does to system memory, at least as long as over-fetch can be minimised by coalescing memory transactions.

The focus on thread level parallelism makes the programming model more flexible. And while CUDA is a proprietary language, the code is not necessarily restricted to one particular architecture. It can be compiled for scalar and SIMD processors alike. If a vector processor is not able to conditionally disable individual threads and automatically serialise diverging execution paths, then the compiler should be able to add intrinsic instructions or use predicated execution to mask the results so that they are only active for the desired threads. However, this usually comes at a performance cost if there are more than a few instructions in the conditional statement. The increasingly parallel nature of most processors makes the need for such a new programming model with focus on large scale, fine grained parallelism apparent. Most new processors also integrate a combination of scalar processing cores and vector units, making it all the more important to use a model that can express parallelism in such a way that it can be mapped to all suitable processing circuits available on the chip. The CUDA programming model is a significant step into this direction. Although it is so far mainly used to develop applications for NVIDIA GPUs, efforts for compilers that generate code for CPUs that are based on the x86-architecture with SIMD extensions like SSE and AVX are already underway [196].

9.2.2. The Open Computing Language

The Open Computing Language (OpenCL) [197] is an open standard for parallel, cross-platform computing on heterogeneous systems. It defines a low-level abstraction to create portable applications. Originally created by Apple, OpenCL is now being developed by the Khronos Group with support from a large number of major companies, including AMD, Intel, NVIDIA, IBM, ARM and Imagination Technologies just to name a few that are particularly relevant, as they offer implementations of the standard for their respective hardware products. OpenCL offers both task and data level parallelism.

In the OpenCL platform model, each CPU or GPU is called a compute device. A compute device has one or more compute units – scalar cores on the CPU and streaming multiprocessors on a CUDA GPU – which in turn are made up of one or more processing elements. Compute kernels in OpenCL are written in a C-like language and are conceptually very similar to CUDA kernels. The problem space is divided into so called work-groups, which are equivalent to CUDA thread blocks and are always processed on the same compute unit. Each work-group consists of work-items that can share memory and be synchronised with other work-items in the same group. OpenCL defines private, local, constant and global memory, which correspond to registers & local memory, shared memory, constant memory and global memory in CUDA terminology. One or more command queues, which can be in-order or out-of-order, are used to assign kernels to a particular device. The OpenCL event model is used to coordinate the execution of commands between the host and compute devices.

With the strong support from the industry, OpenCL is one of the most promising frameworks for portable, cross-platform, high-performance applications. Its data parallel programming model for GPUs is closely related to CUDA, and the additional support of other data and even task parallel architectures is intriguing. However, being portable does not mean that the same code performs equally well on different systems. Significant performance improvements can be achieved by optimising the OpenCL code for a particular device. The variety of on-chip cache and shared memory types and sizes, memory access restrictions and processing unit features between devices from different manufacturers and even among product generations from the same company make it necessary to tune the code for the target architecture if high performance is desired. For this, a number of extensions for widely available but optional features are defined in the OpenCL specification [197], and manufacturers can also define their own extensions.

However, some features that are available to developers who are using CUDA are not yet accessible in OpenCL even through vendor extensions. This includes useful C++ features like templates, namespaces, operator overloading and classes (with some restrictions) and also more general features like recursion and device function pointers (from device code only). But it also includes certain capabilities that can directly affect the performance of the application. For example, GPUDirect v2.0, which has been introduced with CUDA 4.0 [198], enables peer-to-peer communication between GPUs over the PCI Express bus and thus makes it possible to completely bypass host memory when sharing data in a multi-GPU application. CUDA is arguably also more developer friendly and mature, as it is focused on GPU development for a particular architecture and not burdened by features that are necessary to make the code portable across a variety of compute devices. Generally speaking, OpenCL has the upper hand when it comes to portability and it is an important framework for any application that targets a broad range of devices, but CUDA supports the latest hardware and software features available for NVIDIA GPUs. With the focus on high performance, data intensive graph algorithms and no particular need for portability, CUDA is the obvious choice for this thesis and most other applications that can exclusively target NVIDIA devices.

9.2.3. Other Programming Models

Intel Array Building Blocks (ArBB) [199] is a dynamic library that offers an embedded language and compilation framework for C++. It targets both multi-core and heterogeneous many-core architectures

and supports shared and distributed memory models. ArBB is compatible with standard C++ compilers, including the Intel C++ compiler (ICC) and the GCC. It offers an abstraction from the actual hardware with the aim to simplify the way developers can express task and data level parallelism for a wide range of heterogeneous architectures. ArBB is meant to be easy and safe to use. It helps to prevent deadlocks and race conditions by means of a serial, deterministic ordering of the results of all computations and the use of by-value semantics. A new collection is created whenever an existing collection is modified by an expression. While this approach may simplify the code and work for some mainstream applications, it has the potential to significantly increase the memory footprint of the program.

It also seems questionable whether the entire approach can produce results that get at least close to the execution speed of an optimised high performance application, especially when it is data intensive. For many-core architectures, the memory interface tends to become the bottleneck and significant performance gains can be achieved when the algorithm fully utilises fast on-chip memory for data sharing and reuse. Even though ArBB uses dynamic code generation and compilation to optimise code for the target machine, the outcome still depends on the compiler's ability to recognise such opportunities. Nevertheless, the approach of an embedded language that targets heterogeneous parallel architectures with runtime code generation is interesting and it will be worthwhile to keep track of future developments.

The Intel SPMD Program Compiler (ispc) [200] is an open source compiler for a C-like language that can be used to write task and data parallel code for x86 CPUs. It uses the LLVM [201] compiler infrastructure for its back-end. The ispc implements the single program, multiple data (SPMD) paradigm, where a number of instances of a mostly serial program are executed in parallel. Ispc creates a thin abstraction layer over the hardware to remove the need for writing intrinsic instructions to access the SIMD units. The ispc generates regular C/C++ object files and supports sharing data without reformatting directly via pointers. SPMD functions can be called using standard calling conventions. This is an attractive approach due to its simplicity. The downside is that the ispc is limited to x86 CPUs only.

Chapter 10

Conclusions & Future Work

Processor architectures are undoubtedly moving towards an increased level of parallelism. The number of scalar processing cores is increasing and SIMD units for vector processing are becoming wider and more capable. GPUs in particular have evolved into highly parallel compute powerhouses. Initially very specialised for graphics acceleration, their architectures are becoming more flexible with every generation. New programming toolkits have emerged to expose these capabilities to developers so as to be used for general purpose computing tasks. By doing so, GPUs are increasingly competing with CPUs in the high performance market. This thesis describes how well the data parallel architecture of modern graphics accelerators copes with complex, irregular graph structures, which are at the core of many scientific simulation models.

10.1. The GPU as Compute Accelerator

The high peak performance of today's GPUs, relative to the CPU, makes them a very exciting alternative for certain compute tasks. To avoid disappointment after a lot of effort has been invested into the development of a GPU accelerated application, it is essential to get a feeling for the types of algorithms that not only parallelise well, but that parallelise well on the graphics architecture. Not every algorithm is suited for execution on the GPU and the generally higher complexity of writing a GPU implementation, compared to a sequential or even multi-threaded CPU algorithm, may not seem to pay off if the speed-up is only moderate. On the other hand, even a single digit speed-up of a certain algorithm may be worth every effort if it is used in a critical part of the application and gets called repeatedly. After all, it makes a big difference if the tensely awaited results of an important simulation are available after a week or after two months. On the other end of the scale, there are algorithms that perfectly fit the data parallel architecture of the GPU and reward the developer with a performance that is several orders of magnitude higher than what can be achieved on the CPU.

The graph generator algorithms discussed in Chapter 5 demonstrate a scenario where the significantly more complex implementations for the GPU struggle to keep up and at least get a small speed-up compared to the multi-threaded CPU implementations. Especially the implementation of the generator for scale-free graphs suffers from the fact that the algorithm is not very compute intensive to begin with, but instead puts high demands on the memory management, making it necessary to repeatedly re-allocate adjacency-list structures and copy data from one array to another. These are tasks that do not benefit much from parallel processing cores, as they are limited by the memory bandwidth of the device. And sure enough, not only the CUDA algorithm struggles, the TBB implementation for multi-core CPUs only achieves speed-ups of around a factor of two on a six-core processor compared to the sequential implementation. This

can generally be taken as clear sign for an algorithm that will not parallelise well on the more restrictive architecture of the GPU.

That said, the GPU is still able to outperform the sequential implementations in all test cases and the multi-threaded implementation at least in most of the tests. Assuming that the graph generation is not the overall objective, but that the data structure is to be further used by other algorithms on the same device, then running the generator on the graphics accelerator may have the additional advantage of not having to send the data structure over the slow PCI Express bus to transfer it from host to device memory. While this is not time saved by the generator itself, it reduces copy times and therefore the overall execution time of the entire process. Generally speaking, if there are additional benefits from running an algorithm on the graphics accelerator, then the savings in the total processing time may be significantly higher than just the performance difference between a single CPU and GPU algorithm.

Sometimes there are different algorithmic solutions available for the same task. The Metropolis and Wolff update algorithms discussed in Chapter 7 are an example for such a scenario. Even though they work very differently from each other, both can be used to evolve the Ising model simulation and obtain the same results for properties like the system energy and magnetisation once the model has reached its equilibrium configuration. The phase transition from the random to the ordered regime can also be observed with either one of the update algorithms. While the Wolff cluster update algorithm with its non-local update dynamics is able to decorrelate a system near the critical temperature in fewer simulation steps, the Metropolis algorithm is more easily parallelised. The experimental results show that the better match of the Metropolis algorithm to the data parallel graphics hardware makes it the superior choice for this architecture, even when it requires a larger number of update steps to decorrelate the system to the same degree. This example shows that it is not only essential to optimise an algorithm for the hardware that it runs on, but to make the best algorithmic choice based on the particular strengths and weaknesses of the architecture. For a highly data parallel device like the GPU, an algorithm that can be implemented with few conditional branches and little unpredictable memory accesses is likely a better choice than an algorithm with fewer instructions that actually need to be processed, but where the chosen execution path depends on some data value and regularly diverges from one element to the next.

Even better than just having the choice between different algorithmic solutions for a given problem is to be able to make some of the design decisions with the hardware architecture in mind. Depending on the project, it is sometimes possible to make certain choices that do not have a negative effect on the outcome, or where the effects are at least tolerable, but which may have a significant influence on how well the implementation performs on a particular device. One such example is the network structure of the cortical model. As described in Chapter 8, all neurons in the model have the same in-degree, but varying out-degrees. Together with the use of an information pull approach instead of a push approach, the implementation can use this fact to fully coalesce the look-up of the cell IDs of neighbouring neurons and thus maximise the usage of the available memory bandwidth in one of the two most performance critical code blocks of the simulation. Similarly, the decision to model the conduction delays on the receiving neurons and not on each individual link improves the execution time of the second critical section, as it makes it possible to coalesce the global memory writes to the circular buffer that records the incoming action potentials for every neuron. Like the degree, the delay is thus always the same for incoming connections but varies for outgoing connections.

While it is often most interesting to compare the performance of a particular algorithm optimised for the GPU to the same algorithm optimised for the CPU, sometimes it is more important that by processing one task on the graphics device, the main processor is able to work on a different task altogether. Most mainstream computers and workstations have some sort of graphics accelerator, be it integrated into the CPU, soldered onto the mainboard or a dedicated device plugged into a PCI Express slot. But unless the user is running a graphics intensive application, the compute capabilities of these circuits remain mostly

unused. At times like that, it makes sense to utilise the dormant processing units of the GPU for other tasks. Even algorithms that only achieve little to no speed-ups over a multi-threaded CPU implementation are sometimes better executed on the graphics accelerator, leaving the CPU cores almost entirely available for other tasks and thus improving the overall compute throughput.

A hybrid solution that splits a task between the GPU and CPU can also reduce the execution time, assuming that the algorithm does not require a lot of communication between the parts processed by the host and those processed by the graphics accelerator or, alternatively, that the data transfer times can be overlapped with computation. This is demonstrated by the multi-GPU implementation of the neural network simulation in Section 8.4. While it is used to efficiently utilise more than one GPU in this example, the concept is the same for a hybrid CPU/GPU approach and the performance depends even more on the ability to hide the memory transfers with computation, because the data does not only have to be copied from graphics to host memory, but in a second step it also has to be copied to the device memory of the other graphics devices. The implementation shows how – with detailed knowledge about the data structure and data exchange requirements – the system can be partitioned into two parts, one that depends on data from other devices and one that does not. Tests comparing a single GPU to four devices of the same type, all installed in a single host system, demonstrate that the algorithm is able to effectively hide most or all of the transfer times, completing the simulation run about 3.5 to 3.9 times more quickly when using the multi-GPU setup, depending on the system size and the width of the border region that needs to be copied between devices.

One feature that is specific to the GPU as compute accelerator is the ability to interface with graphics libraries like OpenGL or Direct3D and to update the data used for rendering directly on the device. This avoids memory copies over the PCI Express bus and enables the developer to utilise the processing units of the accelerator to compute vertex coordinates and other associated information. Section 7.3 describes how this feature is used to quickly update vertex colour values and by doing so visualise the individual states of all spins in the Ising model. This interoperability is essential when the GPU is to be used for both general purpose computation and rendering. Without it, data would not only need to be transferred to the graphics device as it is usually done to render a scene, but the data would first have to be copied from the compute accelerator to the host, thus effectively doubling the amount of information that needs to be moved over the relatively slow bus. It is a powerful feature for any application that visualises information based on data in graphics memory.

10.2. Irregular Graph Structures on a Data Parallel Architecture

There are a few obstacles that typically need to be addressed when implementing a graph based algorithm on the GPU. The first one has to do with the representation of the graph structure in device memory. One of the most important aspects of code optimisation for the graphics device is to reduce transfers to and from device memory. This can be achieved either by the clever use of fast on-chip memory – assuming that data is being re-used or threads need to exchange information – or by substituting memory look-ups with computation. With the large number of processing units available on today's GPUs, computation is cheap compared to memory bandwidth. And if the graph represents a grid or similarly regular structure, then it is often not necessary to store information about neighbours explicitly. Instead, the adjacency-list for a vertex can be inferred from its own position in the graph. The kernel for unmodified cells in the Ising model, which is discussed in Section 7.1.3, demonstrates how this works even when the graph structure is not entirely regular.

When it is not possible to compute the adjacency-list information for individual vertices or for the entire graph, then it should be stored in memory in such a way that the memory bandwidth requirements are minimised when reading the data. For CUDA devices this specifically means that memory requests issued

by multiple threads need to be coalesced whenever possible to avoid over-fetch. When each vertex has the same number of neighbours, then this can be achieved by interleaving the memory access from sequential threads and using a stride of N between elements in each thread's adjacency-list, where N is the system size and any padding necessary to correctly align it to the next segment in device memory. The exact alignment requirements depend on the device and configuration, and while 32 and 64 byte alignment can be sufficient, alignment to 128 byte segments is the safe bet for all currently available CUDA devices. The simulation of the cortical model uses the interleaved adjacency-list structure as discussed in Section 8.2. However, when vertex degrees are not uniformly the same, then this access pattern wastes memory, as it requires that every adjacency-list is padded to the length of the longest of the lists. While this may be feasible if degrees only vary very slightly, it is not feasible when dealing with complex graph structures where the degrees vary significantly.

If this is the case, then a different memory layout has to be used. Which one gives the best performance depends on the graph structure and on the way the algorithm iterates over adjacency-lists. Several different approaches are discussed throughout this thesis. One of the component labelling kernels described in Section 6.1 uses an approach that assumes that vertex degrees do not vary drastically, but too much to use the memory layout discussed above. The adjacency-lists of sequential vertices are concatenated and stored in a single arc array. Every vertex is mapped to a thread and the start index of its adjacency-list in the arc array is recorded in a second array so that sequential threads load subsequent elements from memory. The indices can be loaded using coalesced memory transactions and the vertex degrees can be determined from the indices loaded by the following threads, making use of shared memory to share this information. Although the adjacency-lists are not read using coalesced memory transfers, the situation can be somewhat improved by the use of 1D texture fetches for these transactions. This utilises the texture cache to reduce the impact of over-fetches, making use of the knowledge that the next element in the adjacency-list will soon be needed by the same thread. On Fermi-architecture devices, the L2 and L1 caches do this automatically and the use of texture fetches is not necessary.

However, the memory transactions are not the only problem in such an algorithm. When each thread iterates over the adjacency-list of a particular vertex, then this leads to warp divergence when the lists processed by threads in the same warp are of different lengths. It is possible to minimise the amount of divergence by sorting the vertices by their degrees. This approach can reduce the kernel execution time when the algorithm has to iterate over significantly varying adjacency-lists, but the overhead introduced by sorting the vertices can easily be larger than the savings. Algorithms that map threads to vertices based on their degrees are discussed in Sections 6.1 and 6.2, but do not lead to an overall performance improvement in those situations. This optimisation strategy is most likely to be of success if the algorithm repeatedly iterates over the same graph structure.

If the adjacency-lists vary significantly in length and the algorithm mainly processes the interactions between pairs of vertices, without much processing time spent on each individual vertex, then it may be better to change from an approach that assigns threads to vertices to an approach that assigns threads to edges or arcs. This is demonstrated by one of the implementations of the component labelling algorithm in Section 6.1. Instead of storing adjacency-lists in memory, the graph structure is directly expressed in terms of the edges that connect the nodes of the graph. Every edge records the two vertices that it connects and sequential threads process edges stored at subsequent memory addresses in global memory, thus making it possible to fully coalesce the look-up of neighbour information. This also eliminates any warp divergence from iterating over adjacency-lists of varying lengths. However, generally it is necessary to load secondary information about the vertices, which would likely be stored in the order of the vertex IDs and may lead to uncoalesced memory transactions.

The implementation that gives the best results often depends on the graph structure. If it is known that there is some regularity, then this should be exploited to reduce the number of device memory transactions.

When the adjacency-lists can be arranged so that memory access is coalesced without being overly wasteful with the available memory, then this should be done. And if vertex degrees follow a power law, then it may be best to sort the vertices by their degrees or to express the graph structure in terms of the connections instead of the nodes. When it is not known what type of graph has to be processed and the algorithm is time consuming, then it may be best to attempt to analyse the graph structure before the implementation used to process it is chosen. Even a quick scan over a randomly selected subset of the graph's vertices may provide enough information about the degree distribution and other properties of interest to make a more informed choice.

Graph based algorithms tend to put a lot of strain on the memory bandwidth. Not only the neighbours structure, but also properties of the vertices and connections have to be loaded from device memory and updated values have to be written back to it. Every request that can be served by one of the fast on-chip caches or from shared memory instead of having to transfer the data from device memory has therefore the potential to significantly improve the performance of the algorithm.

The main objective of this thesis is to investigate whether the highly data-parallel architecture of modern GPUs can achieve good performance for scientific simulations based on complex networks with irregular data structures. Such networks are at the core of many scientific simulations, in particular when the model under investigation attempts to recreate a phenomena or behaviour observed in a natural system. Based on the experiences collected throughout this thesis, the author is confident that the data-parallel architecture of today's GPUs can be utilised to accelerate a wide variety of applications. As long as the runtime of the project is long enough to justify the development of a highly optimised algorithm for a particular hardware architecture, the GPU should definitely be considered.

While there are certainly algorithms that are sequential in nature and do not parallelise at all and other algorithms that are not suited for the data-parallel paradigm of the graphics accelerator, many problems that do not appear to be a good match for the GPU can be adapted so that they can efficiently use the processing resources available. Many irregular graph problems fall into the last category. With the exception of a few graph structures that are very serial in nature, all algorithms discussed in this thesis perform well enough on the GPU to at least use it as an accelerator when available and by doing so leave the CPU free to process other tasks or to split the work between the different processing units and thus improve the overall performance of the system. Other algorithms, like the simulation of the rewired Ising model in Chapter 7, perform so well on the graphics accelerator that they justify to be executed on a system that is constructed with the GPU as its main compute resource in mind.

The GPU is not there to replace the CPU, but to complement it. As the data exchange between different processing units in the system gets faster to the point where heterogeneous architectures are combined on the same chip, hybrid solutions that switch between execution modes to utilise the most optimal processing units for a given task become increasingly powerful.

10.3. Opportunities for Future Work

The work begun in the course of these studies leaves various opportunities for future projects. The experiments with the rewired Ising model show how irregularities in the underlying graph structure shift the critical point of the phase transition that marks the emergence of order in the initially unordered system. The irregularities are currently created by random rewiring of a fraction of the existing connections. It would be interesting to see how different rewiring strategies, based on a non-uniform random distribution for the neighbour selection, change the critical point. Alternatively, it may be worth investigating how additional long distance links added to the otherwise untouched regular lattice structure change the dynamics in comparison to the results presented here.

The cortical model used for the simulation of the neural network can be extended in many ways to make it more realistic and to add additional functionalities not yet considered by the current implementation. A number of possible extensions are discussed in Section 8.8. However, additional work on the model should first concentrate on tuning the existing parameters to obtain the most realistic results achievable before new features make the task more complex. Once this is achieved, it will be interesting to analyse both temporal and spatial spike patterns and how they change when anaesthetic drugs are simulated by the system.

New generations of CPUs, GPUs and chips that combine the two architectures into one package will offer many research opportunities. Development tools and programming models that enable users to efficiently utilise all the processing resources in a machine are quickly evolving. Frameworks for heterogeneous architectures need to be flexible but also offer enough control over the hardware so that developers can target specific features of a particular device. Especially GPUs require relatively low level tuning to reach their full potential. While this is true to some extent for all hardware architectures, compilers for CPUs are already much more advanced and can be trusted to perform a higher level of optimisation automatically.

The quickly increasing compute capabilities of existing and upcoming parallel devices enable scientists to tackle problems with a level of detail that was not feasible in the past. Graph analysis algorithms and complex systems simulations like those discussed in this thesis benefit from ever increasing system sizes. Bigger system not only tend to produce more accurate results with smaller statistical errors, but they may also allow new types of complex behaviour to emerge.

Appendices

Appendix A

Milgram's Small-World Experiment

To find an answer to his research question, Stanley Milgram performed his experiment two times before he published his findings in an article with the title "The Small-World Problem" [64]. For the second study, which the article describes in detail, a number of persons were randomly chosen from the people living in Omaha, Nebraska. They acted as the source of a message which was to be sent to a randomly selected target person, a stockbroker who worked in Boston and lived in Sharon, Massachusetts. The source persons and the target person were all chosen from the population of the United States of America. If this is a good representation of the entire human population is another question, but the results were certainly interesting and stimulated many scientists to conduct further research. The rules for the study were as follows [64]:

> Each person who volunteered to serve as a starting person was sent a folder containing a document, which served as the main tool of investigation. Briefly, the document contains:
>
> 1. The name of the target person as well as certain information about him. This orients the participants toward a specific individual.
> 2. A set of rules for reaching the target person. Perhaps the most important rule is: "If you do not know the target person on a personal basis, do not try to contact him directly. Instead, mail this folder ... to a personal acquaintance who is more likely than you to know the target person ... it must be someone you know on a first-name basis." This rule sets the document into motion, moving it from one participant to the next, until it is sent to someone who knows the target person.
> 3. A roster on which each person in the chain writes his name. This tells the person who receives the folder exactly who sent it to him. The roster also has another practical effect; it prevents endless looping of the folder through participants who have already served as links in the chain, because each participant can see exactly what sequence of persons has led up to his own participation.
>
> In addition to the document, the folder contains a stack of 15 business reply, or "tracer" cards. Each person receiving the folder takes out a card, fills it in, returns it to us, and sends the remaining cards along with the document.

Only 44 of the 160 chains that started in Nebraska arrived at their destination. This is not very surprising though, because each chain has a chance to fail at every step when a participant fails to forward the message. However, the 44 chains that were completed successfully only needed between 2 and 10 intermediaries – with an average of 5 – to reach the target person. There is a chance that the chains that failed to complete

would have been longer, and failed exactly because the chance of failing increased with every intermediary. On the other hand, being human the participants are likely to make decisions that are not optimal, overlooking an acquaintance that would be better suited as the next link in the chain.

Appendix B

The Small-World α-Model

Watts defines the α-model [83] for choices of number of nodes n, connectivity k and parameter α as follows:

1. Consider in turn each vertex i. The vertices $i = 1...n$ are chosen in random order, but once a vertex has been wired by choosing a new neighbour, it may not choose again until all other vertices have taken their turn in this round.

2. For every other vertex $j \neq i$, compute $R_{i,j}$ according to Equation B.1, imposing the additional constraint that $R_{i,j} = 0$ if vertices i and j are already connected.

3. Then sum the $R_{i,j}$ over all $j \neq i$ and normalise each to obtain variables $P_{i,j} = R_{i,j} / \sum_{l \neq i} R_{i,l}$. Now $\sum_j P_{i,j} = 1$, $P_{i,j}$ can be interpreted as the probability that i will be connected to j. In addition, $P_{i,j}$ can be interpreted geometrically as dividing $[0,1)$ – the unit interval – into $n-1$ half-open subintervals with length $P_{i,j}, \forall j \neq i$.

4. A uniform pseudo-random variable is generated on $[0,1)$. It will fall into one of the subintervals, which is identified as corresponding to j_*.

5. Connect i to j_*.

The five steps of this procedure are then repeated until the predetermined number of edges $(M = (k*n)/2)$ has been constructed.

$$R_{i,j} = \left\{ \begin{array}{ll} 1, & m_{i,j} \geq k \\ \left[\frac{m_{i,j}}{k}\right]^{\alpha} (1-p) + p, & k > m_{i,j} > 0 \\ p, & m_{i,j} = 0 \end{array} \right\} \tag{B.1}$$

where:

$R_{i,j}$ = a measure of vertex i's propensity to connect to vertex j (zero if they are already connected)

$m_{i,j}$ = the number of vertices which are adjacent both to i and j

k = the average degree of the graph

p = a baseline, random probability of an edge (i,j) existing $(p \ll \binom{n}{2}^{-1})$

α = a tunable parameter, $0 <= \alpha <= \infty$.

Appendix C

Binder Cumulant Results: Continued

Section 7.7.2 continued: The detailed results for the remaining Binder cumulant measurements.

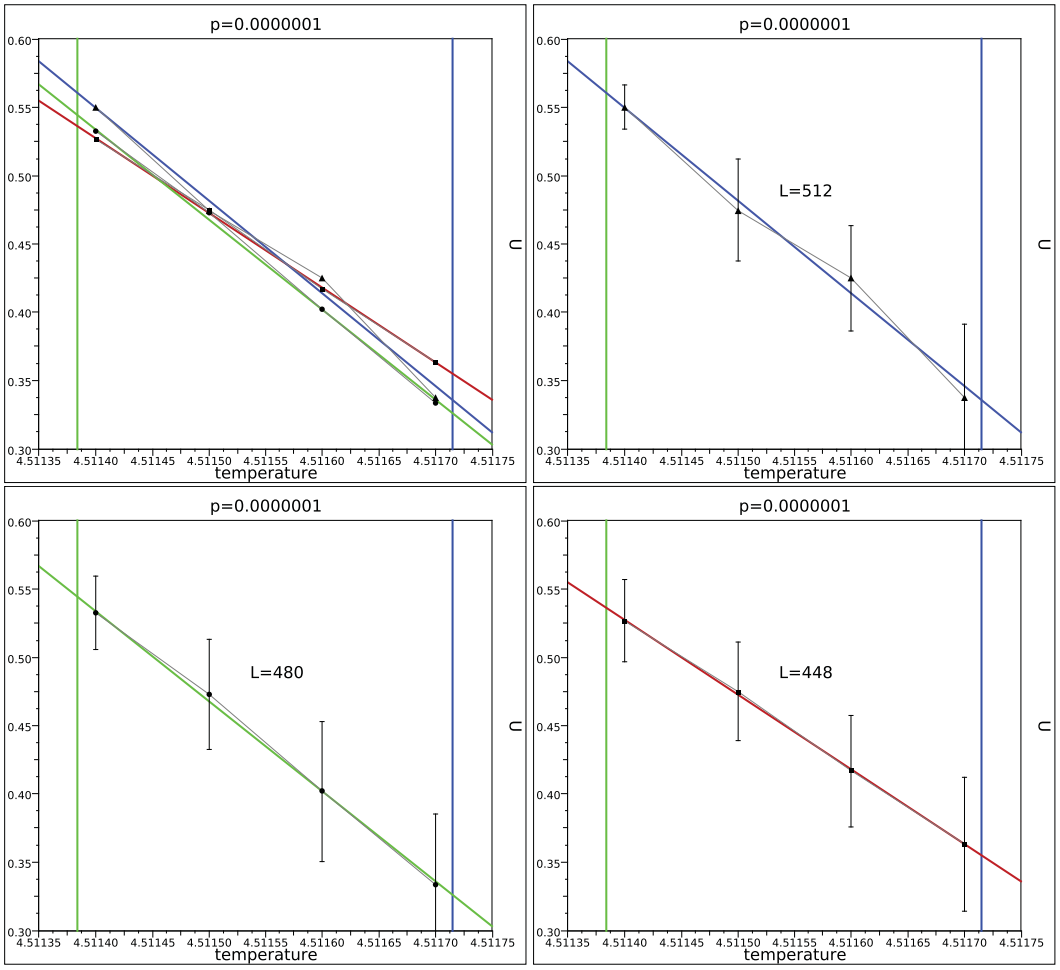


Figure C.1.: Intersection at $T_c(p = 10^{-7}) = 4.51175 \pm 0.00041$.

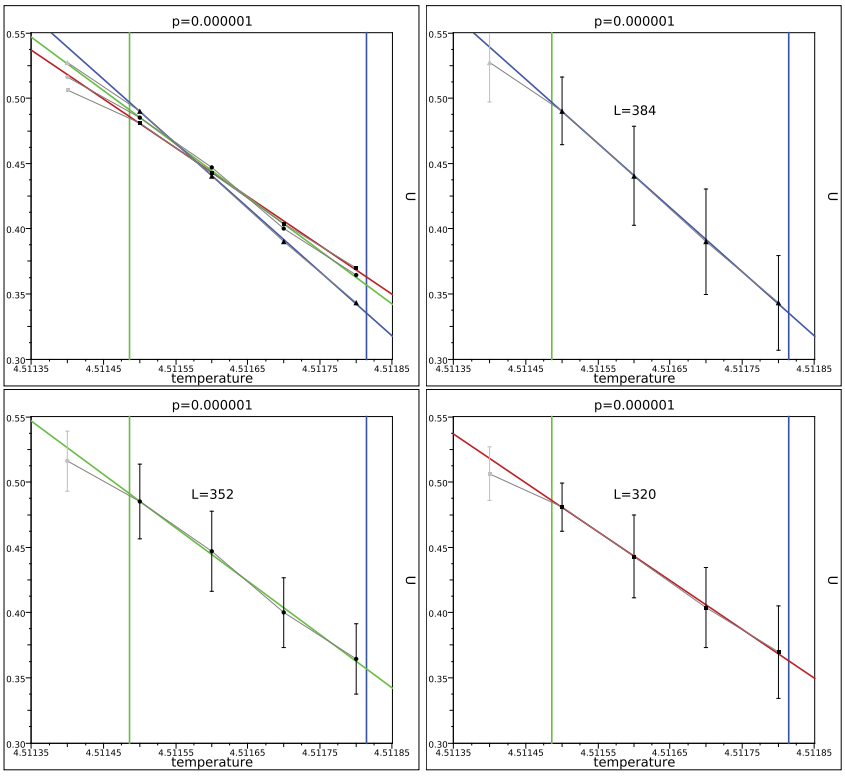


Figure C.2.: Intersection at $T_c(p = 10^{-6}) = 4.51159 \pm 0.00004$.

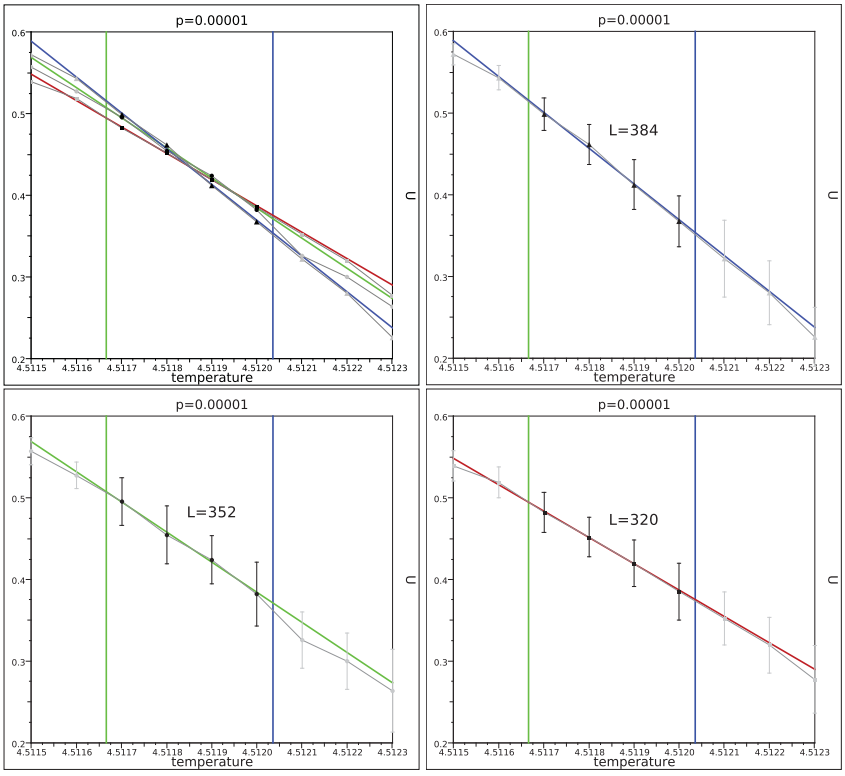


Figure C.3.: Intersection at $T_c(p = 10^{-5}) = 4.51186 \pm 0.00008$.

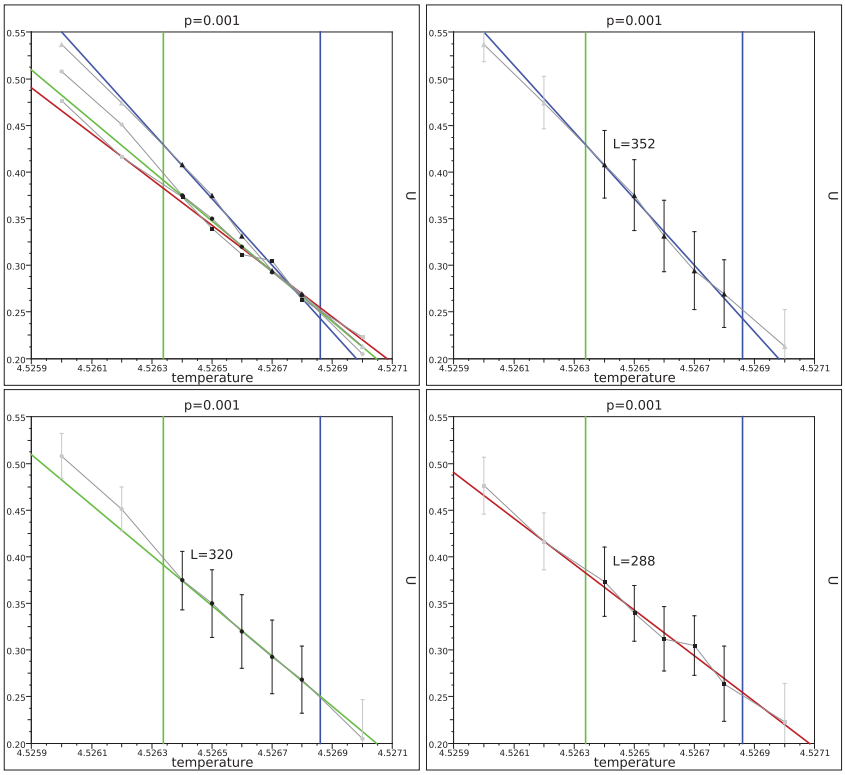


Figure C.4.: Intersection at $T_c(p = 10^{-3}) = 4.52674 \pm 0.00004$.

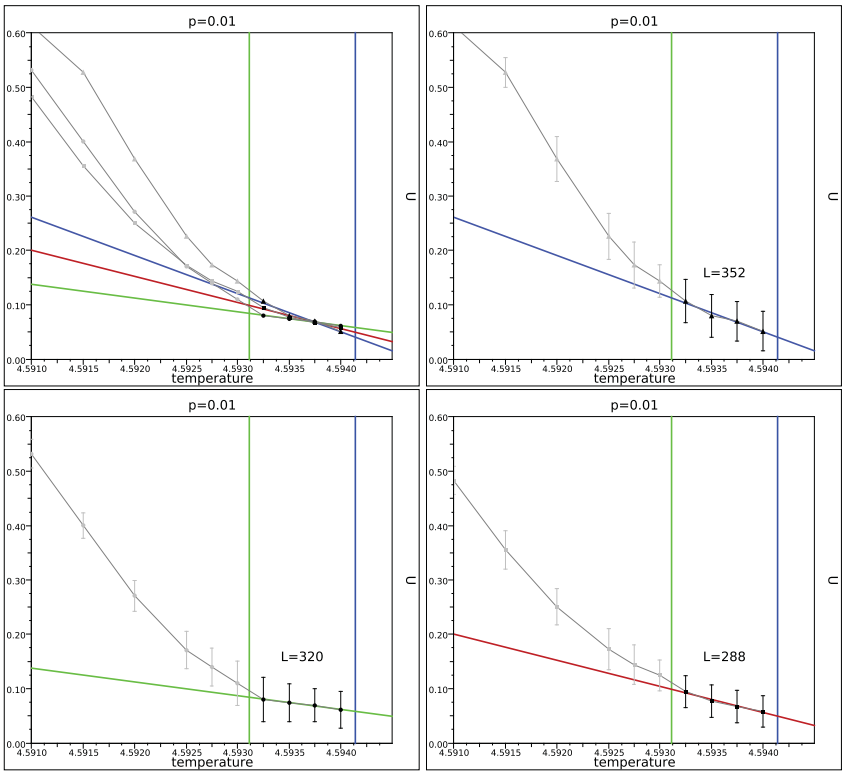


Figure C.5.: Intersection at $T_c(p = 10^{-2}) = 4.593754 \pm 0.000007$.

Appendix D

Different Types of Spiking Neurons

Figure D.1 illustrates the different types of firing patterns produced by neurons in the Izhikevich model. The excitatory and inhibitory cortical neurons visualised here are used in the neural network simulation discussed in Chapter 8.

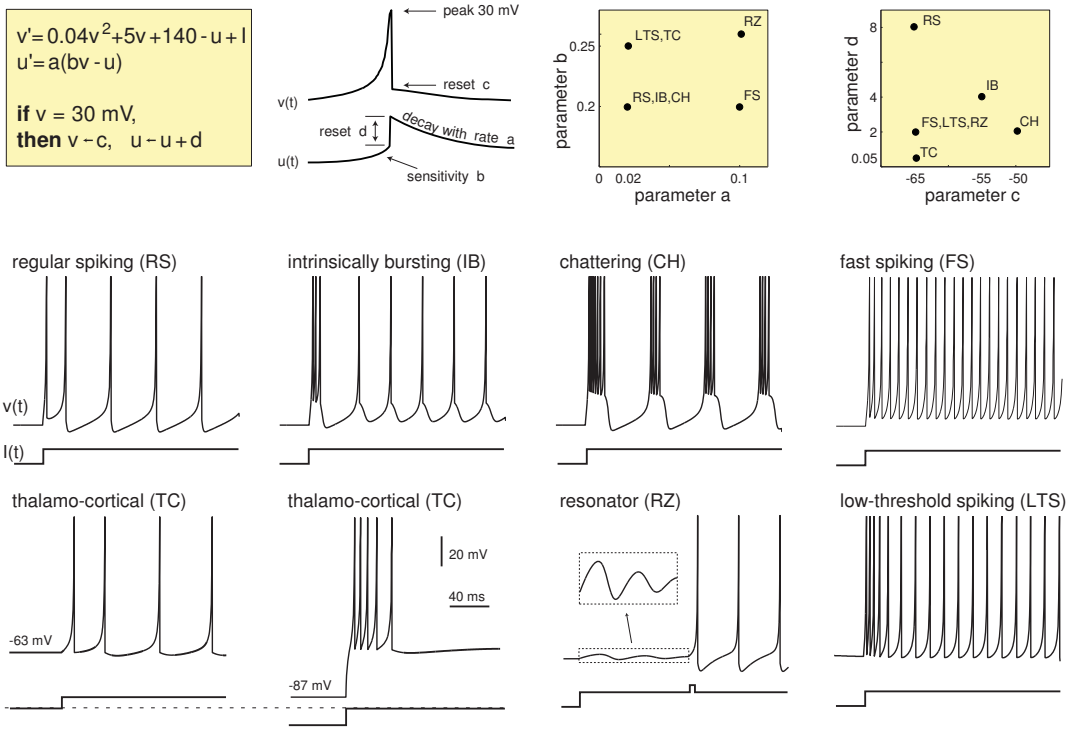


Figure D.1.: The types of spiking and bursting neurons reproduced by the Izhikevich model correspond to different values of parameters a, b, c, d [182]. Regular spiking, intrinsically bursting and chattering type cells are cortical excitatory neurons. Fast spiking and low-threshold spiking type cells are cortical inhibitory neurons. The insets show the voltage response of a model neuron to a step of dc-current $I = 10$. This figure is reproduced with permission from www.izhikevich.com. (Electronic version of the figure and reproduction permissions are freely available at www.izhikevich.com.)

Appendix E

List of Publications

The author's refereed conference and journal publications in reverse chronological order:

A. Leist and K. A. Hawick, "Graph Generation on GPUs using Dynamic Memory Allocation," in *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'11)*, no. PDP3939, Las Vegas, USA, July 2011

A. Leist, K. A. Hawick, and D. P. Playne, "GPGPU and Multi-Core Architectures for Computing Clustering Coefficients of Irregular Graphs," in *Proc. International Conference on Scientific Computing (CSC'11)*, no. CSC2720, Las Vegas, USA, July 2011

V. Du Preez, M. G. B. Johnson, **A. Leist**, and K. A. Hawick, "Performance and Quality of Random Number Generators," in *International Conference on Foundations of Computer Science (FCS'11)*, no. FCS4818, Las Vegas, USA, July 2011

A. P. Gerdelan, K. A. Hawick, **A. Leist**, and D. P. Playne, "Simulation Frameworks for Virtual Environments," in *Proc. International Conference on Internet Computing (ICOMP'11)*, no. ICM4087, Las Vegas, USA, July 2011

K. A. Hawick, **A. Leist**, D. P. Playne, and M. J. Johnson, "Speed and Portability issues for Random Number Generation on Graphical Processing Units with CUDA and other Processing Accelerators," in *Proc. Australasian Computer Science Conference (ACSC 2011)*, 2011

K. A. Hawick, **A. Leist**, and D. P. Playne, "Regular Lattice and Small-World Spin Model Simulations using CUDA and GPUs," *International Journal of Parallel Programming*, vol. 39, no. 2, pp. 183–201, April 2011

K. A. Hawick, **A. Leist**, and D. P. Playne, "Parallel Graph Component Labelling with GPUs and CUDA," *Parallel Computing*, vol. 36, no. 12, pp. 655–678, December 2010

K. A. Hawick, **A. Leist**, D. P. Playne, and M. J. Johnson, "Comparing Intra- and Inter-Processor Parallelism on Multi-Core CellBE Processors for Scientific Simulations," in *Proc. Parallel and Distributed Computing and Systems (PDCS 2010)*, 2010

A. Leist, D. P. Playne, and K. A. Hawick, "Interactive Visualisation of Spins and Clusters in Regular and Small-World Ising Models with CUDA on GPUs," *Journal of Computational Science*, vol. 1, no. 1, pp. 33–40, May 2010

A. Leist, D. P. Playne, and K. A. Hawick, "Visualising spins and clusters in regular and small-world Ising models with GPUs," in *Procedia Computer Science*, vol. 1, no. 1, May 2010

A. Leist, D. P. Playne, and K. A. Hawick, "Exploiting Graphical Processing Units for Data-Parallel Scientific Applications," *Concurrency and Computation: Practice and Experience*, vol. 21, no. 18, pp. 2400–2437, December 2009

A. Leist and K. A. Hawick, "Circuits as a Classifier for Small-World Network Models," in *Proc. WORLDCOMP 2009 International Conference on Foundations of Computer Science (FSC 09)*, July 2009

A. Leist and K. A. Hawick, "A Small-World Network Model for Distributed Storage of Semantic Metadata," in *Proceedings of the 7th Australasian Symposium on Grid Computing and e-Research (AUSGRID 2009)*, ser. Conferences in Research and Practice in Information Technology (CRPIT), W. Kelly and P. Roe, Eds., vol. 99, January 2009

Bibliography

[1] S. R. M. Barros and T. Kauranne, "On the parallelization of global spectral weather models," *Parallel Computing*, vol. 20, no. 9, pp. 1335–1356, September 1994.

[2] F. F. Abraham, S. W. Koch, and R. C. Desai, "Computer-Simulation Dynamics of an Unstable Two-Dimensional Fluid: Time-Dependent Morphology and Scaling," *Phys. Rev. Lett.*, vol. 49, no. 13, pp. 923–926, 1982.

[3] C.-Y. Chow, *An Introduction to Computational Fluid Mechanics*. John Wiley & Sons, 1979.

[4] M. G. B. Johnson, D. P. Playne, and K. A. Hawick, "Data-Parallelism and GPUs for Lattice Gas Fluid Simulations," in *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'10)*, Las Vegas, USA, July 2010.

[5] S. D. Shtovba, "Ant algorithms: Thory and applications," *Programming And Computer Software*, vol. 31, no. 4, pp. 167–178, 2005.

[6] E. Hadavandi, H. Shavandi, and A. Ghanbari, "Integration of genetic fuzzy systems and artificial neural networks for stock price forecasting," *Knowledge-Based Systems*, vol. 23, no. 8, pp. 800–808, December 2010.

[7] C. J. Scogings and K. A. Hawick, "Emergent Societal Effects of Crimino-Social Forces in an Animat Agent Model," in *Proc. 4th Australiasian Conference on Artificial Life (ACAL'09), Melbourne, Australia.*, ser. LNAI, no. 5865. Springer, December 2009, pp. 191–200.

[8] J. M. Watts, "Animats: Computer-simulated animals in behavioral research," *Journal of Animal Science*, vol. 76, no. 10, pp. 2596–2604, October 1998.

[9] R. Albert, H. Jeong, and A.-L. Barabási, "Diameter of the World-Wide Web," *Nature*, vol. 401, no. 6749, pp. 130–131, September 1999.

[10] S. F. Reddaway, "DAP - A Distributed Array Processor," in *Proceedings of the 1st annual symposium on Computer Architecture*, 1973.

[11] R. M. Russell, "The CRAY-1 computer system," *Communications of the ACM*, vol. 21, no. 1, pp. 63–72, 1978.

[12] D. J. Wallace, "Supercomputing With Transputers," *Computing Systems in Engineering*, vol. 1, no. 1, pp. 131–141, 1990.

[13] L. W. Tucker and G. G. Robertson, "Architecture and Applications of the Connection Machine," *Computer*, vol. 21, no. 8, pp. 26–38, 1988.

[14] M. J. Flynn, “Some Computer Organizations and Their Effectiveness,” *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, 1972.

[15] C. Boyd, “Data-Parallel Computing,” *acm queue*, vol. 6, no. 2, pp. 30–39, March/April 2008.

[16] P. J. Denning and J. B. Dennis, “The Resurgence of Parallelism,” *Communications of the ACM*, vol. 53, no. 6, pp. 30–32, June 2010.

[17] J. Lengyel, M. Reichert, B. R. Donald, and D. P. Greenberg, “Real-time Robot Motion Planning Using Rasterizing Computer Graphics Hardware,” *Computer Graphics*, vol. 24, no. 4, pp. 327–335, 1990.

[18] C. A. Bohn, “Kohonen Feature Mapping through Graphics Hardware,” in *Proceedings of the International Conference on Computational Intelligence and Neurosciences*, 1998.

[19] M. J. Harris, “Real-Time Cloud Simulation and Rendering,” Ph.D. dissertation, University of North Carolina, 2003.

[20] M. J. Harris, “General-Purpose Computation on Graphics Hardware,” http://gpgpu.org (last accessed July 2011).

[21] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, “GPU Cluster for High Performance Computing,” in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, 2004.

[22] TOP500.org, “TOP 500 Supercomputer Sites,” http://www.top500.org/ (last accessed July 2011).

[23] NVIDIA® Corporation, “The Compute Unified Device Architecture (CUDA),” http://developer.nvidia.com/ (last accessed July 2011), 2008.

[24] Advanced Micro Devices, Inc., “AMD Stream SDK,” http://www.amd.com/stream/ (last accessed July 2011), 2008.

[25] NVIDIA® Corporation, “PhysX,” http://developer.nvidia.com/physx/ (last accessed July 2011), 2011.

[26] S. L. Johnsson, “Data Parallel Supercomputing,” in *The Dawn of Massively Parallel Processing in Meteorology: Proceedings of the 3rd Workshop on use of Parallel Processors in Meteorology*, Thinking Machines Corporation. Springer, 1990.

[27] Graph500.org, “The Graph 500 List,” http://www.graph500.org/ (last accessed July 2011).

[28] Y. Frishman and A. Tal, “Multi-Level Graph Layout on the GPU,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, pp. 1310–1317, November/December 2007.

[29] P. Harish and P. J. Narayanan, “Accelerating large graph algorithms on the GPU using CUDA,” in *High Performance Computing - HiPC 2007: 14th International Conference, Proceedings*, S. Aluru, M. Parashar, R. Badrinath, and V. K. Prasanna, Eds., vol. 4873. Goa, India: Springer-Verlag, December 2007, pp. 197–208.

[30] **A. Leist**, D. P. Playne, and K. A. Hawick, “Exploiting Graphical Processing Units for Data-Parallel Scientific Applications,” *Concurrency and Computation: Practice and Experience*, vol. 21, no. 18, pp. 2400–2437, December 2009.

[31] G. Marsaglia, A. Zaman, and W. W. Tsang, “Toward a universal random number generator,” *Statistics & Probability Letters*, vol. 9, no. 1, pp. 35–39, January 1990.

[32] E. Ising, "Beitrag zur Theorie des Ferromagnetismus," *Zeitschrift fuer Physik A Hadrons and Nuclei*, vol. 31, no. 1, pp. 253–258, 1925.

[33] J. P. Glusker and K. N. Trueblood, *Crystal Structure Analysis: A Primer*, 3rd ed. Oxford University Press, 2010.

[34] D. P. Landau and K. Binder, *A Guide to Monte-Carlo Simulations in Statistical Physics*. Cambridge University Press, 2009, vol. 3rd edition.

[35] P. Erdős and A. Rényi, "On random graphs," *Publicationes Mathematicae*, vol. 6, pp. 290–297, 1959.

[36] R. Solomonoff and A. Rapoport, "Connectivity Of Random Nets," *Bulletin of Mathematical Biophysics*, vol. 13, pp. 107–117, 1951.

[37] E. N. Gilbert, "Random Graphs," *Annals of Mathematical Statistics*, vol. 30, no. 4, pp. 1141–1144, 1959.

[38] A. Papoulis and S. U. Pillai, *Probability, Random Variables and Stochastic Processes*, 4th ed. McGraw-Hill, 2002.

[39] M. E. J. Newman, "The Structure and Function of Complex Networks," *SIAM Review*, vol. 45, no. 2, pp. 167–256, June 2003.

[40] L. d. F. Costa and F. A. Rodrigues, "Seeking for simplicity in complex networks," *EPL*, vol. 85, no. 4, pp. 48 001–p1–48 001–p6, February 2009.

[41] A.-L. Barabási, "Scale-Free Networks: A Decade and Beyond," *Science*, vol. 325, no. 5939, pp. 412–413, July 2009.

[42] N. Wormald, *Handbook of Graph Theory*. CRC Press, 2004, ch. 8.2 Random Graphs, pp. 817–836.

[43] C. E. Shannon, "A Mathematical Theory of Communication," *Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.

[44] D. F. Styer, "Insight into entropy," *American Journal of Physics*, vol. 68, no. 12, pp. 1090–1096, 2000.

[45] D. J. C. MacKay, *Information Theory, Inference, and Learning Algorithms*, 7th ed. Cambridge University Press, 2005.

[46] N. Rashevsky, "Life, information theory, and topology," *Bulletin of Mathematical Biology*, vol. 17, no. 3, pp. 229–235, 1955.

[47] E. Trucco, "A note on the information content of graphs," *Bulletin of Mathematical Biology*, vol. 18, no. 2, pp. 129–135, 1956.

[48] A. Mowshowitz, "Entropy And The Complexity Of Graphs: I. An Index Of The Relative Complexity Of A Graph," *Bulletin of Mathematical Biophysics*, vol. 30, no. 1, pp. 175–204, 1968.

[49] M. Dehmer and A. Mowshowitz, "A history of graph entropy measures," *Information Sciences*, vol. 181, no. 1, pp. 57–78, January 2011.

[50] L. d. F. Costa, M. Kaiser, and C. Hilgetag, "Beyond the average: detecting global singular nodes from local features in complex networks," Universidade de Sao Paulo, University of Newcastle, International University Bremen, Tech. Rep., July 2006.

[51] L. d. F. Costa, F. A. Rodrigues, C. C. Hilgetag, and M. Kaiser, "Beyond the average: Detecting global singular nodes from local features in complex networks," *EPL*, vol. 87, no. 1, pp. 18 008–p1–18 008–p6, July 2009.

[52] L. d. F. Costa, F. A. Rodrigues, G. Travieso, and P. R. V. Boas, "Characterization of complex networks: A survey of measurements," *Advances in Physics*, vol. 56, no. 1, pp. 167–242, 2007.

[53] J. Kim and T. Wilhelm, "What is a complex graph?" *Physica A: Statistical Mechanics and its Applications*, vol. 387, no. 11, pp. 2637–2652, April 2008.

[54] M. E. J. Newman, S. H. Strogatz, and D. J. Watts, "Random graphs with arbitrary degree distributions and their applications," *Physical Review E*, vol. 64, no. 2, p. 026118, July 2001.

[55] J. J. Binney, N. J. Dowrick, A. J. Fisher, and M. E. J. Newman, *The Theory of Critical Phenomena - An Introduction to the Renormalization Group*. Oxford University Press, 1992.

[56] H. Jeong, B. Tombor, R. Albert, Z. N. Oltvai, and A.-L. Barabási, "The large-scale organization of metabolic networks," *Nature*, vol. 407, no. 6804, pp. 651–654, October 2000.

[57] D. J. d. S. Price, "Networks of Scientific Papers," *Science*, vol. 149, no. 3683, pp. 510–515, July 1965.

[58] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, October 1999.

[59] S. N. Dorogovtsev, A. V. Goltsev, and J. F. F. Mendes, "Critical phenomena in complex networks," Universidade de Aveiro, Tech. Rep., November 2007.

[60] M. Buchanan, *NEXUS - Small Worlds and the Groundbreaking Science of Networks*. W.W. Norton & Company, 2002.

[61] R. Albert, H. Jeong, and A.-L. Barabási, "Error and attack tolerance of complex networks," *Nature*, vol. 406, no. 6794, pp. 378–382, July 2000.

[62] P. Holme, B. J. Kim, C. N. Yoon, and S. K. Han, "Attack vulnerability of complex networks," *Physical Review E*, vol. 65, no. 5, May 2002.

[63] J. Xu and H. Chen, "The Topology of Dark Networks," *Communications of the ACM*, vol. 51, no. 10, pp. 58–65, October 2008.

[64] S. Milgram, "The Small-World Problem," *Psychology Today*, vol. 1, pp. 61–67, 1967.

[65] **A. Leist** and K. A. Hawick, "Circuits as a Classifier for Small-World Network Models," in *Proc. WORLDCOMP 2009 International Conference on Foundations of Computer Science (FSC 09)*, July 2009.

[66] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, no. 6684, pp. 440–442, June 1998.

[67] M. E. J. Newman, "Models of the Small World," *Journal of Statistical Physics*, vol. 101, no. 3-4, pp. 819–841, November 2000.

[68] A.-L. Barabási, *Linked - The New Science of Networks*. Perseus Publishing, April 2002.

[69] J. Guare, *Six Degrees of Separation*. Vintage Books, 1990.

[70] M. Newman, A.-L. Barabási, and D. J. Watts, *The Structure and Dynamics of Networks*. Princeton University Press, 2006.

[71] D. A. Fell and A. Wagner, "The small world of metabolism," *Nature Biotechnology*, vol. 18, no. 11, pp. 1121–1122, November 2000.

[72] A. Wagner and D. A. Fell, "The small world inside large metabolic networks," *Proceedings of the Royal Society B*, vol. 268, no. 1478, pp. 1803–1810, September 2001.

[73] D. S. Bassett and E. Bullmore, "Small-World Brain Networks," *The Neuroscientist*, vol. 12, no. 6, pp. 512–523, 2006.

[74] M. E. J. Newman, "The structure of scientific collaboration networks," *PNAS*, vol. 98, no. 2, pp. 404–409, January 2001.

[75] M. E. J. Newman, "Ego-centered networks and the ripple effect," *Social Networks*, vol. 25, no. 1, pp. 83–95, January 2003.

[76] F. Liljeros, C. R. Edling, L. A. N. Amaral, H. E. Stanley, and Y. Aberg, "The web of human sexual contacts," *Nature*, vol. 411, no. 6840, pp. 907–908, June 2001.

[77] D. Liben-Nowell and J. Kleinberg, "Tracing information flow on a global scale using Internet chain-letter data," *PNAS*, vol. 105, no. 12, pp. 4633–4638, March 2008.

[78] J. M. Kleinberg, "Navigation in a small world," *Nature*, vol. 406, no. 6798, p. 845, August 2000.

[79] **A. Leist** and K. A. Hawick, "A Small-World Network Model for Distributed Storage of Semantic Metadata," in *Proceedings of the 7th Australasian Symposium on Grid Computing and e-Research (AUSGRID 2009)*, ser. Conferences in Research and Practice in Information Technology (CRPIT), W. Kelly and P. Roe, Eds., vol. 99, January 2009.

[80] **A. Leist** and K. A. Hawick, "Small-World Networks, Distributed Hash Tables and the e-Resource Discovery Problem," *Research Letters in the Information and Mathematical Sciences*, vol. 14, pp. 1–16, 2010.

[81] D. J. Watts, *Six Degrees - The Science of a Connected Age*. W.W. Norton & Company, 2003.

[82] M. E. J. Newman and D. J. Watts, "Renormalization group analysis of the small-world network model," *Physics Letters A*, vol. 263, no. 4-6, pp. 341–346, December 1999.

[83] D. J. Watts, *Small Worlds: The Dynamics of Networks between Order and Randomness*. Princeton University Press, 1999.

[84] M. Barthélémy and L. A. N. Amaral, "Small-World Networks: Evidence for a Crossover Picture," *Physical Review Letters*, vol. 82, no. 15, pp. 3180–3183, 1999.

[85] M. Barthélémy and L. A. N. Amaral, "Erratum: Small-World Networks: Evidence for a Crossover Picture," *Phys. Rev. Lett.*, vol. 82, no. 25, pp. 5180–5180, 1999.

[86] M. E. J. Newman and D. J. Watts, "Scaling and percolation in the small-world network model," *Physical Review E*, vol. 60, no. 6, pp. 7332–7342, December 1999.

[87] *IEEE Std. 1003.1c-1995 thread extensions*, IEEE, 1995.

[88] Intel Corporation, *Intel(R) Threading Building Blocks 3.0 Reference Manual*, May 2010.

[89] *Cell Broadband Engine Programming Tutorial Version 3.1*, IBM.

[90] *NVIDIA CUDA C Programming Guide Version 3.2*, NVIDIA® Corporation, 2010, http://www.nvidia.com/ (last accessed June 2011).

[91] *Fermi Architecture Whitepaper Version 1.1*, NVIDIA® Corporation, 2009, http://www.nvidia.com/ (last accessed June 2011).

[92] K. A. Hawick, **A. Leist**, and D. P. Playne, "Mixing Multi-Core CPUs and GPUs for Scientific Simulation Software," *Research Letters in the Information and Mathematical Sciences*, vol. 14, no. ISSN 1175-2777, pp. 25–77, 2010.

[93] K. A. Hawick, **A. Leist**, D. P. Playne, and M. J. Johnson, "Speed and Portability issues for Random Number Generation on Graphical Processing Units with CUDA and other Processing Accelerators," in *Proc. Australasian Computer Science Conference (ACSC 2011)*, 2011.

[94] ID Quantique, "White Paper: Random Number Generation Using Quantum Physics," ID Quantique SA, Switzerland, Tech. Rep. Version 3.0, April 2010.

[95] V. Du Preez, M. G. B. Johnson, **A. Leist**, and K. A. Hawick, "Performance and Quality of Random Number Generators," in *International Conference on Foundations of Computer Science (FCS'11)*, no. FCS4818, Las Vegas, USA, July 2011.

[96] G. Marsaglia, "Diehard Battery Of Tests Of Randomness," http://www.stat.fsu.edu/pub/diehard/ (last accessed June 2011), 1995.

[97] G. Marsaglia and W. W. Tsang, "Some Difficult-to-pass Tests of Randomness," *Journal of Statistical Software*, vol. 7, no. 3, pp. 1–9, January 2002.

[98] W. Hörmann, "The Transformed Rejection Method for Generating Poisson Random Variables," *Insurance: Mathematics and Economics*, vol. 12, no. 1, pp. 39–45, February 1993.

[99] D. B. Thomas, W. Luk, P. H. W. Leong, and J. D. Villasenor, "Gaussian Random Number Generators," *ACM Computing Surveys*, vol. 39, no. 4, p. Article 11, 2007.

[100] P. L'Ecuyer, "Software For Uniform Random Number Generation: Distinguishing The Good And The Bad," in *Proceedings of the 2001 Winter Simulation Conference*, B. A. Peters, J. S. Smith, D. J. Medeiros, and M. W. Rohrer, Eds., vol. 1-2, 2001.

[101] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3–30, 1998.

[102] R. P. Brent, "A Fast Vectorised Implementation of Wallace's Normal Random Number Generator," The Australian National University, Tech. Rep. TR-CS-97-07, 1997.

[103] P. D. Coddington and A. J. Newell, "JAPARA - A Java Parallel Random Number Generator Library for High-Performance Computing," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[104] D. A. Bader, A. Chandramowlishwaran, and V. Agarwal, "On the Design of Fast Pseudo-Random Number Generators for the Cell Broadband Engine and an Application to Risk Analysis," in *Proceedings of the International Conference on Parallel Processing (ICPP)*, 2008.

[105] A. M. Ferrenberg and D. P. Landau, “Monte Carlo simulations: Hidden errors from ”good” random number generators,” *Physical Review Letters*, vol. 69, no. 23, pp. 3382–3384, December 1992.

[106] P. D. Coddington, “Analysis of Random Number Generators Using Monte-Carlo Simulation,” *International Journal of Modern Physics C*, vol. 5, no. 3, pp. 547–560, June 1994.

[107] R. P. Brent, “Uniform Random Number Generators for Supercomputers,” in *Proceedings of the Fifth Australian Supercomputer Conference*, 1992, pp. 95–104.

[108] R. P. Brent, “Fast Normal Random Number Generators for Vector Processors,” The Australian National University, Tech. Rep. TR-CS-93-04, 1993.

[109] P. D. Coddington, “Random Number Generators for Parallel Computers,” The University of Adelaide, Tech. Rep., 1996.

[110] T.-C. Lu, Y.-S. Hou, and R.-J. Chen, “A Parallel Poisson Generator Using Parallel Prefix,” *Computers & Mathematics with Applications*, vol. 31, no. 3, pp. 33–42, 1996.

[111] W. B. Langdon, “A Fast High Quality Pseudo Random Number Generator for nVidia CUDA,” in *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference*, 2009, pp. 2511–2514.

[112] K. A. Hawick, **A. Leist**, and D. P. Playne, “Regular Lattice and Small-World Spin Model Simulations using CUDA and GPUs,” *International Journal of Parallel Programming*, vol. 39, no. 2, pp. 183–201, April 2011.

[113] G. Marsaglia and L.-H. Tsay, “Matrices and the structure of random number sequences,” *Linear Algebra and its Applications*, vol. 67, pp. 147–156, 1985.

[114] **A. Leist**, K. A. Hawick, and D. P. Playne, “GPGPU and Multi-Core Architectures for Computing Clustering Coefficients of Irregular Graphs,” in *Proc. International Conference on Scientific Computing (CSC’11)*, no. CSC2720, Las Vegas, USA, July 2011.

[115] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes - The Art of Scientific Computing: Third Edition*, 3rd ed. Cambridge University Press, 2007, ch. 7 Random Numbers, pp. 340–418.

[116] **A. Leist** and K. A. Hawick, “Graph Generation on GPUs using Dynamic Memory Allocation,” in *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’11)*, no. PDP3939, Las Vegas, USA, July 2011.

[117] E. W. Dijkstra, “A Note on Two Problems in Connexion with Graphs,” *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[118] R. W. Floyd, “Algorithm 97: Shortest Path,” *Communications of the ACM*, vol. 5, no. 6, p. 345, 1962.

[119] D. B. Johnson, “Finding All The Elementary Circuits Of A Directed Graph,” *SIAM Journal on Computing*, vol. 4, no. 1, pp. 77–84, March 1975.

[120] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction To Algorithms*, 2nd ed. MIT Press, 2001.

[121] M. Harris, J. D. Owens, S. Sengupta, S. Tzeng, Y. Zhang, and A. Davidson, *CUDPP: CUDA Data Parallel Primitives Library*.

[122] P. Grindrod and D. J. Higham, "Evolving graphs: dynamical models, inverse problems and propagation," *Proceedings of the Royal Society A*, vol. 466, no. 2115, pp. 753–770, 2010.

[123] J. Bohannon, "Counterterrorism's New Tool: 'Metanetwork' Analysis," *Science*, vol. 325, no. 5939, pp. 409–411, 2009.

[124] A. H. Gebremedhin and F. Manne, "Scalable parallel graph coloring algorithms," *Concurrency: Practice and Experience*, vol. 12, pp. 1131–1146, 2000.

[125] E. G. Boman, D. Bozdag, U. Catalyurek, A. H. Gebremedhin, and F. Manne, "A Scalable Parallel Graph Coloring Algorithm for Distributed Memory Computers," in *Proceedings of Euro-Par 2005 Parallel Processing*. Springer-Verlag, 2005, pp. 241–251.

[126] C. F. Baillie and P. D. Coddington, "Comparison of cluster algorithms for two-dimensional Potts models," *Physical Review B*, vol. 43, no. 13, pp. 10 617–10 621, May 1991.

[127] C. F. Baillie and P. D. Coddington, "Cluster Identification Algorithms for Spin Models – Sequential and Parallel," *Concurrency: Practice and Experience*, vol. 3, no. 2, pp. 129–144, April 1991.

[128] K. A. Hawick, **A. Leist**, and D. P. Playne, "Parallel Graph Component Labelling with GPUs and CUDA," *Parallel Computing*, vol. 36, no. 12, pp. 655–678, December 2010.

[129] T. Schank and D. Wagner, "Approximating Clustering Coefficient and Transitivity," *Journal of Graph Algorithms and Applications*, vol. 9, no. 2, pp. 265–275, 2005.

[130] A. H. Abdo and A. P. S. de Moura, "Clustering as a measure of the local topology of networks," Universidade de São Paulo, University of Aberdeen, Tech. Rep., February 2008, http://arxiv.org/abs/physics/0605235v4 (last accessed June 2011).

[131] J. C. Tiernan, "An efficient search algorithm to find the elementary circuits of a graph," *Communications of the ACM*, vol. 13, no. 12, pp. 722–726, 1970.

[132] K. A. Hawick and H. A. James, "Enumerating Circuits and Loops in Graphs with Self-Arcs and Multiple-Arcs," Massey University, Tech. Rep. CSTN-013, November 2005.

[133] M. Niss, "History of the Lenz-Ising Model 1920-1950: From Ferromagnetic to Cooperative Phenomena," *Archive for History of Exact Sciences*, vol. 59, no. 3, pp. 267–318, 2005.

[134] L. Onsager, "Crystal Statistics. I. A Two-Dimensional Model with an Order-Disorder Transition," *Physical Review*, vol. 65, no. 3-4, pp. 117–149, 1944.

[135] S. Istrail, "Statistical mechanics, three-dimensionality and NP-completeness: I. Universality of intractability of the partition functions of the Ising model across non-planar lattices," in *Proceedings of the 32nd ACM Symposium on the Theory of Computing (STOC00)*, 2000.

[136] B. A. Cipra, "The Ising Model Is NP-Complete," *SIAM News*, vol. 33, no. 6, August 2000.

[137] G. F. Newell and E. W. Montroll, "On The Theory Of The Ising Model Of Ferromagnetism," *Reviews of Modern Physics*, vol. 25, no. 2, pp. 353–389, 1953.

[138] R. B. Potts, "Some Generalized Order-Disorder Transformations," in *Proceedings of the Cambridge Philosophical Society*, vol. 48, no. 1, 1952.

[139] J. J. Binney, N. J. Dowrick, A. J. Fisher, and M. E. J. Newman, *The Theory of Critical Phenomena - An Introduction to the Renormalization Group*. Oxford University Press, 1992, ch. 2 Statistical mechanics, pp. 33–53.

[140] P. Svenson, "Damage spreading in small world Ising models," *Physical Review E*, vol. 65, no. 3, p. 036105, 2002.

[141] A. Barrat and M. Weigt, "On the properties of small-world network models," *The European Physical Journal B*, vol. 13, no. 3, pp. 547–560, 2000.

[142] C. P. Herrero, "Ising model in small-world networks," *Physical Review E*, vol. 65, no. 6, p. 066110, 2002.

[143] C. P. Herrero, "Ising model in scale-free networks: A Monte Carlo simulation," *Physical Review E*, vol. 69, no. 6, p. 067109, 2004.

[144] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of State Calculations by Fast Computing Machines," *Journal of Chemical Physics*, vol. 21, no. 6, pp. 1087–1092, 1953.

[145] K. Binder, "Finite Size Scaling Analysis Of Ising Model Block Distribution Functions," *Zeitschrift fuer Physik B*, vol. 43, no. 2, pp. 119–140, 1981.

[146] G. S. Pawley, R. H. Swendsen, D. J. Wallace, and K. G. Wilson, "Monte Carlo renormalization-group calculations of critical behavior in the simple-cubic Ising model," *Physical Review B*, vol. 29, no. 7, pp. 4030–4040, 1984.

[147] C. F. Baillie, R. Gupta, K. A. Hawick, and G. S. Pawley, "Monte Carlo renormalization-group study of the three-dimensional Ising model," *Physical Review B*, vol. 45, no. 18, pp. 10 438–10 453, 1992.

[148] J. J. Binney, N. J. Dowrick, A. J. Fisher, and M. E. J. Newman, *The Theory of Critical Phenomena - An Introduction to the Renormalization Group*. Oxford University Press, 1992, ch. 12 The renormalization group at $T \neq T_c$, pp. 299–319.

[149] A. M. Ferrenberg and D. P. Landau, "Critical behavior of the three-dimensional Ising model: A high-resolution Monte Carlo study," *Physical Review B*, vol. 44, no. 10, pp. 5081–5091, September 1991.

[150] H. E. Stanley, *Introduction to Phase Transitions And Critical Phenomena*. Oxford University Press, 1971.

[151] A. Linke, D. W. Heermann, P. Altevogt, and M. Siegert, "Large-scale simulation of the two-dimensional kinetic Ising model," *Physica A: Statistical Mechanics and its Applications*, vol. 222, no. 1-4, pp. 205–209, December 1995.

[152] K. Okano, L. Schülke, K. Yamagishi, and B. Zheng, "Universality and scaling in short-time critical dynamics," *Nuclear Physics B*, vol. 485, no. 3, pp. 727–746, February 1997.

[153] B. Zheng, "Monte Carlo simulations and numerical solutions of short-time critical dynamics," *Physica A: Statistical Mechanics and its Applications*, vol. 283, no. 1-2, pp. 80–85, 2000.

[154] R. J. Glauber, "Time-Dependent Statistics of the Ising Model," *Journal of Mathematical Physics*, vol. 4, no. 2, pp. 294–307, 1963.

[155] R. H. Swendsen and J. S. Wang, "Nonuniversal Critical Dynamics in Monte Carlo Simulations," *Physical Review Letters*, vol. 58, no. 2, pp. 86–88, 1987.

[156] U. Wolff, "Comparison Between Cluster Monte Carlo Algorithms in the Ising Model," *Physics Letters B*, vol. 228, no. 3, pp. 379–382, 1989.

[157] E. Marinari and G. Parisi, "Simulated Tempering - a New Monte-Carlo Scheme," *Europhysics Letters*, vol. 19, no. 6, pp. 451–458, July 1992.

[158] W. K. Hastings, "Monte-Carlo Sampling Methods Using Markov Chains And Their Applications," *Biometrika*, vol. 57, no. 1, pp. 97–107, 1970.

[159] W. L. Jorgensen, "Perspective on "Equation of state calculations by fast computing machines"," *Theoretical Chemistry Accounts*, vol. 103, no. 3-4, pp. 225–227, February 2000.

[160] S. Chib and E. Greenberg, "Understanding the Metropolis-Hastings Algorithm," *American Statistician*, vol. 49, no. 4, pp. 327–335, November 1995.

[161] U. L. Fulco, L. S. Lucena, and G. M. Viswanathan, "Efficient search of critical points in Ising-like systems," *Physica A: Statistical Mechanics and its Applications*, vol. 264, no. 1-2, pp. 171–179, February 1999.

[162] F. W. S. Lima and D. Stauffer, "Ising model simulation in directed lattices and networks," *Physica A: Statistical Mechanics and its Applications*, vol. 359, no. 1, pp. 423–429, January 2006.

[163] **A. Leist**, D. P. Playne, and K. A. Hawick, "Visualising spins and clusters in regular and small-world Ising models with GPUs," in *Procedia Computer Science*, vol. 1, no. 1, May 2010.

[164] **A. Leist**, D. P. Playne, and K. A. Hawick, "Interactive Visualisation of Spins and Clusters in Regular and Small-World Ising Models with CUDA on GPUs," *Journal of Computational Science*, vol. 1, no. 1, pp. 33–40, May 2010.

[165] R. A. Drebin, L. Carpenter, and P. Hanrahan, "Volume rendering," in *Proceedings of the 15th annual conference on Computer graphics and interactive techniques (SIGGRAPH'88)*, 1988.

[166] T. Ritschel, "Fast GPU-based Visibility Computation for Natural Illumination of Volume Data Sets," in *Eurographics (Short Papers)*, 2007, pp. 57–60.

[167] S. Bruckner and M. E. Groller, "Enhancing depth-perception with flexible volumetric halos," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, pp. 1344–1351, 2007.

[168] M. Meyer, R. M. Kirby, and R. Whitaker, "Topology, arccuracy, and quality of isosurface meshes using dynamic particles," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, pp. 1704–1711, 2007.

[169] W. E. Lorensen and H. E. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," *Computer Graphics*, vol. 21, no. 4, pp. 163–169, 1987.

[170] K. A. Hawick, **A. Leist**, and D. P. Playne, "Cluster and Fast-Update Simulations of Regular and Rewired Lattice Ising Models Using CUDA and Graphical Processing Units," Massey University, Tech. Rep. CSTN-104, 2010. [Online]. Available: http://www.massey.ac.nz/~kahawick/cstn/

[171] K. Pearson, "Note on Regression and Inheritance in the Case of Two Parents," *Proceedings of the Royal Society*, vol. 58, pp. 240–242, 1895.

[172] J. L. Rodgers and W. A. Nicewander, "Thirteen Ways to Look at the Correlation Coefficient," *American Statistician*, vol. 42, no. 1, pp. 49–66, 1988.

[173] H.-O. Heuer, "Critical crossover phenomena in disordered Ising systems," *Journal of Physics A*, vol. 26, no. 6, pp. 333–339, 1993.

[174] M. L. Steyn-Ross, D. A. Steyn-Ross, J. W. Sleigh, and L. C. Wilcocks, "Toward a theory of the general-anesthetic-induced phase transition of the cerebral cortex. I. A thermodynamics analogy," *Physical Review E*, vol. 64, no. 1, p. 011917, 2001.

[175] D. A. Steyn-Ross, M. L. Steyn-Ross, L. C. Wilcocks, and J. W. Sleigh, "Toward a theory of the general-anesthetic-induced phase transition of the cerebral cortex. II. Numerical simulations, spectral entropy, and correlation times," *Physical Review E*, vol. 64, no. 1, p. 011918, 2001.

[176] M. L. Steyn-Ross, D. A. Steyn-Ross, and J. W. Sleigh, "Modelling general anaesthesia as a first-order phase transition in the cortex," *Progress in Biophysics & Molecular Biology*, vol. 85, no. 2-3, pp. 369–385, 2004.

[177] M. T. Alkire, A. G. Hudetz, and G. Tononi, "Consciousness and Anesthesia," *Science*, vol. 322, no. 5903, pp. 876–880, 2008.

[178] E. B. Friedman, Y. Sun, J. T. Moore, H.-T. Hung, Q. C. Meng, P. Perera, W. J. Joiner, S. A. Thomas, R. G. Eckenhoff, A. Sehgal, and M. B. Kelz, "A Conserved Behavioral State Barrier Impedes Transitions between Anesthetic-Induced Unconsciousness and Wakefulness: Evidence for Neural Inertia," *PloS ONE*, vol. 5, no. 7, p. e11903, 07 2010.

[179] A. L. Hodgkin and A. F. Huxley, "A Quantitative Description of Membrane Current and its Application to Conduction and Excitability in Nerve," *Journal of Physiology*, vol. 117, no. 4, pp. 500–544, 1952.

[180] R. Y. d. Camargo, L. Rozante, and S. W. Song, "A multi-GPU algorithm for large-scale neuronal networks," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 6, pp. 557–572, April 2011.

[181] E. M. Izhikevich, "Which Model to Use for Cortical Spiking Neurons?" *IEEE Transactions On Neural Networks*, vol. 15, no. 5, pp. 1063–1070, Septermber 2004.

[182] E. M. Izhikevich, "Simple Model of Spiking Neurons," *IEEE Transactions On Neural Networks*, vol. 14, no. 6, pp. 1569–1572, November 2003.

[183] J. M. Nageswaran, N. Dutt, J. L. Krichmar, A. Nicolau, and A. V. Veidenbaum, "A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors," *Neural Networks*, vol. 22, no. 5-6, pp. 791–800, July-August 2009.

[184] E. M. Izhikevich, "Neural Excitability, Spiking and Bursting," *International Journal of Bifurcation and Chaos*, vol. 10, no. 6, pp. 1171–1266, June 2000.

[185] J. A. Talavera, S. K. Esser, F. Amzica, S. Hill, and J. F. Antognini, "Modeling the Gabaergic Action of Etomidate on the Thalamocortical System," *Anesthesia & Analgesia*, vol. 108, no. 1, pp. 160–167, January 2009.

[186] Open MPI, "An Open Source Implementation of the Message Passing Interface," http://www.open-mpi.org/ (last accessed July 2011).

[187] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes - The Art of Scientific Computing: Third Edition*, 3rd ed. Cambridge University Press, 2007, ch. 12 Fast Fourier Transform, pp. 600–639.

[188] I. J. Rampil, "A Primer for EEG Signal Processing in Anesthesia," *Anesthesiology*, vol. 89, no. 4, pp. 980–1002, October 1998.

[189] H. Schwilden, "Concepts of EEG processing: from power spectrum to bispectrum, fractals, entropies and all that," *Best Practice & Research: Clinical Anaesthesiology*, vol. 20, no. 1, pp. 31–48, March 2006.

[190] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes - The Art of Scientific Computing: Third Edition*, 3rd ed. Cambridge University Press, 2007, ch. 13 Fourier and Spectral Applications, pp. 640–719.

[191] C. Bennett, L. J. Voss, J. P. M. Barnard, and J. W. Sleigh, "Practical Use of the Raw Electroencephalogram Waveform During General Anesthesia: The Art and Science," *Anesthesia & Analgesia*, vol. 109, no. 2, pp. 539–550, August 2009.

[192] Intel Corporation, "Corporate Website," http://www.intel.com/ (last accessed July 2011).

[193] Advanced Micro Devices, Inc., "Corporate Website," http://www.amd.com/ (last accessed July 2011).

[194] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK benchmark: past, present and future," *Concurrency and Computation: Practice and Experience*, vol. 15, no. 9, pp. 803–820, 2003.

[195] Green500.org, "The Green 500 List of Supercomputers," http://www.green500.org/ (last accessed July 2011).

[196] The Portland Group, "PGI CUDA-x86," http://www.pgroup.com/resources/cuda-x86.htm (last accessed July 2011).

[197] Khronos OpenCL Working Group, *The OpenCL 1.1 Specification*, June 2011.

[198] *NVIDIA CUDA C Programming Guide Version 4.0*, NVIDIA® Corporation, 2011, http://www.nvidia.com/ (last accessed July 2011).

[199] C. J. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. D. Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang, "Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, Chamonix, France, 2011.

[200] Intel Corporation, "Intel SPMD Program Compiler," http://ispc.github.com/ (last accessed July 2011).

[201] LLVM Project, "The LLVM Compiler Infrastructure," http://llvm.org/ (last accessed July 2011).

[202] A. P. Gerdelan, K. A. Hawick, **A. Leist**, and D. P. Playne, "Simulation Frameworks for Virtual Environments," in *Proc. International Conference on Internet Computing (ICOMP'11)*, no. ICM4087, Las Vegas, USA, July 2011.

[203] K. A. Hawick, **A. Leist**, D. P. Playne, and M. J. Johnson, "Comparing Intra- and Inter-Processor Parallelism on Multi-Core CellBE Processors for Scientific Simulations," in *Proc. Parallel and Distributed Computing and Systems (PDCS 2010)*, 2010.