Res. Lett. Inf. Math. Sci., 2010, Vol. **14**, pp. 25–77
Available: http://www.massey.ac.nz/massey/learning/departments/iims/research/research-letters/

Mixing Multi-Core CPUs and GPUs for Scientific Simulation Software

K.A. Hawick, A. Leist and D.P. Playne
Computer Science
Institute of Information & Mathematical Sciences
Massey University at Albany, Auckland, New Zealand
Email: {k.a.hawick | a.leist | d.p.playne}@massey.ac.nz

Recent technological and economic developments have led to widespread availability of multi-core CPUs and specialist accelerator processors such as graphical processing units (GPUs). The accelerated computational performance possible from these devices can be very high for some applications paradigms. Software languages and systems such as NVIDIA's CUDA and Khronos consortium's open compute language (OpenCL) support a number of individual parallel application programming paradigms. To scale up the performance of some complex systems simulations, a hybrid of multi-core CPUs for coarse-grained parallelism and very many core GPUs for data parallelism is necessary. We describe our use of hybrid applications using threading approaches and multi-core CPUs to control independent GPU devices. We present speed-up data and discuss multi-threading software issues for the applications level programmer and offer some suggested areas for language development and integration between coarse-grained and fine-grained multi-thread systems. We discuss results from three common simulation algorithmic areas including: partial differential equations; graph cluster metric calculations and random number generation. We report on programming experiences and selected performance for these algorithms on: single and multiple GPUs; multi-core CPUs; a CellBE; and using OpenCL. We discuss programmer usability issues and the outlook and trends in multi-core programming for scientific applications developers.

Keywords: multi-core; accelerators; GPU; CUDA; CellBE; OpenCL; data parallelism; thread model.

Contents

1 Introduction

Power consumption and physical size constraints have led to a "slowing down of Moore's Law" [1,2] for processing devices, in this, the first decade of the millennium. A major consequence is that parallel computing techniques – such as incorporating multiple processing cores and other acceleration technologies – are increasing in importance [3] both for high performance computing systems but also for game machines, desktop and medium-range computing systems. This is an exciting time for parallel computing but there are some as yet unsolved challenges to be addressed for parallel applications programmers to be able to fully exploit multi-core and many-core computing devices [4, 5].

In this article we consider the challenge of using current and emerging multi- and many-core accelerator processing devices for application in scientific simulations. We consider three algorithmic areas we believe are representative of this applications area – solving partial differential equations with finite differencing techniques; analysis algorithms for measuring complex networks; and pseudo-random number generation. These areas are not only relevant for typical high performance

simulations but are also highly relevant to computer games and other virtual reality simulations which are important economic drivers for the accelerator device technology market.

We are particularly interested in numerical algorithms, for which a road map of ideas and developments is emerging [6]. In the course of our group's research on complex systems and simulations [7,8] we have identified a number of scientific application paradigms such as those based on particles, field models [9], and on graph network representations [10]. The use of accelerators for the paradigm of N-Body particle simulation [11] has been well reported in the literature already [12, 13].

Other areas of relevance we have studied but do not have space to report on in this present article include: traditional models of phase transitions such as the Ising model [14] and the use of accelerated supercomputers for studying large simulated systems; uses of accelerators and multi-core devices for computational finance Monte Carlo Finance [15, 16]; and other Monte Carlo applications such as generating networks and measuring their properties [17]. Many of these are potentially hybrid applications that do not fit a single program model paradigm but which need the full performance available from processors and any associated accelerators.

1.1 Multi-Core Processing

In recent years there has been a steady increase in the number of processing cores available on a modern processor chip. Dual core systems are now the norm at the time of writing and quad core systems are already becoming economically viable. Hardware vendors have announced and promised economically affordable six and eight core conventional CPU systems and it seems likely this trend will continue. However with increasing concerns over computer power consumption, physical constraints on power dissipation and approaching spatial limitations on present chip electronics fabrication technology there has been a perceptible "slowing down of Moore's law" at least in terms of conventional approaches using uniform and monolithic core designs.

However, some of the most exciting recent accelerated approaches to improving processor performance include use of much larger numbers of simpler specialist processing cores as accelerator supplements to the main processor or processing core. One particularly noteworthy approach is the use of Graphical Processing Units (GPUs) with **very many** cores and which can be used as high performance computing devices in their own right, accelerating computations orchestrated by a conventional CPU program. Another exciting approach is that of CPU designs with more than one core design on a chip. This approach is embodied by the Sony/Toshiba/IBM (STI) consortium's Cell Broadband Engine (CellBE) processor design with its conventional PowerPC main core(PPE) supplemented by eight Synergistic Processing Element(SPE) or auxiliary cores.

Graphical Processing Units [18] are an attractive platform for optimising the speed of a number of application paradigms relevant to the games and computer-generated character animation industries [19]. Chip manufacturers have been successful in incorporating ideas in parallel programming into their family of CPUs, but for backwards compatibility reasons it has not been trivial to make similar major advances with complex CPU chips. The concept of the GPU as a separate accelerator with fewer legacy constraints is one explanation for the recent dramatic improvements in the use of parallel hardware and ideas at a chip level.

GPUs can be used for many parallel applications in addition to their originally intended purpose of graphics processing algorithms. Relatively recent innovations in high-level programming language support and accessibility have caused the wider applications programming community to consider using GPUs in this way [20–22]. The term general-purpose GPU programming (GPGPU) [19] has been adopted to describe this rapidly growing applications level use of GPU hardware.

We explore some detailed coding techniques and performance optimisation techniques offered by the Compute Unified Device Architecture (CUDA) [22] GPU programming language. NVIDIA's CUDA has emerged within the last year as one of the leading programmer tools for exploiting GPUs. It is not the only one [23–26], and inevitably it builds on many past important ideas, concepts and discoveries in parallel computing.

1.2 Re-embracing Parallelism

Many of the data-parallel ideas currently in vogue for many-core devices such as GPUs are not new but owe their origins to important massively parallel and data-parallel supercomputer architectures such as the AMT DAP [27] and the Thinking Machines Connection Machine [28, 29]. There have been successful attempts at portable embodiments of data-parallel computing ideas into software libraries, kernels [30,31] and programming languages [32–35]. Much of this software still exists and is used, but was perhaps somewhat overshadowed by the subsequent era of coarser-grained multi-processor computer systems – perhaps best embodied by cluster computers, and also by software to exploit them such as message-passing systems like MPI [36]. We have since seen some blending together of these ideas in software languages and systems like OpenMP [37]. Interestingly these ideas too are also finding their way onto multi-core devices and can also be used in the context of multiple accelerator devices including GPUs, CellBE, and other multi-core processors.

The economics of mass production and consumer demand for faster desktops and computer games machines means that the high performance computing market is also strongly driven by these devices. There are many possibilities for combining multiple devices and for building hybrid computing systems that can effectively ride the market forces to attain good processing costs for scientific application areas. This however means that in addition to the challenges of programming individual devices, there are also challenges in devising software models to combine them together effectively into clusters and supercomputers [38, 39].

We are interested in how an application programmer makes use of the processing power of these devices and how standards and architectural models will evolve to make this programming effort easier in the future. In this article we use our three applications areas to explore the use of threading and master/slave accelerator device control software architectures for making use of: single and multiple GPUs; CellBEs; and of course conventional multi-core CPUs for purposes of comparison.

1.3 Article Organisation

We report on programming fragments and experiences in terms of ease of programming as well as performance analysis. We consider high-level optimisations for individual cores - such as the SIMD and vector instructions, as well as multi-threading approaches to MIMD parallelism. In addition to Intel's Threading Building Block (TBB) system; NVIDIA's Compute Unified Device Architecture (CUDA) and conventional approaches such as PThreads, we report on some preliminary work with the Open Compute Language (OpenCL).

In Section 2 we review some of the important ideas and systems currently prevalent for accelerator device approaches to parallel computing, and in particular we discuss: multi-core CPU issues (Section 2.1); GPU programming architecture ideas (Section 2.2); Multi-GPU issues (Section 2.2.4) and CellBE related issues in Section 2.3.

We report on detailed algorithmic and programming development work in Section 3 where we focus on three broad applications areas: solving regular partial differential equations (Section 4); computing clustering coefficients for complex graph networks (Section 5); and pseudo random number generation (Section 6). We provide some specific code and pseudo-code fragments and implementation ideas as well as performance data on various devices. In Section 7 we discuss some of the overall issues for programming accelerator devices and hybrid system solutions and we offer some tentative conclusions concerning the future of unifying software models like OpenCL in Section 8.

2 Review of Accelerator Platforms

The concept of accelerating a (central) processor's performance with a specialist device is not a new one. Two interesting examples of this approach within living memory are the 8087 FPU

floating point accelerator chip used by the Intel 8086 microprocessor to accelerate floating point calculations; and the use of specialist array processors such as the original ICL Distributed Array Processor (DAP) which was in effect a multi-processor accelerator for a conventional mainframe.

Present use of additional processor cores can take the form of: completely separate chips such as GPUs that can contain very many specialist cores and are not dissimilar to the DAP SIMD approach in architecture; the addition of uniform extra cores within the existing main CPU, such as the Intel and AMD and other mainstream CPU vendor approaches; or the form of non-uniform accelerating cores supplementing a conventional core such as the case of the synergistic processing elements (SPEs) in the Sony/Toshiba/IBM Cell Broadband Engine.

As we discuss in this paper all these approaches have merit and can produce very good performance results - but they are all different architecturally and unfortunately need to be programmed differently at an applications level. This is a significant obstacle to applications programmers being able to gain optimal device performance. Therefore in addition to considering the details of the hardware solutions that are emerging in the field of performance accelerators, it is also very interesting to review some of the detailed approaches that are emerging in hardware and in software tools to program such devices, and also how to integrate together hybrid solutions and unify ideas into a common model.

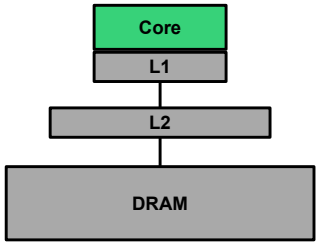

Figure 1: Single-Core CPU Architecture, against which many applications programmers optimise their software.

Figure 1 shows what has become the standard model architecture for nodes and for which target model much present software is optimised. It is useful to contrast this with architectural models that applications programmers must be aware of for the accelerator platforms we discuss below.

We review some specifics on: multi-core CPUs (Section 2.1); GPUs (Section 2.2); use of multiple GPUs (Section 2.2.4); the STI Cell Broadband Engine (Section 2.3) and some other approaches such as the use of reconfigurable hardware accelerator devices such as field programmable gate arrays (FPGAs) in Section 2.4.

2.1 Multi-Core CPU

Todays multi-core CPUs make it necessary to re-think the way we develop applications software [40]. New hardware generations do not come with significantly higher clock frequencies like they used to in the past, but instead give us more processing cores that can be used concurrently. In fact it is necessary for applications programmers to explicitly use concurrency to attain anything even close to the peak performance of modern processors. To make use of all the available processing power for computationally intensive tasks, the tasks need to be split up into sub-tasks, which can then be executed on different cores. There are a number of different approaches to parallel programming for multi-core CPUs, ranging from low-level multi-tasking or multi-threading, over high level libraries that provide certain abstractions and features that attempt to simplify concurrent software development, to programming languages or language extensions that have been developed specifically with concurrency in mind [41, 42].

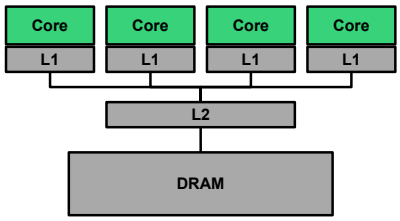

Figure 2: Multi-Core CPU Architecture, four cores with their own L1 caches and a shared L2 cache.

The explicit use of system-threads may have the highest performance potential, but it can be a complex task. It is often not easy to keep all processing cores equally utilised and to keep the overhead introduced by thread management and possible inter-thread communication to a minimum. A low-level threading library, for example the POSIX Thread (PThread) library [43], provides an API for creating and manipulating such platform-dependent threads.

A higher level approach to parallelising computationally intensive work that takes care of some of the tasks that the developer would otherwise have to deal with is the use of a parallel library like the Intel® Threading Building Blocks (TBB). This particular C++ library automatically maps logical parallelism onto low-level threads. It emphasises data-parallel programming, executing different parts of a collection on multiple threads using a task scheduler to distribute the work and keep all the processing cores busy. Like the C++ Standard Template Library (STL), TBB makes extensive use of generic programming.

TBB makes it relatively easy to execute loops whose iterations are independent from one another in parallel, invoke functions concurrently, or create and run parallel tasks. It also provides a number of concurrent containers (hash map, queue, vector), different mutex types, atomic operations, as well as scalable memory allocators to name just a few of TBB's features [44].

Some programming languages come with language-level features for concurrent programming. Java [45], for example, has the `synchronized` keyword which enforces mutually exclusive access to blocks of code. Erlang [41], on the other hand, provides language-level features for creating and managing processes and uses message passing for inter-process communication, thus avoiding the need for locks.

2.2 Graphical Processing Units

In recent years GPUs have emerged as one of the most favourable and popular accelerator devices available. These GPUs were originally developed to keep up with the increasing demands of the gaming industry which requires them to render increasingly detailed, high-quality computer graphics in real time. As the demands of the computer games increased faster than processor clock speeds, GPU developers adopted the multi-core architectures currently emerging in modern CPU design to keep up with requirements. Over time these GPUs have developed into more general, highly-parallel, many-core processing architectures. A typical high-range GPU contains between 128-480 processors (as compared the the 1-4 cores generally available in CPUs).

As these GPU architectures have moved towards more general processing designs, a number of libraries have been developed that allow users to write non-graphics applications that are computed on GPUs. This is known as General Purpose computation on GPUs or GPGPU. The development of GPGPU libraries such as NVIDIA's CUDA [46], BrookGPU [23], ATI's Stream SDK [47] and sh [25] have made GPGPU applications increasingly easy to develop. They allow developers to implement GPGPU applications in a C-style language. The libraries act as extensions to the C language and handle the interaction between the host CPU and the GPU.

GPUs are very effective at accelerating a wide range of scientific applications. The performance

benefits they offer depends on the application's suitability for parallel decomposition. Much of the research into computer science over the last several decades has been focused on developing methods for parallelising applications. Thus there exist many well-known methods for decomposing applications onto the parallel GPU architecture.

GPUs have data-parallel architectures reminiscent of parallel computers from the 80's such as the DAP [48,49] and the CM [28]. As most of the parallel computing research over the last decade has been focused on developing applications for distributed architectures (i.e. cluster and grid computers). A certain degree of adaptation is required to transfer those parallel designs to the data-parallel architecture of GPUs. Most of the research into GPU programming involves finding the optimal way to solve a problem on the data-parallel architecture while making the best use of the optimisations specific to the GPU architectures.

One of the most inviting features of the GPU is the ease of accessibility. GPUs are relatively cheap and provide easy access to a highly-parallel architecture. This along with easy to use GPGPU APIs allow a large user-base access to high-power parallel applications with little start-up cost. Their computing power to cost ratio is also highly desirable, a $250 (US) graphics card has the potential to provide over 100x speed-up over a conventional single-core CPU [8].

2.2.1 NVIDIA Tesla Architecture

Of the many GPGPU APIs available [23, 25, 46, 47], NVIDIA's CUDA stands out as the most developed and advanced API. As this library has been developed by NVIDIA, it only operates on NVIDIA GPUs. Our development of GPGPU applications uses this CUDA API and is thus limited to NVIDIA graphics cards. While we specifically discuss the NVIDIA Tesla architecture, ATI's GPU architecture is very similar and many concepts are transferable between the two vendors' products.

Tesla architecture GPUs contain a scalable array of multi-threaded processing units known as Streaming Multiprocessors (SMs). These SMs each contain eight Scalar Processor (SP) cores which execute the actual instructions. Each SM can perform computation independently but the SP cores within single multiprocessor all execute instructions synchronously. NVIDIA call this paradigm SIMT or Single Instruction Multiple Thread [46].

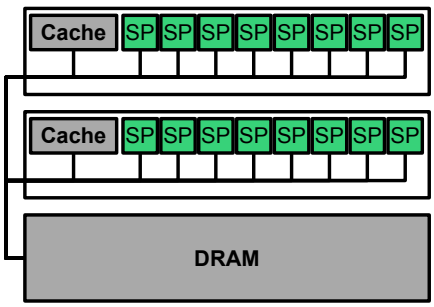

Figure 3: A Tesla architecture GPU with two Streaming Multiprocessors each containing eight Scalar Processors. This architecture can be easily extended to contain many more multi-processors.

Figure 3 emphasises the hierarchical nature of the multi-processors and cores in the Tesla architecture. This architecture is designed to support a very fine-grained parallelism. GPU applications perform best when split into many threads, each of which performs some smallest unit of work. For example, image processing tasks can be split by assigning one thread for each pixel. As the management, scheduling and execution of threads on the processor cores is performed in hardware on the GPU, managing a large number of threads presents no additional overhead [46].

2.2.2 NVIDIA Tesla Memory

Tesla GPUs contain several different types of memory. As there is no implicit caching on GPUs, memory access can be the limiting factor for many GPU applications. CPUs contain several levels of implicit caching to provide the best memory access speeds possible. GPUs do not contain nearly the same level of memory caches, instead they provide smaller, explicitly accessible caching methods (see Figure 3 as compared to Figure 1 for a comparison between the two architectures). The GPU memory types accessible from CUDA are as follows [46]:

- **Registers** - Used to store the local variables belonging to each thread. Provided there are at least 192 active threads, the access time for registers is effectively zero extra clock cycles. If the registers cannot store all the local variables for the threads, Local Memory is used. Stored on-chip.
- **Local Memory** - Actually stored in Global memory, used to store any local variables that cannot fit into the registers. Access times are the same as Global memory. Stored on-device.
- **Global Memory** - The largest section of memory. Only memory type that the CPU can read from and write to. Accessible by all threads, also has the slowest access times (400-600 clock cycles). Total transaction time can be improved by coalescing - 16 sequential threads that access 16 sequential and word-aligned memory addresses can coalesce the read/write into a single memory transaction. Stored on-device.
- **Shared Memory** - Read/Write memory that can be used to share data between threads executing on the same multiprocessor (i.e. in the same block). Provided that the threads access different memory banks, the memory transactions are as fast as reading from a register. Stored on-chip.
- **Texture Memory** - Cached method of accessing Global memory. Global memory bound to a texture will only be read if the desired value is not already stored in the cache. If so the cache is reloaded with that value and the values surrounding it in the spatiality defined by the texture dimensions. On a cache hit, the transaction is as fast as reading from registers. Stored on-chip.
- **Constant Memory** - Another cached method of accessing Global memory, a Global memory read is only required in the event of a cache miss. If all threads read the same value from constant memory, the cost is the same as reading from registers. Stored on-chip.

As these different types of memory must be used explicitly by the developer, a great deal more consideration of memory access is required when developing a GPU application as compared to a CPU implementation. Correct use of these memory types is vital to the performance of a GPU application [8]. Correct use is not limited to the type of memory use but also the access patterns and thread block arrangements. The memory access patterns are dependent on the organisation of threads into blocks and thus decisions about block sizes and shapes must be carefully considered.

2.2.3 CUDA Threads

When developing a CUDA application, the algorithm must be decomposed into a number of SIMT threads. The actual scheduling, management and execution of these threads is performed by the GPU itself but a certain degree of control is still available to the developer. A CUDA application has a number of threads organised into blocks, which are all arranged into a grid (See Figure 4). A block of threads has three dimensions (x,y,z) and can contain no more than 2^9 threads. A grid has two dimensions (x,y) and has a maximum size of 2^{16} in each dimension. CUDA allows the user to control the arrangement of threads into blocks and the blocks into the grid. The developer can control how these threads are arranged in order to make optimal use of the on-chip memory in

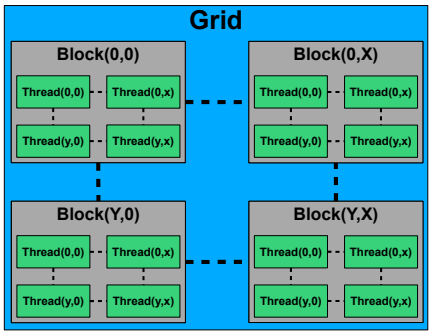

Figure 4: The hierarchy of CUDA threads, this grid contains (X, Y) blocks, each of which contains (x, y) threads.

the multiprocessors (e.g. block widths should most often be a multiple of 16 to allow for coalesced global memory).

When a CUDA application is executed, each block is assigned to execute on one multiprocessor. This multiprocessor will manage and execute the threads within the block on its SPs. While the block arrangement does not have a significant impact on the execution of the threads on the multiprocessors, it strongly affects the manner in which threads access memory and use the memory optimisations available on the GPU. Threads should be arranged into blocks such that the maximum amount of data can be shared between threads in shared/texture/constant memory and that any unavoidable global memory accesses be coalesced.

2.2.4 Multi-GPU

While GPUs are computationally powerful, many problems are so large that we wish to compute them on more than one GPU. Each GPU performs its own thread management automatically in hardware. This management is restricted to the individual GPUs, thus additional management is required when splitting an application across multiple GPUs. CPU level code is required to decompose tasks across multiple GPUs and deal with the challenges of data communication, and synchronisation.

Many motherboards have several PCI-E x16 slots and are thus capable of hosting multiple GPU devices. Some graphics cards such as the GeForce GTX 295 or the GeForce 9800 GX2 contain two GPU cores on a single graphics card. In such configurations, multiple threads are required to interact with the GPU devices. While single-core machines are capable of managing multiple threads interacting with multiple GPUs, it is obviously advantageous to have as many CPU cores as there are GPU devices. Figure 5 shows a multi-GPU configuration with a dual-core CPU controlling two GPUs on separate PCI-E x16 slots.

Communicating data between CPU threads controlling GPU devices is not as simple as communication between threads. CPU processor cores share memory and L2 cache and simply have their own L1 cache (See Figure 2). Data that needs to be communicated between CPU threads can be performed simply by writing to memory, synchronising and then reading. This is not quite as simple when communicating data between multiple GPUs. The CPU threads can communicate information as usual but it requires that the data first be retrieved from the GPU device, exchanged and then written back into the appropriate device.

These systems encounter the usual problem of communication versus computation time. For multiple GPUs to be used efficiently, the time taken to perform the computation should be significantly larger than the time required to communicate the necessary data between them. For multi-GPU systems, this requires very large problem sizes as each GPU is in itself a powerful,

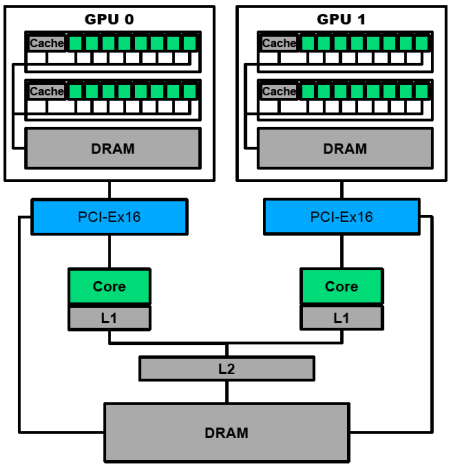

Figure 5: A multi-GPU machine containing two GPUs each hosted on separate PCI-E x16 buses and managed by separate CPU cores.

highly-parallel computer. The best method for improving this communication to computation ratio is obviously dependent on the problem; however, CUDA does provide a very useful piece of functionality that can be useful for many applications.

CUDA has support for asynchronous execution and data-transfer [46]. This type of execution can allow the GPU to execute threads while at the same time the CPU can transfer data in and out of the GPU device. Using this functionality can allow the communication to be performed at the same time as the computation which, in the right conditions, can allow the effective communication time be reduced to zero. Using this functionality has been shown to make the use of multiple GPU devices scale effectively (See Section 4.9).

Such Multi-GPU machines have been shown to produce good scalability for a variety of problems such as saliency map models [50], RNA folding algorithms [51] and Smith-Waterman sequence alignment [52] etc. The scalability of Multi-GPU machines is limited by the number of GPUs that can be hosted on a single machine. Some motherboards are capable of hosting up to four graphics cards which could potentially contain two GPU cores, limiting the maximum number of GPUs on a single machine to eight.

Such machines would require large power supplies but it is still worth considering. How well an application can be divided across eight GPU cores is, as always, dependent on the application. If no communication between the GPUs is necessary, then we can expect a near linear speedup, however as the communication between the GPUs increases we expect the speedup to decay. In these situations optimal use of functions such as asynchronous memory copy and execution is vital to attaining good scalability across the multiple GPU cores.

2.2.5 GPU-Accelerated Clusters

Another option for multi-GPU computation is to host the GPUs on multiple nodes spread across a cluster or other distributed device. These GPUs then take the form of accelerators for the individual nodes on the cluster. GPU accelerated clusters simply combine two technologies, the relatively new GPGPU libraries and the well-developed distributed computing frameworks such as MPI [36]. GPU clusters with promising scaling results have been described for a number of applications - Finite Differencing [53], Biomedical Analysis [54], lattice Boltzmann model [55], Fluid Dynamics [56].

The CPU cores in these cluster nodes perform inter-node communication using common cluster

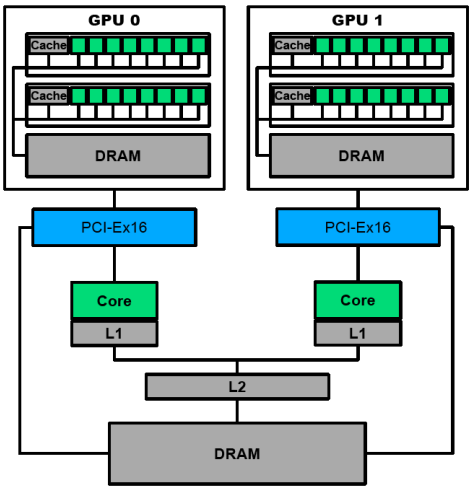

Figure 6: A diagram showing a GPU cluster configuration containing N GPU-accelerated nodes. In this cluster each node contains a single GPU.

communication methods such as MPI [36] and use a GPGPU library such as CUDA [46] to perform calculations on the GPU. The nodes on these clusters are not limited to a single GPU and can in fact consist of a distributed architecture of multi-GPU nodes. In such cases there is communication between the GPUs in the same node via the host memory and inter-node communication across the cluster communication network.

Such GPU-accelerated clusters face more challenges in terms of achieving a high computation-to-communication ratio. While the GPUs can be utilised to accelerate the nodes of the cluster to reduce computation time, they can provide little benefit in terms of improving the inter-node communication speeds. This issue has been documented by GPU cluster developers [57]. The GPUs do not reduce the performance of the cluster and are almost always successful in accelerating it [55–57]; however, the limitations of the node communication infrastructure become more apparent.

Like multi-GPU machines, GPU-accelerated clusters can make use of CUDA's asynchronous memory copies and execution to perform communication while the GPU continues to perform calculations. This can be used effectively with MPI's asynchronous Send/Recv to communicate data between GPUs on the same node and between nodes in the cluster without the GPUs being left idle.

2.3 Cell Broadband Engine

The consortium of Sony, Toshiba and IBM (STI) produced a multi-core processor chip that is known as the Cell Broadband Engine (CellBE) [58] and which is now widely known as the processor in the Sony PlayStation 3 (PS3) games machine. The CellBE comprises a PowerPC processing core (PPE) but is unusual in that it is supported by eight "synergistic processing elements" (SPEs) that provide additional performance acceleration [59]. A number of other products now have processors based on the PPE component of the CellBE including the Microsoft Xbox 360 games engine, and the CellBE itself is also available in blade systems. The CellBE design was apparently driven by silicon space and also power consumption considerations [60], and this design paradigm – of a relatively conventional main core supported by specialist processing accelerator cores – may become more prevalent as chip vendors attempt to combine large numbers of cores with other design criteria like minimal power consumption.

The CellBE as supplied on the Sony PlayStation 3 is accessible to a Linux programmer through

the STI software development kit and a compilation system based on the Free Software Foundation GNU gcc compiler [61]. Scarpino's book [59] describes how to program the CellBE on the PS3 in more detail, but in brief it is possible to program both the PPE and the SPE using appropriate libraries called from what are essentially C/C++ application programs. The PS3 implementation uses a hypervisor arrangement so the Gaming Operating system supplied by Sony supervises a running Linux, but only six of the available eight SPEs are exposed to the Linux programmer. Similarly some additional features of the CellBE such as the graphics and storage interface are not directly controllable.

Nevertheless, the PS3/CellBE/Linux combination provides a remarkably cost effective developmental platform for development of application programs for the CellBE and has allowed us to investigate some of the relevant features for comparison with the other software and hardware platforms discussed in this article.

As Dongarra and others have reported [62,63], the PS3 is an appropriate platform for some but not all scientific applications. Although it is possible to use the PS3 network interface to cluster multiple PlayStations together, in this present paper we report only on performance observations using a single CellBE in a single PS3. Our main observation is that, at least at the time of writing, some considerable expertise, time and effort is required on the part of the applications programmer to make full use of the processing power of the CellBE. In fairness this is true of parallel programming generally. If programmer use of parallelism is necessary to attain optimal performance on future processors that have of necessity been designed to have low power considerations, then better software tools and/or programmer expertise will be required.

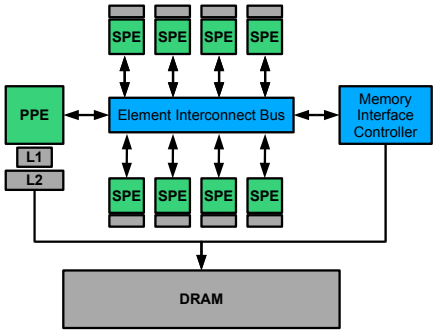

Figure 7: Cell Processor Architecture with eight SPEs. The PPE and SPEs can communicate via the Element Interconnect Bus. The SPEs access memory via the Memory Interface Controller.

Figure 7 summarises the main points of the CellBE architecture. The PPE unit has some common ancestry with IBMs's other PowerPC chip designs and is a powerful core in its own right. It is supported by eight SPE cores each of which has some dedicated memory (256kB) and which are accessible through a bi-directional dual ring interface. The software development kit and associated library provide a great many library calls to communicate between different core components of the CellBE. Mechanisms include mailboxing and signalling as well as bulk data transfers using a Direct Memory Access (DMA) support infrastructure.

Generally, it is not difficult for an applications programmer to develop a master/slave software model to delegate computations to the SPE from the PPE. A Monte Carlo example for instance works well, providing that tasks for the slave SPE's fit in their own memory. Otherwise, some care in needed to barrier-synchronise their access to global memory to avoid contentions.

A number of software packages are relatively trivially ported to work on the PPE main core, treating it as any other PPC instructions set architecture. For example the Erlang parallel language system [41] worked on the PPE trivially. As a very rough indicator, we found the PPE by itself gave a performance equivalent to around half of that of a single CPU core on a contemporary desktop.

This is quite impressive given the CellBE was designed with completely different optimisation criteria. To obtain high performance from the CellBE it is necessary to a) delegate computational work to the SPEs and b) make use of the vector instructions on both the PPE and SPEs.

The PPE and SPEs are capable of processing vector data types performing four instructions synchronously. To make full use of the SPEs, the programmer must use `spu_add, spu_mul, etc` to transform calculations into vector form. Other performance optimisation techniques available are enqueued DMA access (IE fetch memory while computing); bi-directional use of the IE ring; use of the memory flow controller.

2.4 Other Accelerator Hardware Approaches

Another approach to gaining applications performance involves field programmable gate arrays (FPGAs) [64]. These reconfigurable devices can be programmed at a very low level for arbitrary logical operations and can thus perform specific and particular applications algorithms at high speeds. Modern FPGA devices operate around power dissipation levels of around 25 watts compared with levels around 60-70 for heterogeneous core devices like the CellBE or in excess of 100 watts for conventional monolithic multi-core processors.

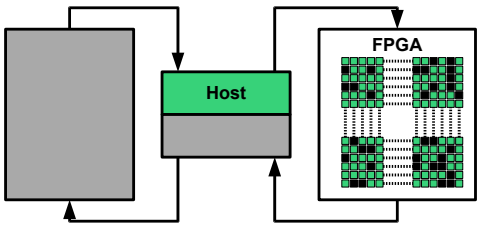

Figure 8: A host device utilising an FPGA to accelerate specific calculations.

Figure 8 shows the block outline of an FPGA program as it might be used to accelerate a particular operation such as an image processing kernel or a Fourier Transform. A modern FPGA is typically programmed using design tools that can draw on libraries of the low level logic pattern – effectively small processing cores – that can perform a particular algorithmic aspect of an application. FPGAs tend not perform well with non-floating point calculations or with less general specific operations. As affordable FPGAs become larger and have larger number of primitive gates available however, it is likely that general purpose floating point cores may be readily embeddable in FPGA program designs.

FPGAs have traditionally been used for signal and image processing applications but have been more recently successfully used for financial simulations and calculations [65]. Although their power characteristics, and performance/cost ratios are potentially very attractive, the main obstacle to wider applications deployment of FPGAs lies in the difficulties (and hence costs) of programming them at an applications level. Other issues facing applications developers include the considerable variation in the data-types and bit-lengths of operating words and buses for different FPGAs and families of devices. Some portability for applications codes is likely needed to lower the cost of FPGA software development.

Vendor specific languages such as Mitrion-C [66] and its associated toolset provide a potential route for applications programmers to develop relatively high level applications programs that can be automatically decomposed into VHDL logic patterns for programming FPGAs. Mitron supply a Turing-complete library of predesigned processing core components that can be combined to support applications. This approach of writing applications programs that can be used as a hardware independent "device middleware" is a promising one. We speculate that languages such as OpenCL may play a similar role for other compute devices in the future.

We have attempted to review the main classes of accelerator devices and their characteristics. We have not covered all vendors and have necessarily been restricted by our own time and hardware budgets. AMD and other vendors also produce powerful and attractive multi-core processors. The Jaguar Supercomputer [39] at the Oak Ridge National Laboratory in the USA is based around over 180,000 AMD Opteron processing cores.

Other systems such as the the Roadrunner Supercomputer [38] at the Los Alamos Laboratory in the USA is based around use a hybrid of different multi-core processors including IBM PowerPC designs similar to the CellBE heterogeneous design. There are also a number of new devices forthcoming such as Intel's Larrabee and AMD's Fusion multi-core processors that may have different optimisation characteristics than those discussed below. We discuss the CellBE as it was accessible to us within Sony PlayStation 3's, but it is also possible to obtain specialist accelerator devices based around this processor, without the six SPE restrictions. We discuss the SIMD and array-processing aspects of NVIDIA's GPUs but devices such as ClearSpeed [67] which are based around a multi-threaded array processor (MTAP) approach also find uses as specialist accelerators in a range of different compute systems and applications [68, 69].

The future of accelerated processing units is likely to prosper, albeit with the expected continued push to incorporate the successful cores or accelerator designs on to future multi-core designs that are better integrated. It is still non-trivial to compare the available groups of accelerator devices without focussing on specific implementation and performance measurement details. We attempt this in the next sections.

3 Performance Experiments

We provide a comparison between the different accelerator devices we have discussed by presenting the implementation of several scientific applications on each device and comparing their results. The scientific applications we have chosen as representative of most scientific applications are: Finite Differencing (Section 4), Graph Clustering (Section 5) and Random Number Generation (Section 6).

Finite differencing algorithms operate on the cells of a regular mesh. This regularity makes them particularly well suited for SIMD architectures. Furthermore, data dependencies exist between neighbouring cells, which means that hardware support for spatial locality in the data, for example texture memory in CUDA devices, is very beneficial.

Algorithms that iterate over arbitrary graphs, on the other hand, do not benefit as much from SIMD architectures and spatial locality in the data, as the graph structure is not known beforehand. But with the right data layout and access strategies, they can nevertheless perform well even on these kinds of accelerators.

After these two somewhat opposing scenarios are covered, we look at one of the most widely used facilities in scientific simulations, namely random number generation. A parallel scientific application often uses a separate stream of random numbers for each thread in order to ensure that the simulation is repeatable. This is exactly what most of the implementations described in this article do as well. Another advantage is that independent generators do not introduce communication overhead between the threads.

The platform we have used for all performance experiments except for the CellBE algorithms runs the Linux distribution Kubuntu 9.04 64-bit. It uses an Intel® Core™2 Quad CPU running at 2.66GHz with 8GB of DDR2-800 system memory and an NVIDIA® GeForce® GTX 295 graphics card, which has 2 GPUs with 896MB of global memory each on board.

The platform used to run the CellBE implementations is a PlayStation®3 running Yellow Dog Linux 6.1. It uses a Cell processor running at 3.2GHz, which consists of 1 PowerPC Processor Element and 8 Synergistic Processor Elements, 6 of which are available to the developer. It has 256MB of system memory.

In view of the importance of presenting coding details but also trying to bring out compar-

isons between the different architectures and devices and software, we give algorithms in stylised pseudo-code with device-specific software library calls embedded in them. We believe this makes it easier to compare different device implementations than either verbatim program code or a terser mathematical notation would be.

4 Finite Differencing

Finite differencing methods can be used to numerically approximate solutions to n-dimensional systems described by a partial differential equation. They approximate the continuous space of some physical or mathematical system by a mesh of discrete cells. Sub-categories of finite differencing applications include field-equations and image processing (physical space is approximated by the camera as a mesh of pixels). Finite differencing applications perform many regular, local operations on the cells in the mesh.

These applications are easy to decompose for execution on parallel accelerator devices. The mesh can be split into sections and each one processed separately. This mesh separation can take a very coarse form (split the field into two for execution on two processors) right down to a very fine-grained decomposition (each section is one cell). The regular instructions and memory access patterns make them good candidates for many optimisations available on accelerator devices.

4.1 Neighbour Operations

The operations of finite differencing problems usually perform some calculation on how to change the value in the cell depending on the values of the cells surrounding it (the neighbouring values that must be accessed are called the memory halo of the problem). In general, more complex equations that require accesses to more neighbours will have slower simulation times due to the extra memory fetches and calculations. Figure 9 shows some common memory halos used by various finite differencing applications.

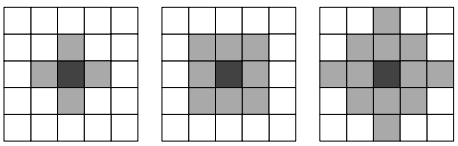

Figure 9: Some common memory halos used by finite-differencing applications.

This is especially true for accelerator devices such as GPUs. For devices with specific memory caches with limited sizes, the memory halo can have a large impact on performance. If the values required by neighbouring threads on a GPU cannot fit within the caches, cache thrashing can occur which may severely reduce performance.

The memory halo also has a significant impact on the parallelisation of a finite differencing problem. This halo determines the amount of communication required by nodes responsible for neighbouring sections of the field. A larger memory halo will mean less cells can be processed between communications and more data must be transfered between nodes. Simulations with large memory halos will parallelise less easily, as there is no way to reduce this memory halo this issue cannot be resolved but it is important to consider it in simulation design as different parallelisation methods may suit different halo sizes.

4.2 CPU - Sequential Algorithm

The most basic finite-differencing simulation is implemented on one core of a CPU. This application iterates through each cell in the mesh calculating the change in the value of the cell and writing

it to some output mesh. In this example we call the function `update` for each element in the mesh, the actual calculation and neighbouring cells accessed will depend on the finite-differencing equation being applied to the mesh. Algorithm 1 shows the CPU algorithm for iterating over a two-dimensional mesh.

Algorithm 1 Pseudo-code for the serial CPU implementation of the Finite Differencing Simulation. Computes some update function for a field of size (X, Y) over *no_steps* iterations.

```
declare input
declare output
initialise(input)
for t ← 0 to no_steps do
  for y ← 0 to Y do
    for x ← 0 to X do
      update(input, output, x, y)
    end for
  end for
  swap(input, output)
end for
```

The `update` function will read in the cell at position `(ix,iy)` and the required neighbouring values from the mesh `input`, calculate the new value of the cell and then write it into the mesh `output`. At the end of each step, the `input` and `output` meshes are swapped, the output from the last time-step becomes the input for the next and so on.

While the `update` function is equation specific, the code to iterate over the mesh will remain the same for almost all finite differencing simulations. This simulation is expected to be the slowest implementation as it only uses a single CPU core. We can accelerate this simulation by using multiple cores or accelerator devices.

4.3 Multi-Core: POSIX Threads

To compute a finite-differencing simulation on multiple cores we must make use of a multi-threading library to create and manage multiple threads. The most basic option is the well known POSIX thread library [43]. This library allows the developer to explicitly create, destroy and synchronise threads with the use of mutexes.

As this is a relatively low-level threading library it requires the developer to consider and explicitly define the behaviour of each thread including problem decomposition, task distribution, communication and synchronisation. This results in a rather longer development time and can be more prone to inconsistencies (especially with more complex simulations). The advantage of such a low-level library is that developer can make low-level tuning to extract the highest-performance possible.

The pseudo-code describing the algorithm of the main thread is shown in Algorithm 2 and the algorithm each of the threads performs is described in Algorithm 3.

The separate threads each execute `process_block` which will wait for the mutex `start` to be unlocked. It will then process a section of the field and unlock the mutex `stop`. The iterative loop in the main thread will loop `no_steps` times unlocking mutex `start` for each thread and then wait until all threads have unlocked `stop`. This ensures that the threads remain in sync and the field will be properly updated.

4.4 Multi-Core: Threading Building Blocks

Intel's Threading Building Blocks (TBB) can reduce the development time and improve the reliability of multi-threaded applications by abstracting common multi-threaded functions into the

Algorithm 2 Pseudo-code for the host code for the PThreads implementation of the finite-differencing simulation. Each thread launched will execute the function `process_block`, see Algorithm 3

```
declare input
declare output
declare THREADS ← {t_0, t_1 ... t_{n-1}}
initialise(input)
for all t_i ∈ THREADS do
  create(process_block, t_i)
end for
for t ← 0 to no_steps do
  for all t_i ∈ THREADS do
    signal start(t_i)
  end for
  for all t_i ∈ THREADS do
    wait stop(t_i)
  end for
  swap(input, output)
end for
```

Algorithm 3 Pseudo-code for the algorithm performed by each thread in the PThreads implementation of the finite-differencing simulation.

```
function process_block(id)
  while TRUE do
    wait start(t_id)
    height ← Y/n
    for y ← id × height to ((id + 1) × height) do
      for x ← 0 to X do
        update(input, output, x, y)
      end for
    end for
    signal stop(t_id)
  end while
end function
```

library itself. This can improve the development time of many parallel applications that can make use of these constructs. In this case we wish to iterate over a mesh and apply the field equation function to every element in the mesh. For this we can use TBB's `parallel_for` function which will split the task of iterating over the mesh across the cores available in the CPU.

The range of the mesh to be iterated over in parallel is defined by a TBB construct `blocked_range2d`. This specifies a X and Y range as well as a grain size for decomposition. The X and Y grain sizes specify the size of the blocks that the mesh will be decomposed into. This allows the user to control how finely grained the parallelism is. Algorithm 4 and Algorithm 5 show the code for TBB that iterates over a mesh in parallel. In this case the grain sizes specify that the mesh should be divided into 64x64 blocks of cells.

Algorithm 4 Pseudo-code for the host code of the TBB implementation of the finite-differencing simulation.

```
declare input
declare output
declare blocked_range2d br(0, Y, 64, 0, X, 64)
initialise(input)
for t ← 0 to no_steps do
  parallel_for(br, update)
  swap(input, output)
end for
```

Algorithm 5 The update function used by the TBB implementation of the finite-differencing simulation.

```
function update(blocked_range2d br)
  declare y_begin ← br.rows().begin()
  declare y_end   ← br.rows().end()
  declare x_begin ← br.cols().begin()
  declare x_end   ← br.cols(). end()
  for y ← y_begin to y_end do
    for x ← x_begin to x_end do
      update(input, output, x, y)
    end for
  end for
end function
```

4.5 GPU - CUDA

These applications are very well suited to GPU processing, GPUs were originally designed for generating computer graphics images. As finite-differencing meshes are very similar to images, and may in-fact be images, finite-differencing applications can make effective use of many of the optimisations available on the GPU.

When executing a finite-differencing application on a GPU, one thread is created for every cell in the mesh. This task decomposition method is similar to the `parallel_for` concept in TBB except that each thread is only responsible for a single cell (effectively a grain size of 1). Because GPU architectures are designed for managing, scheduling and executing thousands of threads on hardware, the program does not suffer any thread management overhead from such fine-grained parallelism.

Each thread will calculate the position of its element in the mesh from the thread's block and thread id. It can then access this cell's value and the values of the necessary neighbouring cells,

compute the new value of the cell and write it to some output mesh. Algorithm 6 shows a CUDA simulation that launches $X \times Y$ threads organised into `(16,16)` blocks. This block size provides a square shape for best caching effects, a width of 16 to allow coalesced global memory access and fits within the restriction 512 threads maximum per block.

Algorithm 6 Pseudo-code for the host portion of the CUDA implementation. One thread is created for each cell in the mesh and will perform the update calculation for that cell.

```
declare block(16,16);
declare grid(X/block.x, Y/block.y)
declare texture
declare input
declare output
initialise(input)
copy texture ← input
for t ← 0 to no_steps do
  do in parallel on the device using (grid, block):
      call update(output)
  copy texture ← output
end for
```

For the best performance, this implementation makes use of CUDA texture memory. This texture memory cache is very effective for applications performing local neighbour operations as it automatically caches values in the same spatial locality. This has been shown to provide the best performance for field equation simulations [70].

4.6 Multi-GPU - CUDA & POSIX Threads

The multi-GPU implementation is the most complex of our finite differencing simulations, this is due to the several levels of parallelism within the simulation. For ease of description we discuss a dual-GPU implementation, however the same concepts can easily applied to systems containing more GPUs. The first level of parallelism is splitting the field across multiple cores on the GPU. In this case we have two-GPUs so we split the field into two sections.

One thread is created for each section. These two threads will connect to one GPU each and compute the interactions of the cells in their half of the field. This computation is performed in parallel on the GPU (the second level of parallelism).

The cells on the border of each half-field must access cells from the other half-field to correctly calculate their change in value. Thus after each time-step the threads must communicate a border of cells to the other device. The width of this border depends on the memory halo of the finite-differencing equation. In this case we discuss an equation with a border of width 2.

The field is split in the y-dimension as this allows each field segment to be stored in contiguous memory addresses. This provides advantages when copying border information as it is faster to copy 2 segments of memory with $2x$ elements (the top and bottom borders each contain $2x$ elements) than copying $2y$ segments containing 2 elements.

One way to do this communication is to: compute the simulation for the entire half-field, stop, retrieve the bordering cells from the GPU, exchange these cells with the other thread and then continue. While this method would work, it leaves the GPU idle while the cells are retrieved from the device and exchanged between the threads. This is valuable computing resources going to waste.

A solution to this is to make use of asynchronous execution. Instead of computing the simulation all at once, we can launch two separate asynchronous kernels. The first computes the cells on the border (a small fraction of the total half-field size) while the other computes all of the other cells. As the first kernel will complete before the second (there are less cells in the border than in the

main body of the field), we can asynchronously copy the border cells out of the device and exchange them with the other thread while the main GPU kernel is still executing. When the second main kernel call finishes, the communication of the border cells is already complete and the next step can be started immediately. This allows us to hide the communication time and make the maximum use of the GPU computing resources.

For this implementation we have made use of POSIX threads for managing the multi-threading. Textures in CUDA must be referenced explicitly within the code, this requires the threads to initialise, load and use the same texture reference throughout the simulation. This application can be implemented using TBB threads, it would be effectively the same just with TBB syntax as opposed to PThreads syntax. The algorithm for the host thread can be seen in Algorithm 7 and the code for each thread can be seen in Algorithm 8.

Algorithm 7 Pseudo-code for the host algorithm for the PThreads & CUDA multi-GPU implementation of the finite-differencing simulation.

```
declare input
declare output
declare THREADS ← {t_0, t_1 ... t_{n-1}}
for all t_i ∈ THREADS do
  create(process, t_i)
end for
for t ← 0 to no_steps do
  for all t_i ∈ THREADS do
    signal start(t_i)
  end for
  for all t_i ∈ THREADS do
    wait stop(t_i)
  end for
end for
```

This implementation can provide a 2x (linear) speedup over a single GPU implementation provided that the problem is large enough that the communication time is less than is required for the main kernel call. A field length of $N >= 2048$ can make full use of both GPUs. For field sizes less than this, the multi-GPU simulation may run slower than a single GPU implementation as the communication between the two devices can take longer than the actual computation. This effect can be seen in Figure 10.

4.7 OpenCL

The currently available OpenCL implementations we have developed finite differencing simulations on are still pre-beta releases and thus we do not present performance data on their computational speed. We do however include a code example showing how the program iteratively computes simulation steps.

As OpenCL is device independent, the optimal sizes of `local` (which defines the grain size) should be set differently depending on the device. OpenCL running on a GPU will perform best with a grain size of 1 while larger sizes similar to those used by TBB will provide the best performance for multi-core CPUs. The code to iteratively call an OpenCL kernel on a field of size `(X,Y)` with grain size `16` can be seen in Algorithm 9.

OpenCL does not work with one single type of device and when writing code it is not known what type of device will be used to execute the code (although certain types of device can be requested). When the OpenCL application executes, it queries the machine to find OpenCL compatible devices that it can use. In the pseudo-code this is shown as `get_context()`, which will load a context on a OpenCL compatible device that it will use to execute the application.

Algorithm 8 Pseudo-code for the thread algorithm for the PThreads & CUDA multi-GPU implementation of the finite-differencing simulation. Calling update($stream, output, Y$) will launch a kernel in an asynchronous stream *stream*, writing data to *output* starting from row Y.

```
function process(size_t id)
  declare id2 ← other thread's id
  copy texture ← input
  while TRUE do
    wait start(t_id)
    declare block(16, 16)
    declare grid(X/block.x, 1)
    do in parallel on the device using (grid, block):
        call update(stream1, output, 2)
    do in parallel on the device using (grid, block):
        call update(stream1, output, Y2 + 2 − block.y)
    grid ← (X/block.x, Y − 4)
    do in parallel on the device using (grid, block):
        call update(stream2, output, 2 + block.y)
    synchronize (stream1)
    copy swap[id2][0] ← output[2 × X], 2 × X
    copy swap[id2][2 × X] ← output[Y2 × X], 2 × X
    signal copy(t_id2)
    wait copy(t_id)
    copy output[(Y2 + 2) ∗ X] ← swap[id][0], 2 × X
    copy output[0] ← swap[id][2 × X], 2 × X
    synchronize (stream2)
    copy texture ← output
    signal stop(t_id)
  end while
end function
```

Algorithm 9 Pseudo-code for an OpenCL finite-differencing implementation.

```
  declare local ← {16,16}
  declare global ← {X, Y}
  declare context
  declare kernel
  context ← get_context()
  kernel ← load_kernel()
  for n ← 0 to no_threads do
    set kernel argument(kernel, 0, input)
    set kernel argument(kernel, 1, output)
    do in parallel on the device using (global, local):
        call kernel (context, kernel)
  end for
```

While our OpenCL implementation of the finite-differencing simulation does run and produce correct results, we have chosen not to present performance results for this implementation. This is due to the fact that the OpenCL implementation used is still in a pre-release stage and some optimisations do not correctly function. Thus we feel that it would be unfair to compare its performance to the other implementations.

4.8 Cell Processor - PS3

The PS3 implementation of the simulation is designed to make use of the vector processing abilities of the SPEs and the queued memory access. The PPE itself acts as a synchronisation and control unit for the SPEs which perform the actual processing. The PPE uses the mailbox system to communicate with the SPEs and ensure that they remain in sync. As the SPEs and PPE can access the same memory space there is no need for data transfer, simply correct synchronisation. The code to create the threads executing on the SPEs and to communicate with them via mailbox is shown in Algorithm 10.

Algorithm 10 Pseudo-code for an CellBE finite-differencing implementation.

```
declare SPE ← {spe_0, spe_1, ..., spe_5}
for all spe_i ∈ SPE do
  spe_i = spe_create_thread(update, input)
end for
for t = 0 to no_steps do
  for all spe_i ∈ SPE do
    write to inbox(spe_i)
  end for
  for all spe_i ∈ SPE do
    read from outbox(spe_i)
  end for
end for
```

In this simulation the field is defined as an array of type `vector float`. This allows the SPEs to load values and compute the simulation using vector operations (allows four scalar operations to be performed as a single instruction). This effectively processes four cells in the simulation at once.

To improve performance the simulation makes use of enqueued memory access. Initially each SPE will load the vectors required to compute the simulation for the next 16 vectors. Before processing these values, the extra vectors required for the next 16 vectors are enqueued to be fetched. As the memory fetching can be performed asynchronously with the SPE execution, the SPE can compute the simulation for a set of vectors while the values required for the next set are being fetched. This reduces the time the SPE must wait for the memory to be fetched and greatly improves the performance of the simulation.

One challenge with using vector operations is correctly calculating the neighbours. As each element in the vector is a different cell in the simulation, each element's neighbours must be fetched. This is not as simple as fetching the neighbouring vector as this will be the vector neighbour rather than the element neighbour. Instead vectors containing the element neighbours must be generated using suitable insert, extract and rotate methods. See Algorithm 11.

4.9 Finite Difference Results

To test and compare these different implementations, we measured the average computation-time per 1024 simulation time-steps. These results, presented in Figure 10, have been gathered for multiple field lengths to compare the performance across multiple mesh sizes.

Algorithm 11 Pseudo-code for accessing neighbours using vectors on an SPE. This algorithm only fetches the four direct neighbours of a cell.

declare vector float $above \leftarrow input[y-1][x]$
declare vector float $cell \leftarrow input[y][x]$
declare vector float $below \leftarrow input[y+1][x]$
declare vector float $left$
declare vector float $right$
declare float $temp$
$temp \leftarrow$ extract($input[iy][x-1]$, 3)
$left \leftarrow$ rotate_right(vyx)
$left \leftarrow$ insert($temp$, $left$, 0)
$temp \leftarrow$ extract($input[iy][x+1]$, 0)
$right \leftarrow$ rotate_left(vyx)
$right \leftarrow$ insert($temp$, $right$, 3)

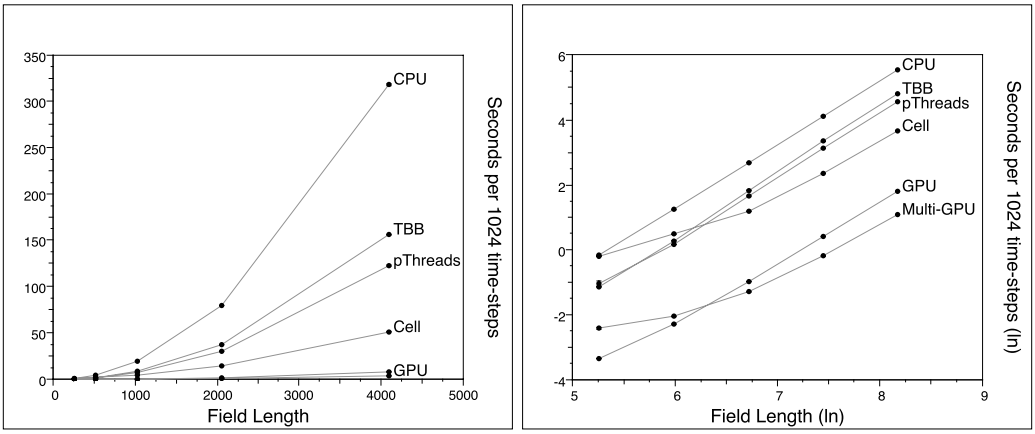

Figure 10: A performance comparison of the finite-differencing results implemented on several devices, shown in normal scale on the left and ln-ln scale on the right. Please note that these graphs do have error bars, however they are all smaller that the points on the graph.

As expected, the single CPU implementation provides the worst performance but provides a meaningful baseline for comparing the other devices. The POSIX threads and Threading Building Blocks implementations provide similar performance, the TBB implementation is the slower of the two.

The simulation running on the CellBE device produced the same results over twice as fast as the PThreads implementation for system sizes of $N > 1024$. System sizes below this were not large enough to make use of the CellBE capabilities.

The GPU implementations provided the best performance running many times faster than the CPU implementation. For a system size of $N >= 2048$ the Multi-GPU version can provide an almost linear (2x) speedup over the single GPU implementation. However, for system sizes any smaller than this, the performance degrades significantly and can take longer to compute the simulation than the single-GPU implementation. This is caused by the need to communicate the values of the border cells through CPU memory. For small system sizes, this communication takes longer than the actual computation and thus the use of multiple GPU devices slows the simulation.

5 Newman's Clustering Coefficient

This section describes how various parallel architectures and toolkits can be used to parallelise Newman's clustering coefficient [71]. This graph metric is based on the concept of clustering in social networks, sometimes also called network transitivity, introduced by Watts and Strogatz [72]. It is commonly used when analysing networks with small-world characteristics [72–74].

5.1 CPU - Sequential Algorithm

Algorithm 12 Pseudo-code for the sequential CPU implementation of the clustering coefficient.

```
function CLUSTERING(G)
  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 (i.e. its degree).
  R ← determine reverse adjacency-list lengths
  declare t //triangle counter
  declare p //paths counter
  for all v_i ∈ V do
    for all v_j ∈ A_i do
      if v_j = v_i then
        p ← p − R[v_i] //correct for self-arcs
        continue with next neighbour v_{j+1}
      end if
      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_k ≠ v_j AND 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
    end for
  end for
  return (3t)/p //the clustering coefficient
```

Algorithm 12 illustrates how the clustering coefficient is calculated. It is defined as:

$$C = \frac{3 \times (\text{number of triangles on the graph})}{\text{number of connected triples of vertices}}$$

Here triangles are elementary circuits of length three (i.e. three distinct vertices connected by three arcs creating a cycle). A connected triple is a path of length two (i.e. two arcs) that connects three distinct vertices.

5.2 Multi-Core: POSIX Threads

The outermost loop of the sequential implementation executes once for every vertex $v_i \in V$. The iterations 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. Algorithms 13 and 14 describe an implementation that uses POSIX Threads (PThreads) to achieve parallelism.

Algorithm 13 Pseudo-code for the multi-core CPU implementation of the clustering coefficient using POSIX Threads. See Algorithm 12 for a description of the input parameter G and the actual triangle and paths counting algorithm.

declare v_{curr} //the current vertex
declare mutex m //mutual exclusion for v_{curr}
function CLUSTERING(G)
  $R \leftarrow$ determine reverse adjacency-list lengths
  $v_{curr} \leftarrow 0$ //initialise current vertex v_{curr}
  $n \leftarrow$ numer of CPU cores
  do in parallel using n threads: **call** PROCESS(G, R)
  wait for all threads to finish processing
  declare $t \leftarrow$ sum of triangles found by threads
  declare $p \leftarrow$ sum of paths found by threads
  return $(3t)/p$ //the clustering coefficient

Algorithm 14 Algorithm 13 continued. The function *PROCESS* is executed in parallel.

function PROCESS(G, R)
  declare t //local triangle counter
  declare p //local paths counter
  declare v_s //start vertex
  declare v_e //end vertex
  repeat
    acquire lock on mutex m
    $v_s \leftarrow v_{curr}$
    $v_e \leftarrow v_s +$ work block size //v_e 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 in the sequential CPU implementation
    end for
  until $v_s \geq |V|$
  return $\{t, p\}$

5.3 Multi-Core: Threading Building Blocks

The TBB implementation, like the PThreads version explained before, applies the parallelism to the outermost loop. TBB's `parallel_reduce` can be used to do this parallel reduction without having to explicitly specify the chunk size and number of threads or having to worry about keeping all the threads busy.

Algorithm 15 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 threshold is reached. Then

TBB uses available worker threads to execute `PROCESS_TASK` (Algorithm 16) in parallel. When the two halves of a range have been processed, then TBB invokes function `JOIN` (Algorithm 17) 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 this root object.

Algorithm 15 Pseudo-code for the multi-core CPU implementation of the clustering coefficient using TBB. See Algorithm 12 for a description of the input parameter G and the actual triangle and paths counting algorithm.

function CLUSTERING(G)
  $R \leftarrow$ determine reverse adjacency-list lengths
  declare blocked_range $br(0, |V|)$
  call parallel_reduce(br, PROCESS_TASK)
  retrieve results and calculate clustering coefficient

Algorithm 16 TBB executes `PROCESS_TASK` in parallel if worker threads are available.

declare t //task local triangle counter
declare p //task local paths counter
function PROCESS_TASK(br, G, R)
  declare $v_s \leftarrow br.begin()$ //start vertex
  declare $v_e \leftarrow br.end()$ //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 17 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.
  $x.t \leftarrow x.t + y.t$
  $x.p \leftarrow x.p + y.p$

5.4 GPU - CUDA

The CUDA implementation of the clustering coefficient is much more complex due to the different hardware architecture and lower level performance tuning necessary to achieve high performance on the GPU.

Arbitrary graphs, like small-world networks, where the structure is not known beforehand, can be represented in different ways in memory. For CUDA applications, the data representation and resulting data accesses often have a major impact on the performance and have to be chosen carefully. Figure 11 illustrates the data structure used to represent a graph in device memory.

Another issue when processing arbitrary graphs with CUDA is that the adjacency-lists differ in length, and often it is necessary to iterate over such a list of neighbours. But since in CUDA's single-instruction, multiple-thread (SIMT) architecture all 32 threads of 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 of the

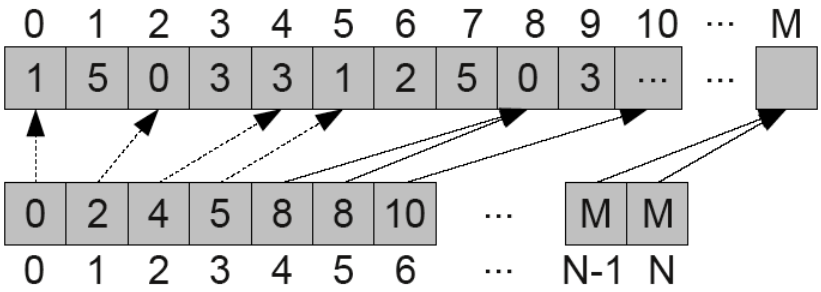

Figure 11: The data structure used to represent the graph in graphics card device memory. It shows the vertex set V (bottom) and the arc set A (top). Every vertex $v_i \in V$ stores the start index of its adjacency-list A_i at index i of the vertex array. The adjacency-list length $|A_i|$ can be calculated 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.

clustering coefficient algorithm, this particular case even requires 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 [75]. The CUDA implementation is described in Algorithm 18.

Algorithm 18 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.

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

As the performance results section shows, this implementation performs well for graphs with only slightly varying vertex degrees, like for example Watts-Strogatz small-world networks [72].

Algorithm 19 Algorithm 18 continued. The device kernel is the piece of code that executes on the GPU.

```
function KERNEL(V, A, S, t, p)
  declare i ← thread ID queried from CUDA runtime
  declare v_i ← S[i] //arc source
  declare 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
```

If the vertex degrees of the input graph vary considerably, as it is typical for scale-free graphs with power-law degree distributions [76–78], a small variation of this implementation performs considerably better. In this second approach, the input array S not only references the source 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's `parallel_sort`, which utilises all available CPU cores.

Further CUDA specific optimisations applied to both versions of the clustering kernel are shown in Algorithm 20. 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 transaction per counter and thread block. Furthermore, texture fetches are used when iterating over the adjacency-lists of vertices, taking advantage of the locality of the data. 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 prefetched and written to shared memory.

5.5 Multi-GPU - CUDA & POSIX Threads

When multiple GPUs are available in the same host system, then it may be desirable to utilise all of them 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 (See Section 5.2). 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

Algorithm 20 Performance optimisations to the CUDA kernel.

```
// shared memory counters for the thread block
__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;
...
const int prefetchCount = 7;
__shared__ int nbr2Prefetch[prefetchCount *
                            THREADS_PER_BLOCK];
if (srcVertex > nbr1) {
  for (int nbr2Idx = 0; nbr2Idx < nArcsNbr1;
       ++nbr2Idx) {
    //prefetch 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);
}
```

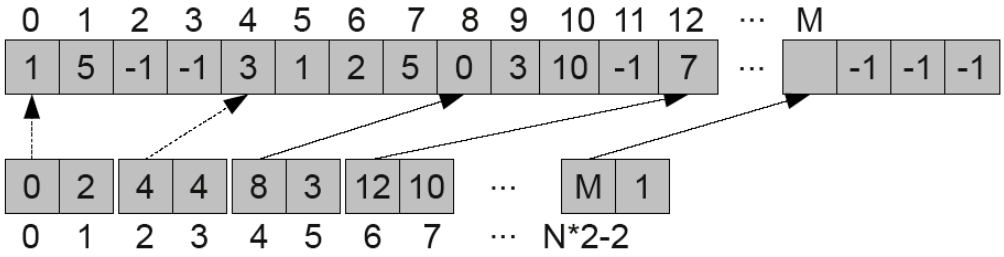

Figure 12: The data structure used to represent the graph in system memory of the Cell BE. It shows the vertex set V (bottom) and the arc set A (top). Every vertex $v_i \in V$ stores the start index of its adjacency-list A_i at index $i \times 2$ of the vertex array. The adjacency-list length $|A_i|$ 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.

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 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)×(blocks per streaming multiprocessor)×(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.

5.6 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 (DMA) used to transfer data from main memory to the local store of a Synergistic Processor Element (SPE) are aligned on 16-byte boundaries makes some changes necessary. See Figure 12 for an illustration and description of the memory layout.

The main task of the Cell's PowerPC Processor Element (PPE) is to manage the Synergistic Processor Elements (SPEs) as illustrated in Algorithm 21. 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 22, 23 and 24). 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 12, 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 25.

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 Synergistic

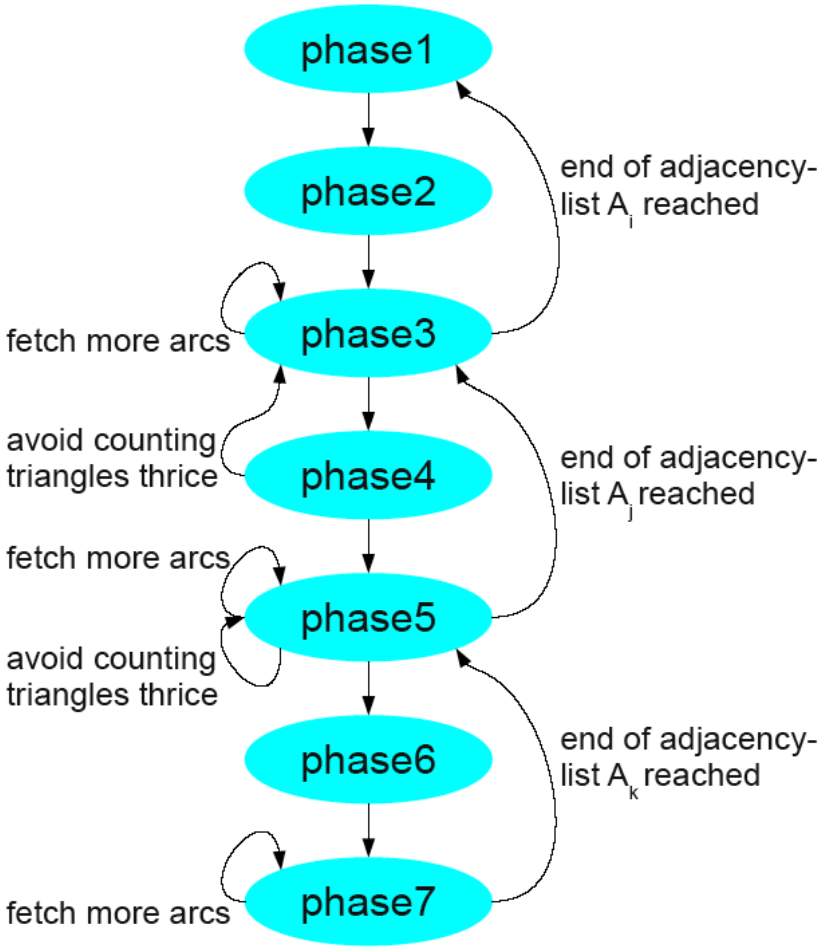

Figure 13: The phases of the SPE implementation and how they are connected to each other. The progression from phase x to phase $x + 1$ is always due to phase x issuing a DMA request to copy data from system memory into local memory which is needed for phase $x + 1$ to execute. Phases with an odd number 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 23 for the pseudo-code of the phases implementation.

Algorithm 21 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 $v_i \in V$ stores the index into the arc set at which its adjacency-list A_i begins in $V[i \times 2]$ and its degree in $V[i \times 2 + 1]$. $|V|$ is the number of vertices in G. $SPE = \{spe_0, spe_1, \ldots, spe_5\}$ is the set of SPEs.
  for all $spe_i \in SPE$ **do**
    initialise spe_i and start processing a block of source vertices
  end for
  while more vertices to process **do**
    for all $spe_i \in SPE$ **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 $spe_i \in SPE$ **do**
    send interrupt signal and wait until spe_i finishes processing
  end for
  aggregate results and calculate clustering coefficient

Processor Unit (SPU) to its Memory Flow Controller (MFC) (every SPE has one SPU and one MFC). However, the performance of the implementation would not be good if the SPU stalled until the requested data becomes available. Instead, the implementation for the SPE is split into phases (See Figure 13 and Algorithms 23 and 24). 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. Whenever a buffer 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 25 describes how the innermost loop of the PPE and SPE implementations makes use of the vector units.

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.

5.7 Clustering Coefficient Results

This section compares the performance of the different clustering coefficient implementations (See Section 3 for a description of the test systems). Two different graphs models are used as input to the algorithms.

The Watts-Strogatz network model [72] generates small-world graphs, where every vertex is

Algorithm 22 The pseudo-code for the SPE implementation of the clustering coefficient on the Cell BE. See Algorithm 21 for the PPE implementation and Algorithm 23 for the different execution phases.

function CLUSTERING(v_s, v_e)
  Input parameters: Each SPE receives an initial work block $[v_s, \ldots, v_e) \subseteq V$ of source vertices to process.
  copy init. data from system mem. to the local store
  initialise buffers $B = \{b_0, b_1, \ldots, b_{15}\}$
  repeat
    $v_{curr} \leftarrow v_s$ //initialise current vertex v_{curr}
    for all $b_i \in B$ **do**
      $b_i.phase \leftarrow phase1$ //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**
      $b \leftarrow$ any "ready" buffer
      call $b.phase$ //execute the next phase of b
    end while
    //check if there is more work to do
    $v_s \leftarrow$ read next value from inbound mailbox
    if no interrupt signal recieved ($v_s \neq -1$) **then**
      $v_e \leftarrow$ read next value from inbound mailbox
    end if
  until interrupt signal received
  copy the results back to system memory

Algorithm 23 Algorithm 22 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 the paths p. A phase ends after an asynchronous DMA request to load data from main memory 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 the 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 for both b.v_i and b.v_j
      b.phase ← phase5
    else if v_k > v_i then
      b.phase ← phase5 //do not count the triangle thrice
    else
      copy_async b.v_k.dat ← load adjacency-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
```

Algorithm 24 Algorithm 23 continued.

function phase7(b)
  for all $b.v_l \in b.A_k$ **do**
    if $b.v_l = b.v_i$ **then**
      $b.t \leftarrow b.t + 1$ //triangle found
    end if
  end for
  $b.phase \leftarrow phase5$

Algorithm 25 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
                              )
                    )
            );
}
```

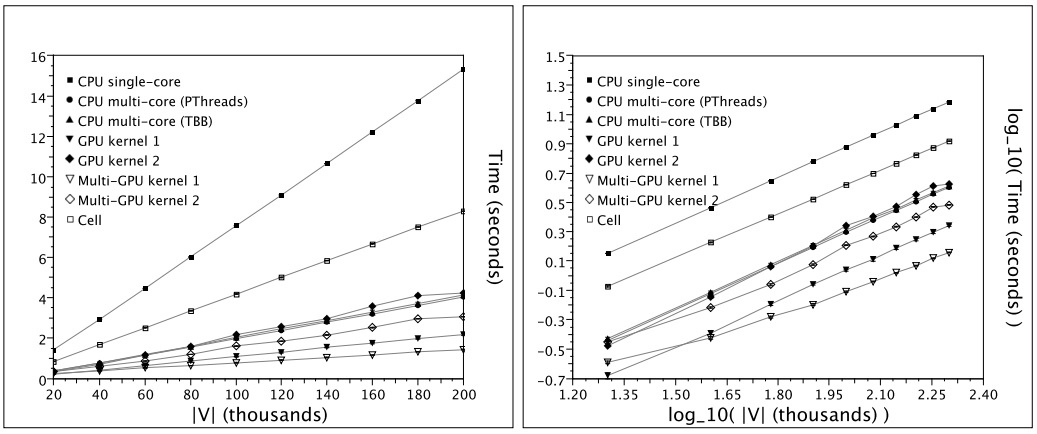

Figure 14: These plots show the performance results for Watts-Strogatz small-world graphs [72] with a rewiring probability $p = 0.1$ and a degree $k = 50$. The number of vertices V ranges from $20,000 - 200,000$. The plot on the right shows the results on a logarithmic scale. The slopes of the least square linear fits rounded to two significant digits are: single-core ≈ 1.03, PThreads ≈ 1.04, TBB ≈ 1.05, GPU kernel 1 ≈ 1.03, GPU kernel 2 ≈ 1.11, multi-GPU kernel 1 ≈ 0.82, multi-GPU kernel 2 ≈ 1.01, and Cell ≈ 0.99. All data points are the mean values of 30 simulation runs. Error bars showing the standard deviations exist, but are smaller then the symbol size.

initially connected to its k nearest neighbours. These edges are then randomly rewired with a probability p. The graphs generated for the performance measurements ($k = 50$ and $p = 0.1$, see Figure 14) have a high clustering coefficient of ≈ 0.53. The vertex degrees do not deviate much from k.

The Barabási-Albert scale-free network model [77], on the other hand, generates graphs with a power-law degree distribution for which the probability of a node having k links follows $P(k) \sim k^{-\gamma}$. Typically, the exponent γ lies between 2 and 3 [76, 78]. The vertex degrees in the resulting graph vary considerably. The graphs generated for the performance measurements ($k \approx 50$, see Figure 15) have a clustering coefficient of ~ 0.01.

The timing results show that the type of graph used as input to the algorithms has a big effect on the execution times. The scale-free graphs take much 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 large adjacency-lists of these few vertices.

The multi-core implementations using PThreads and TBB are $\approx 3.7 - 3.8\times$ faster than the single-core implementation for the small-world graphs, and $\approx 2.4 - 3.9\times$ faster for the scale-free graphs. Thus, they scale well with the 4 CPU cores, especially in the former case, but lose some ground with increasing network size in the latter case. The TBB implementation is in all tests slightly slower than the PThreads implementation. But unless the last percentage of performance has to be squeezed out of the algorithm, the ease of development and scalability to different system configurations easily makes up for this small performance penalty.

As mentioned in Section 5.4, we have two CUDA kernels that differ in one aspect. 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 $|A|\times$ (size of integer) more space and introduces some processing overhead, which shows in the lower performance when processing the small-world graphs. Even the multi-GPU implementation of kernel 2 is slower then the single-GPU implementation of kernel 1. However, the reduced warp divergence gained through this overhead pays off when

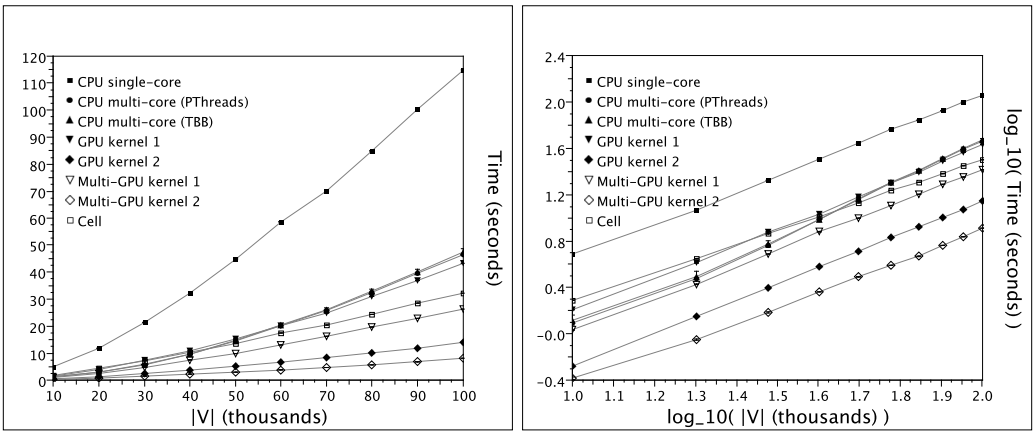

Figure 15: These plots show the performance results for Barabási's scale-free graphs [77] with a degree $k \approx 50$. The number of vertices V ranges from $10,000 - 100,000$. The plot on the right shows the results on a logarithmic scale. The slopes of the least square linear fits rounded to two significant digits are: single-core ≈ 1.37, PThreads ≈ 1.56, TBB ≈ 1.67, GPU kernel 1 ≈ 1.45, GPU kernel 2 ≈ 1.43, multi-GPU kernel 1 ≈ 1.40, multi-GPU kernel 2 ≈ 1.40, and Cell ≈ 1.23. All data points are the mean values of 30 simulation runs. Error bars showing the standard deviations exist, but are smaller then the symbol size.

processing scale-free graphs. Here the scenario is reversed and kernel 2 clearly outperforms kernel 1 by an even larger margin. This shows once again [8, 10] that the performance of graph algorithms running on the GPU in many cases depends on the graph structure and that there is not one best implementation for all cases. If the graph structure is not known beforehand, then it may be worthwhile to attempt to automatically determine the type of graph in order to be able to choose the optimal CUDA implementation.

The multi-GPU implementations using the 2 GPUs of the GeForce GTX295 perform best when the graph size is large enough to keep all processing units of both devices busy. The multi-GPU version of kernel 1 compared to the single-GPU version is $\approx 1.3\times$ slower when processing the smallest instance of the Watts-Strogatz graphs with $|V| = 20,000$, but already surpasses it when comparing the next larger network size with $|V| = 40,000$ with a performance gain of $\approx 1.1\times$. When processing the largest graph with $|V| = 200,000$ vertices, then it is $\approx 1.5\times$ faster. The multi-GPU version of kernel 2 is always faster than its single-GPU sibling when processing the scale-free graph instances, which is not further surprising as these graphs take much longer to process and the overhead of using two GPUs is therefore less significant. The performance gains range from $\approx 1.3\times$ for the smallest instance to $\approx 1.7\times$ for the largest instance.

Last but not least, the Cell implementation positions itself between the single- and multi-threaded CPU implementations when processing the small-world graphs or the smaller instances of the scale-free graphs. Due to its better scaling, it surpasses the performance of the multi-core algorithms when the size of the scale-free graphs reaches $|V| \sim 50,000$.

6 Random Number Generators

Random number generation is one of the most widely used facilities in computer simulations. A number of different algorithms are widely used [79, 80], ranging from fast but low quality system supplied generators such as the rand()/random() generators available on Unix [81] systems to slower but high quality 64-bit algorithms such as the Mersenne Twistor generator [82]. Marsaglia's

lagged-Fibonacci generator [83] is a 24-bit algorithm that produces good quality uniform deviates and which has been widely used in Monte Carlo work [84]. It is convenient for our purposes in this present paper as not all our target accelerator hardware platforms uniformly support 64-bit floating point calculations.

6.1 CPU - Sequential Algorithm

The Marsaglia lagged-Fibonacci random number generator (RNG) has been described in full elsewhere [83], but in summary the details are given in Algorithm 26, which we provide for completeness.

Algorithm 26 Marsaglia Uniform Random Number Generator, where an initialisation procedure sets the values as given below, and fills the lag table with deviates.

```
declare u[97]
initialise(u, seed)
declare i ← 96
declare j ← 32
declare c ← 362436.0/16777216.0
declare d ← 7654321.0/16777216.0
declare m ← 16777213.0/16777216.0
for n ← 0 to N do
  uniform(i, j, c, d, m, u)
end for
```

Algorithm 27 Marsaglia Uniform Random Number Generator, each call will generate a single random number.

```
function uniform(i, j, c, d, m, u)
  declare result ← u[i] − u[j]
  if result < 0 then
    result ← result + 1
  end if
  u[i] ← result
  i ← i − 1
  if i < 0 then
    i ← 96
  end if
  j ← j − 1
  if j < 0 then
    j ← 96
  end if
  c ← c − d
  if c < 0 then
    c ← c + m
  end if
  result ← result − c
  if result < 0 then
    result ← result + 1
  end if
  return result
end function
```

Where i, j index a lag table which is shown here of 97 deviates, but which can be any suitable prime, subject to available memory and where c, d, m are suitable values.

A number of optimisations for this sort of random number generation algorithm are possible on the various implementation platforms. One obvious one is to synchronise a separate thread that can produce an independent stream of random deviates that are consumed by the main application thread. Other algorithms, whose descriptions are beyond the space limitations of our present paper, generate whole vectors or arrays of deviates together using a SIMD approach which can be used in applications that have similarly shaped work arrays or objects such as images or model data fields.

6.2 Multi-Core: POSIX Threads

The PThreads implementation of the lagged-Fibonacci generator launches separate threads that each generate separate streams of random numbers. To do this each thread creates and initialises its own lag-table with a unique seed. The threads can then simply generate random numbers using this unique stream and the same `uniform` function as described in Algorithm 28.

Algorithm 28 Marsaglia Uniform Random Number Generator, where an initialisation procedure sets the values as given below, and fills the lag table with deviates.

function generate(id)
  declare $u[97]$
  declare $i \leftarrow 96$
  declare $j \leftarrow 32$
  declare $c \leftarrow 362436.0/16777216.0$
  declare $d \leftarrow 7654321.0/16777216.0$
  declare $m \leftarrow 16777213.0/16777216.0$
  initialise(u, id)
  for $n \leftarrow 0$ to N **do**
    uniform(i, j, c, d, m, u)
  end for
  signal complete($t_i d$)

Each thread that is created will generate `N` random numbers and then signal the main thread that it has completed its work. This code merely generates random numbers and does not make any use of them but it is assumed that any PThreads application that uses random numbers would make use of them within this thread loop.

6.3 Multi-Core: Threading Building Blocks

Like the PThreads implementation, the TBB implementation of the lagged-Fibonacci generator creates a number of independent RNG instances to generate streams of random numbers. However, the RNG instances are not associated with a particular hardware thread. Instead, they are each contained in a structure that can also store additional, application specific information related to the RNG instance. For example, it may also 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 29 for a description of this initialisation process.

The parallel random number generation using these RNGs is invoked by passing the begin and end iterators of the vector to TBB's `parallel_for_each` function, together 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`.

Algorithm 29 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)
  declare V //vector
  declare $r_0 \leftarrow$ new RngStruct
  initialise(r_0, s_0)
  for $i \leftarrow 1$ to t **do**
    declare $r_i \leftarrow$ new RngStruct
    declare $s_i \leftarrow$ uniform(r_0) $*$ INT_MAX //seed for r_i
    initialise(r_i, s_i)
    append r_i at the end of vector V
  end for
  return V

The function called by `parallel_for_each` can then use the RNG instance passed to it to fill the array or 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 function with a particular RNG structure instance as parameter can be different 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. While a larger number of RNG instances allows the code to scale to more processor cores, it also increases the overhead introduced by switching tasks. If there are no other processes or threads consuming a significant amount of processing resources, then setting the number of RNG instances equal to the number of hardware threads gives the highest and most consistent performance in our tests. If, however, other threads are using some processing power, too, then splitting the problem into a larger number of smaller tasks gives the task scheduler more flexibility to best utilise the remaining resources.

6.4 GPU - CUDA

The 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 is repeatable and fast as race conditions are avoided and no communication between threads is required. Algorithms 30 and 31 illustrate the implementation of Marsaglia's algorithm in CUDA. 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 for the table per thread. The input seed value is used to initialise a random number generator (RNG) 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 random number generator 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.

The CUDA implementation is 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 transaction per half-warp.

Algorithm 30 CUDA implementation of Marsaglia's RNG that produces T independent streams of random numbers, where T is the number of threads. See Algorithm 31 for the CUDA kernel.

declare $T = 30720$ //thread count
declare $L = 97$ //lag table length
function RNG1(s)
  Input parameters: s is the initialisation seed.
  declare $S[T]$ //array of seeds
  initialise host RNG with s
  $S \leftarrow$ generate T random deviates on the host
  declare $S_d[T]$ in device memory
  copy $S_d \leftarrow S$
  declare $U_d[T \times L]$ in device mem. //array of lag tables
  declare $C_d[T]$ in device mem. //array of c values
  declare $I_d[T], J_d[T]$ in device mem. //arrays of indices
  do in parallel on the device using T threads:
    call KERNEL(S_d, U_d, C_d, I_d, J_d)

Algorithm 31 Algorithm 30 continued. The device kernel is the piece of code that executes on the GPU. The initialisation and uniform random number generator functions are essentially the same as on the CPU.

function KERNEL(S, U, C, I, J)
  declare $i \leftarrow$ thread ID queried from CUDA runtime
  $C[i] \leftarrow 362436.0/16777216.0$
  $I[i] \leftarrow L - 1$
  $J[i] \leftarrow L/3$
  declare $s \leftarrow S[i]$ //load the thread's seed
  initialise the thread's RNG using s
  generate random deviates when needed

6.5 Multi-GPU - CUDA & POSIX Threads

The multi-GPU version of our approach to implementing Marsaglia's RNG in CUDA is basically the same as its single-GPU counterpart. One PThread is created for every CUDA capable device in the system. These threads are used to control the CUDA kernel preparation and execution on the device associated to them. Instead of having to compute T random deviates as seeds for the thread RNGs, the host now has to generate $T \times N$ seeds, where T is the number of threads per device and N is the number of devices.

6.6 Cell Processor - PS3

Implementing the lagged-Fibonacci generator on the Cell processor requires a certain deal of consideration. There are six separate SPEs each of which can process a vector for four elements synchronously. Vectors types are used to make full use of the SPEs processing capabilities. Thus for each iteration, each SPE will generate four random numbers (one for each element in the vector).

To ensure that unique random numbers are generated, each element in the vector of each SPE must have a unique lag table. Six SPEs with four elements per vector results in twenty-four lag tables. These lag tables are implemented as a single lag table of type `vector float` but each element of the vectors is initialised differently. Care should be taken when initialising these lag tables to make certain that the lag tables do not have correlated values and produce skewed results.

The lagged-Fibonacci generator algorithm has a two conditional statements that affect variables of vector type. These conditional statements both take the form of `if( result < 0.0) result = result + 1.0;` (See Algorithm 26). As each element in the vector will have a different value depending on its unique lag table, different elements in the vector may need to take different branches.

There are two ways of overcoming this issue. The first method is to extract the elements from the vector and process them individually. This method is not ideal as it does not use the vector processing ability of the cell, instead the `spu_sel` and `spu_cmpgt` instructions can be used.

The `spu_cmpgt` instruction will compare two vectors (greater than condition) and return another vector with the bits set to 1 if the condition is true and 0 if the condition is false. The comparison is performed in an element-wise manner so the bits can be different for each element. The `spu_sel` can then select values from two different values depending on the bits in a mask vector (obtained from the `spu_cmpgt` instruction).

Using these two instructions the conditional statement `if( result < 0.0) result = result + 1.0;` can be processed as vectors with different branches for each element. The pseudo-code for this process can be seen in Algorithm 32.

6.7 RNG Implementation Results

The implementations of the lagged-Fibonacci generators have been tested by generating 24 billion random numbers and measuring the time taken. In the performance measures (See Table 1) the random numbers have not been used for any purpose as the only intention was to measure the generation time. 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. Otherwise the random values can simply be written to memory and extracted from the device for use elsewhere.

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 and vector units execute independently from each another, using different lag tables and generating multiple streams of random numbers. The initial set-up time is insignificant compared to the time taken to generate 24 billion random numbers.

Algorithm 32 Pseudo-code for Marsaglia Lagged-Fibonacci algorithm implemented on the CellBE using vectors.

```
declare vector float u[97]
initialise(u)
declare i ← 96
declare j ← 32
declare c ← 362436.0/16777215.0
declare d ← 7654321.0/16777215.0
declare m ← 16777213.0/16777215.0
function uniform()
  declare vector float zero ← spu_splats(0.0)
  declare vector float one ← spu_splats(1.0)
  declare vector float result ← u[i] - u[j]
  declare vector float plus1 ← result + one
  declare vector unsigned sel_mask ← result > zero
  result ← select(result, plus1, sel_mask)
  u[i] ← result
  i = i − 1
  if i == 0 then
    i ← 96
  end if
  j = j − 1
  if j == 0 then
    j ← 96
  end if
  c = c − d
  if c < 0 then
    c ← c + m
  end if
  result ← result−spu_splats(c)
  plus1 ← result + one
  sel_mask ← result > zero
  result ← select(result, plus1, sel_mask)
  return result
end function
```

Table 1: Comparison between implementations of the time taken to generate 24,000,000,000 random numbers using the lagged-Fibonacci generator.

Device	Time (seconds)	Speed-up
CPU	256.45	1.0x
PThreads	66.72	3.8x
TBB	95.40	2.7x
Cell	23.60	10.9x
CUDA	8.40	30.5x
Multi-GPU	4.33	59.2x

7 Discussion

Amdahl's law [85] manifests itself in an interesting manner for multi-core processors [86]. The weakest links in the parallelism chain can be directly attacked by specialist accelerator processing elements even when no one single parallel paradigm suits the whole application code. Scientific simulations can often benefit from Gustafson's corollary [87] or "work-around" to scalability limitations in that the problem size or amount of intrinsic parallelism available in a simulation problem can be adjusted to fit the parallel architecture available. The accelerator effect is in some ways better than the Gustafson "scaling work-around" to the Amdahl limitation in a monolithic parallel architectural design, since it gives the designer the flexibility to add accelerators at various points in an architecture to focus resources where needed. However, the difficulty is handling the resulting architectural diversity of approaches – and consequent systems complexity – within a single application program.

It has traditionally been difficult to make high level comparisons from an applications perspective between accelerator devices since so much detail is needed to be fair to what are usually completely different architecturally modelled systems. We have tried to explain architectures in a common manner and also to give different algorithm and code fragments in a way that allows cross comparison while still referring to key device-specific libraries, special instructions and constructs. There are several common programming model themes that these devices have which may help to promote the use of OpenCL and its likely successors. The challenge at the time of writing is embracing the diversity of devices to choose and exploit the best features – perhaps making use of a hybrid design.

In this section we discuss our experiences and results on the accelerator models (Section 7.1) and speculate on the need for a unifying architectural model (Section 7.2) and the need for a common terminology amongst accelerator devices (Section 7.3.)

7.1 Accelerator Models

The tradeoff of programmer effort against run time performance is a very real and obvious one for the accelerator devices and applications we have described. Of the devices we have discussed, adapting applications to **multi-core CPU** devices presents the least programmer workload. The code and libraries are the same as a single-core CPU and the implicit caching methods of modern CPU devices makes for simple program designs. The main challenge of multi-threaded programming is the synchronisation of threads to ensure no race or gridlock conditions occur. The well-established **PThreads** library provides an adequate toolset for this, but provides no particular guidance on which of the many methods of parallel decomposition to use or provide any protection against the many potential concurrency problems that can ensue.

The development of Intel's **Threading Building Blocks**(TBB) [44] has eased this burden considerably by abstracting many common parallel decomposition methods into library functions. This allows non-expert developers to easily produce code for multi-core CPUs. Although there is a small performance hit against use of PThreads, we believe the added convenience and improved development time and programmer safety accruing from TBB is well worth it. TBB is not of course the only attempt at higher level thread management with systems like Cilk [88] and languages like Java with its in-built applications thread model [89] both available. TBB does however seem a good compromise between performance and usability.

The main limiting factor of **homogeneous multi-core** simulations compared with other accelerator devices we considered arises from the restrictions of processing power. Current generation CPUs are limited to 4 cores, however with the upcoming releases of 6-, 8- and more core mainstream CPUs, this condition may prove to be less of a restriction in future [90].

GPU development has much more of a learning curve than developing multi-core CPU applications. The C-style language of CUDA definitely eases the process of migrating code to GPU architectures, however, a high-level of knowledge about the GPU architecture and memory access

methods is required to produce the results GPUs are capable of. This being said, there is a large user-group that has already accepted CUDA [75] which can provide support for beginning GPU developers.

The major attraction of GPUs as accelerators is the very high-performance they can produce if programmed correctly. All of our experiments showed that GPUs and multi-GPU devices provided the best performance of any device. They have also been shown to provide speed-ups for problems of a serial nature [10] and in favourable conditions can provide speed-up factors of over 150x.

The **Cell Broadband Engine** has shown to provide a significant speed-up over the simple single-core CPU in our experiments. This is particularly impressive also when taking device cost and heat production into account. However, the complexity of code development made this a somewhat less favourable option, especially considering the results produced are not significantly faster than the multi-core CPU applications. Developing code to correctly use the vector processing abilities of the Cell and to reduce the waiting time for memory requests to be completed by the MFCs proved to be a challenge. However, we would expect that ongoing compiler developments do more of this instruction-level and vector-level work in the future.

7.2 Model Unification

Not all of these architectures are mutually exclusive. The GPU still needs to be managed by a host system, which at the time of writing and for the foreseeable future is most likely a multi-core CPU. Memory copies to and from the device as well as CUDA kernel executions can be performed asynchronously, allowing the CPU to perform other tasks while waiting for the results from the graphics device. Thus, a combination of multi-core CPUs and GPUs has an even higher performance potential than either one by itself. This trend is reflected by projects like MAGMA [91], which aims to develop a dense linear algebra library for hybrid architectures, starting with multi-core CPU + GPU systems.

The OpenCL standard is designed to take advantage of heterogenous processing resources such as these. It can be used to describe both data-parallel and task-parallel computing tasks. OpenCL attempts to provide a unified environment for concurrent programming (See Figure 16), but it remains to be seen how well actual implementations will perform.

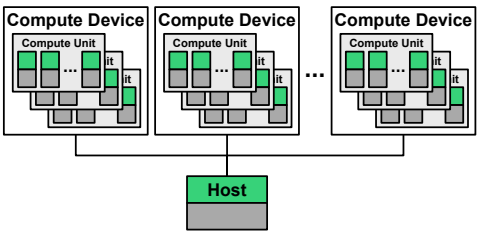

Figure 16: A host processor delegating computational load to OpenCL capable compute devices.

While OpenCL and other systems do promise some unification of ideas for architecture and organising parallel processing, it is still unclear how a programmer unifies all the different memory compromises and sub types into a software design. All the devices we have discussed make use of many different sorts of memory within cores and processors to avoid starving the processing elements of data to work on. These are generally over and above the off-chip caches used to improve main memory management. It is interesting to reflect that while main memory costs have continued to drop dramatically to the point where it is economic to have multiple gigabytes of main memory in a desktop, the speed has not particularly improved. Slow main memory is only feasible in present generation machines because of sophisticated memory caching. It may be that a shift in the balance point between processor speed and memory access speed is needed for much

further progress to be made in usefully accessible aggregate processing speed from applications.

As we have discussed the idea of accelerators is not new but the devices we have explored make use of the accelerator idea in subtle new ways. Devices and their cores now have a greater or lesser power to interact with the operating system services such as hard disk. The CellBE SPE cores have more delegated powers than the cores on a GPU for example. By contrast a conventional core on a homogenous core CPU has complete autonomy with respect to its peers. Autonomy at this level may be difficult to achieve, but is definitely desirable from the perspective of latency hiding in many algorithms.

7.3 Need for a Common Terminology

It will likely be important for future architectural models and their associated communities to agree on a terminology for all the different sorts of in- and off-core memories. Some progress has been possible with different cache models due to a consistent naming scheme and for example, L1, L2 and L3 cache are now relatively well understood concepts, independent of particular chips and vendors [92]. The problem of naming and managing memory models for different devices is still a difficult challenge. Often it is necessary to convert or re-tile application data in different optimal ways to exploit parallelism on different devices [93]. Libraries that do this will likely be feasible, but only if there is common agreement as to what level and shape of device memory is being referred to.

At a general level, the sorts and granularity of concurrency that is available and of interest to the applications programmer are different to those of interest and that can be exploited by systems level programmers. It is generally not a burden for applications programmers to think in terms of low granularity SIMD with large numbers of processors or cores available. Coarse-grained threading or process management is of more interest to the operating systems level programmer but is generally seen as non-portable clutter in an applications program. The use by chip vendors of the term SIMD to refer to vector operations that are only four 32-bit integers long for example is in stark contrast to the term's use in the applications programming world where it usually implies a much higher degree of massive parallelism.

A coming together of stakeholder interests and closer common use of such terms is likely over the next few years as both sides battle further with portable and more easily deployable parallelism. Standards like OpenCL will undoubtedly help and we might hope for similar consortium approaches to a better threading system, that absorbs some of the clearly helpful TBB ideas into a future PThread portable standard.

Portable parallel language and tool development will continue to be a key area of interest for applications developers. While we remain optimistic about high level parallelism in languages like Erlang [41,94] and programmer additions to Fortran and C like languages, it is likely that the need for applications developers to know and use lower level features for at least part of their scientific simulation codes will only increase.

It may be that the grail of producing the perfect applications-level parallel programming language is not attainable in the short term, but a compromise where a C-like language with OpenCL and TBB style features is available as an intermediate language may be a useful and achievable approach. We are exploring this idea through automated generators of applications simulation software that use CUDA and OpenCL level languages as intermediate representations [95].

At the time of writing OpenCL implementations are still in their very early stages of release. The notion of a device as in OpenCL is a powerful one for describing a generalised accelerator architecture. It has been useful that NVIDIA's CUDA language embodies much of the OpenCL ideas already. This has enabled us and others to consider some OpenCL ideas already even though we have not yet been able to quote specific OpenCL implementation performance data. It remains to see whether OpenCL captures all the necessary detail and flexibility to accommodate future devices.

8 Summary and Conclusions

We have discussed three important algorithms widely used in scientific simulation applications – finite differencing solutions to multi-dimensional partial differential field equations; graph network analysis metrics such as Newman's clustering coefficient; and floating point intensive random number generators. We have used these three as vehicles for exploring and comparing some of the hardware accelerator platforms becoming very widely available – multi-core CPUs; auxiliary devices such as GPUs with a very large numbers of specialist cores; and asymmetric-core processors such as the CellBE where there are accelerator cores built-in.

We have reported on work that looks at the device details and considers how multiple devices can be used for HPC. This applies to desktop and medium-sized cluster systems as well as supercomputer installations where the extra effort required to bring power dissipation requirements down may be a necessity in the immediate future. The continued need to embrace multi-core accelerators and the importance of attaining good efficiency to keep down heat dissipation budgets will make hybrid solutions attractive economically and technically. We have found that although these devices are all quite different, a descriptive framework such as OpenCL already provides a standard vocabulary and is beginning to provide a common model for programming them.

As reported in Sections 4.9, 5.7 and 6.7 we found quite significant variations in performance for our applications implemented on the various accelerator devices. Not all the devices provide a scalable solution for the applications algorithms we have discussed. Generally CUDA provides a feasibly scalable solution for the finite differencing and clustering coefficient problems. However the only way to use it effectively for random number generation would eventually be prohibitive in terms of each core having its separate copy of a random number lag or state table. It is hard to conclude that any one device approach is best and it is tempting to predict that a flexible multi-accelerator hybrid approach will be the best for achieving good performance on any but the simplest of applications.

We observe that in all cases, it is not sufficient for applications programmers to rely upon optimising compilers to attain anything like the peak performance of these devices. The programming situation is very similar to that of massively parallel supercomputers in the 1980s – parallel expertise and programming effort is necessary even at the applications programmer level.

These devices have come about in part due to physical constraints and power considerations at the silicon implementation level. In spite of concerted efforts to find new physical technologies, these trends are very likely to continue and programmers will need to expend further effort to exploit these devices. The number of supercomputer users was (and still is) very much smaller than the number of users of these chip level devices and thus the economic forces for driving the market for new support tools and software will be much greater this time around.

We have found potential power in all the hardware devices we have discussed in this paper. There continues to be considerable diversity in the architectural models and software technologies available to program them. While programming raw parallel devices will continue to be a challenge, there will be considerable pressure to find productivity tools to support systems and application programmers.

We believe that the technologies we have explored and discussed all have a useful role to play in scientific simulations programming. It is by no means clear that any one current approach is better than any other and it is likely that hybrid architectural solutions will become more feasible under software models such as that of OpenCL. It appears that OpenCL is a definite step in the right direction. Although it is still a relatively low-level programming language, it does provide a potentially powerful and helpful unifying architectural model against which application designs and even economically worthwhile higher-level software tools and languages can target.

While it is unfortunate that it has taken the present crisis in Moore's Law for the importance of parallel computing to be re-embraced, at least the economic imperatives do now seem to be raising awareness of the importance of parallel computer science at all levels. At least for the foreseeable future it will likely be worthwhile and indeed necessary for applications programmers to learn to

explicitly program parallelism at both inter-device and intra-device levels.

We conclude that the absolute values of individual node performance **are** still important in multi-processor systems and that the use of accelerator devices – and most probably **multiple** and **hybrid** devices – will continue to be an important approach. In view of the difficulties for the applications programmer in achieving good efficiency on multi-core devices, a key tradeoff in the future will be application programmer usability against attainable performance. Various unification architecture models and associated software tools and libraries can of course alleviate this critical problem. While accelerators remain hard to fully exploit, they do provide an edge for smaller research groups who can trade intensive human programmer effort off against brute-force "big iron" HPC solutions. As automated and software tools make it easier to exploit accelerator devices, they will also be highly useful in accelerating supercomputer applications that make use of clustered nodes [96].

References

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine*, vol. April, p. 4, 1965.

[2] S. K. Moore, "Multicore is bad news for supercomputers," *IEEE Spectrum*, vol. 45, no. 11, p. 11, 2008.

[3] G. Goth, "Entering a parallel universe," *Communications of the ACM*, vol. 52, no. 9, pp. 15–17, September 2009.

[4] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," *Communications of the ACM*, vol. 52, no. 9, pp. 89–97, September 2009.

[5] A. Ghuloum, "Face the inevitable, embrace parallelism," *Communications of the ACM*, vol. 52, no. 9, pp. 36–38, September 2009.

[6] A. Trefethen, i.S. Duff, N. Higham, and P. Coveney, "Developing a high performance computing / numerical analysis roadmap," University of Oxford, E-Science Centre, Tech. Rep., January 2009, version 1.0. [Online]. Available: www.oerc.ox.ac.uk/research/hpc-na

[7] D. P. Playne, A. P. Gerdelan, A. Leist, C. J. Scogings, and K. A. Hawick, "Simulation modelling and visualisation: Toolkits for building artificial worlds," *Research Letters in the Information and Mathematical Sciences*, vol. 12, pp. 25–50, 2008.

[8] A. Leist, D. Playne, and K. Hawick, "Exploiting Graphical Processing Units for Data-Parallel Scientific Applications," *Concurrency and Computation: Practice and Experience*, vol. 21, pp. 2400–2437, December 2009, cSTN-065.

[9] D. Playne and K. Hawick, "Data Parallel Three-Dimensional Cahn-Hilliard Field Equation Simulation on GPUs with CUDA," in *Proc. 2009 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'09) Las Vegas, USA.*, no. CSTN-073, 13-16 July 2009.

[10] K. A. Hawick, A. Leist, and D. P. Playne, "Parallel Graph Component Labelling with GPUs and CUDA," Massey University, Tech. Rep. CSTN-089, June 2009, Accepted (July 2010) and to appear in the Journal Parallel Computing.

[11] D. P. Playne, M. G. B. Johnson, and K. A. Hawick, "Benchmarking GPU Devices with N-Body Simulations," in *Proc. 2009 International Conference on Computer Design (CDES 09) July, Las Vegas, USA.*, no. CSTN-077, July 2009.

[12] M. J. Stock and A. Gharakhani, "Toward efficient GPU-accelerated N-body simulations," in *in 46th AIAA Aerospace Sciences Meeting and Exhibit, AIAA 2008-608*, January 2008.

[13] G. D. Kavanagh, M. C. Lewis, and B. L. Massingill, "GPGPU planetary simulations with CUDA," in *Proceedings of the 2008 International Conference on Scientific Computing*, 2008.

[14] K. Hawick, A. Leist, and D. Playne, "Damaged Lattice and Small-World Spin Model Simulations using Monte-Carlo Cluster Algorithms, CUDA and GPUs," Computer Science, Massey University, Tech. Rep. CSTN-093, 2009.

[15] 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 *Proc. 37th IEEE Int. Conf on Parallel Processing*, 2008, pp. 520–527.

[16] C.J.Scogings and K. A. Hawick, "Simulating intelligent emergent behaviour amongst termites to repair breaches in nest walls," Computer Science, Massey University, Tech. Rep. CSTN-097, 2010.

[17] A. Lesit and K. Hawick, "Small-world networks, distributed hash tables and the e-resource discovery problem in support of global e-science infrastructure," Massey University, Tech. Rep. CSTN-069, January 2009.

[18] K. Fatahalian and M. Houston, "A Closer Look at GPUs," *Communications of the ACM*, vol. 51, no. 10, pp. 50–57, October 2008.

[19] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. Purcell, "A survey of general-purpose computation on graphics hardware," in *Eurographics 2005, State of the Art Reports*, September 2005, pp. 21–51.

[20] W. Langdon and W. Banzhaf, "A SIMD Interpreter for Genetic Programming on GPU Graphics Cards," in *Proc. EuroGP*, M. O'Neill, L. Vanneschi, S. Gustafson, A. E. Alcazar, I. D. Falco, A. D. Cioppa, and E. Tarantino, Eds., vol. LNCS 4971, March 2008, pp. 73–85.

[21] P. Messmer, P. J. Mullowney, and B. E. Granger, "GPULib: GPU computing in high-level languages," *Computing in Science & Engineering*, vol. 10, no. 5, pp. 70–73, September/October 2008.

[22] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *ACM Queue*, vol. 6, no. 2, pp. 40–53, March/April 2008.

[23] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: stream computing on graphics hardware," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 777–786, 2004.

[24] *ATI CTM Guide*, AMD, 2006. [Online]. Available: \url{http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf}

[25] M. McCool and S. D. Toit, *Metaprogramming GPUs with Sh.* A K Peters, Ltd., 2004.

[26] Khronos Group, "OpenCL - Open Compute Language," 2008. [Online]. Available: http://www.khronos.org/opencl/

[27] N. Wilding, A. Trew, K. Hawick, and G. Pawley, "Scientific modeling with massively parallel SIMD computers," *Proceedings of the IEEE*, vol. 79, no. 4, pp. 574–585, Apr 1991.

[28] W. D. Hillis, *The Connection Machine.* MIT Press, 1985.

[29] S. L. Johnsson, "Data parallel supercomputing," Yale University, New Haven, CT 06520, Preprint YALEU/DCS/TR-741, sep 1989, the use of Parallel Procesors in Meteorology, Pub. Springer-Verlag.

[30] H. W. Yau, G. C. Fox, and K. A. Hawick, "Evaluation of High Performance Fortran through applications kernels," in *Proc. High Performance Computing and Networking 1997*, Vienna, Austria, April 1997.

[31] K. A. Hawick and P. Havlak, "High performance Fortran motivating applications and user feedback," Northeast Parallel Architectures Center, Tech. Rep. NPAC Technical Report SCCS-692, January 1995.

[32] G. Fox, "Fortran D as a portable software system for parallel computers," jun 1991, center for Research on Parallel Computation, Rice University, PO Box 1892, Houston, Texas, TX 77251-1892, CRPC-TR91128.

[33] B. Chapman, P. Mehrotra, and H. Zima, "Vienna Fortran - a Fortran language extension for distributed memory multiprocessors," Institute for Computer Applications in Science and Engineering, NASA, Langley Research Center, Hampton, Virginia 23665-5225, ICASE Interim Report 91-72, sep 1991.

[34] Z. Bozkus, A. Choudhary, G. C. Fox, T. Haupt, and S. Ranka, "Fortran 90D/HPF compiler for distributed-memory MIMD computers: design, implementation, and performance results," in *Proc. Supercomputing '93*, Portland, OR, 1993, p. 351.

[35] E. A. Bogucz, G. C. Fox, T. Haupt, K. A. Hawick, and S. Ranka, "Preliminary evaluation of high-performance Fortran as a language for computational fluid dynamics," in *Proc. AIAA Fluid Dynamics Conf*, no. AIAA 94-2262, Colorado Springs, June 1994.

[36] P. S. Pacheco, *Parallel Programming with MPI*. Morgan Kaufmann Publishers Inc., 1996.

[37] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers Inc., 2001.

[38] P. Henning and A. B. W. Jr., "Trailblazing with roadrunner," *Computing in Science and Engineering*, vol. 11, no. 4, pp. 91–95, July/August 2009.

[39] Oak Ridge National Laboratory, "Exploring science frontiers at petascale," National Center for Computational Sciences, USA, Tech. Rep., 2008. [Online]. Available: www.nccs.gov/jaguar

[40] B. Cantrill and J. Bonwick, "Real-world concurrency," *Communications of the ACM*, vol. 51, no. 11, pp. 34–39, November 2008.

[41] J. Armstrong, M. Williams, C. Wikstrom, and R. Virding, *Concurrent Programming in Erlang*. Prentice Hall, 1996, iSBN 0-13-285792-8.

[42] B. Meyer, "Systematic concurrent object-oriented programming," *Communications of the ACM*, vol. 36, no. 9, pp. 56–80, September 1993, sCOOP.

[43] IEEE, *IEEE Std. 1003.1c-1995 thread extensions*, 1995.

[44] Intel, *Intel(R) Threading Building Blocks Reference Manual*, 1st ed., Intel, July 2009.

[45] Sun Microsystems, "Java Programming Language," http://java.sun.com, last accessed September 2009.

[46] *CUDA^{TM} 2.0 Programming Guide*, NVIDIA® Corporation, 2008, last accessed November 2008. [Online]. Available: http://www.nvidia.com/

[47] *Technical Overview ATI Stream Computing*, ATI, 2009. [Online]. Available: http://developer.amd.com/gpu_assets/Stream_Computing_Overview.pdf

[48] S. F. Reddaway, "DAP a Distributed Array Processor," in *Proceedings of the 1st annual symposium on Computer Architecture,(Gainesville, Florida)*. ACM Press, New York, 1973, pp. 61–65.

[49] P. M. Flanders, "Effective use of SIMD processor arrays," in *Parallel Digital Processors*. Portugal: IEE, April 1988, pp. 143–147, iEE Conf. Pub. No. 298.

[50] T. Xu, T. Pototschnig, K. Kuhnlenz, and M. Buss, "A high-speed multi-gpu implementation of bottom-up attention using cuda," in *2009 IEEE International Conference on Robotics and Automation*, Kobe International Conference Centre, Kobe, Japan, May 2009, pp. 41–47.

[51] G. Rizk and D. Lavenier, "Gpu accelerated rna folding algorithm," in *ICCS '09: Proceedings of the 9th International Conference on Computational Science*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 1004–1013.

[52] S. A. Manavski and G. Valle, "Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment," *BMC Bioinformatics*, vol. 9, pp. 1–9, 2007.

[53] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. New York, NY, USA: ACM, 2009, pp. 79–84.

[54] T. D. Hartley, U. Catalyurek, A. Ruiz, F. Igual, R. Mayo, and M. Ujaldon, "Biomedical image analysis on a cooperative cluster of gpus and multicores," in *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*. New York, NY, USA: ACM, 2008, pp. 15–25.

[55] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "Gpu cluster for high performance computing," *SC Conference*, vol. 0, p. 47, 2004.

[56] M. Garland, S. L. Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel computing experiences with cuda," *IEEE Micro*, vol. 28, no. 4, pp. 13–27, 2008.

[57] J. C. Phillips, J. E. Stone, and K. Schulten, "Adapting a message-driven parallel application to gpu-accelerated clusters," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–9.

[58] J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM Journal of Research and development*, vol. 49, no. 4/5, pp. 589–604, July 2005.

[59] M. Scarpino, *Programming the Cell Processor for Games, Graphics and Computation*. Prentice Hall, 2009, no. ISBN 978-013-600886-6.

[60] D. Shippy and M. Phipps, *The Race for a New Game Machine*. Citadel Press, 2009, no. ISBN 978-0-8065-3101-4.

[61] Free Software Foundation, "The GNU compiler collection." 2007. [Online]. Available: http://gcc.gnu.org

[62] J. Kurzak, A. Buttari, P. Luszcek, and J. Dongarra, "The playstation 3 for high-performance scientific computing," *Computing in Science and Engineering*, vol. 10, no. 3, pp. 84–87, May/June 2008.

[63] A. Buttari, J. Dongarra, and J. Kurzak, "Limitations of the playstation 3 for high performance cluster computing," University of Tennessee - Knoxville, Tech. Rep., 2007.

[64] G. Danese, F. Leporati, M. Bera, M. Giachero, N. Nazzicari, and A. Spelgatti, "An accelerator for physics simulations," *Computing in Science and Engineering*, vol. 9, no. 5, pp. 16–25, September 2007.

[65] J. A. Bower, D. B. Thomas, W. Luk, and O. Mencer, "A reconfigurable simulation framework for financial computation," 2006, imperial College.

[66] M. AB, "Low power hybrid computing for efficient software acceleration," Mitrion by Mitrionics, Tech. Rep., 2008. [Online]. Available: www.mitrionics.com

[67] ClearSpeed, "Csx processor architecture," ClearSpeed, Tech. Rep., 2006. [Online]. Available: www.clearspeed.com

[68] Y. Nishikawa, M. Koibuchi, M. Yoshimi, K. Miura, and H. Amano, "Performance improvement methodology for clearspeed's csx600," in *Proc. Int. Conf. on Parallel Processing (ICPP2007)*, 10-14 September 2007, p. 77.

[69] V. Heuveline and J. Weiss, "Lattice boltzmann methods on the clearspeed advance accelerator board," *The European Physical Journal - Special Topics*, vol. 171, no. 1, pp. 31–36, April 2009.

[70] K. Hawick and C. Scogings, *Agent-Based Evolutionary Search.* Springer, January 2010, no. ISBN 978-3-642-13424-1, ch. Complex Emergent Behaviour from Evolutionary Spatial Animat Agents, pp. 139–160, cSTN-067.

[71] 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.

[72] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, no. 6684, pp. 440–442, June 1998.

[73] S. Milgram, "The Small-World Problem," *Psychology Today*, vol. 1, pp. 61–67, 1967.

[74] M. E. J. Newman, "The Structure and Function of Complex Networks," *SIAM Review*, vol. 45, no. 2, pp. 167–256, June 2003.

[75] *NVIDIA CUDA^{TM} Programming Guide Version 2.3*, NVIDIA® Corporation, 2009, last accessed August 2009. [Online]. Available: http://www.nvidia.com/

[76] D. J. d. Price, "Networks of Scientific Papers," *Science*, vol. 149, no. 3683, pp. 510–515, July 1965.

[77] A.-L. Barabasi and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, October 1999.

[78] R. Albert, H. Jeong, and A.-L. Barabsi, "Diameter of the World-Wide Web," *Nature*, vol. 401, no. 6749, pp. 130–131, September 1999.

[79] P. L'Ecuyer, "Software for uniform random number generation: distinguishing the good and the bad," in *Proc. 2001 Winter Simulation Conference*, vol. 2, 2001, pp. 95–105.

[80] G. Marsaglia, "Random Number generation," florida Preprint, 1983.

[81] BSD, *Random - Unix Manual Pages*, June 1993.

[82] M. Matsumoto and T. Nishimura, "Mersenne twistor: A 623-diminsionally equidistributed uniform pseudorandom number generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8 No 1., pp. 3–30, 1998.

[83] G. Marsaglia and A. Zaman, "Toward a universal random number generator," 1987, FSU-SCRI-87-50, Florida State University.

[84] K. Binder and D. W. Heermann, *Monte Carlo Simulation in Statistical Physics.* Springer-Verlag, 1997.

[85] G. Amdahl, "Validity of single-processor approach to achieving large-scale computing capability," in *Proc. AFIPS Conference, Reston VA, USA.*, 1967, pp. 483–485.

[86] M. Hill and M. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, July 2008.

[87] J. Gustafson, "Reevaluating amdahl's law," *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, 1988.

[88] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Symp. Principles and Practice of Parallel Programming.* ACM, 1995, pp. 207–216.

[89] S. Oaks and H. Wong, *Java Threads*, 1st ed., ser. Nutshell Handbook. United States of America: O'Reilly & Associates, Inc., 1997, iSBN 1-56592-216-6.

[90] K. Hawick, D. Playne, A. Leist, G. Kloss, and C. Scogings, "Multi-core software for scientific simulations," Massey University, Tech. Rep. CSTN-070, January 2009.

[91] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical Linear Algebra on Emerging Architectures: the PLASMA and MAGMA Projects," *Journal of Physics: Conference Series*, vol. 180, 2009.

[92] K. Dowd and C. Severance, *High Performance Computing*, 2nd ed. O'Reilly, 1998, no. ISBN 1-56592-312-X.

[93] K. A. Hawick and D. P. Playne, "Hypercubic Storage Layout and Transforms in Arbitrary Dimensions using GPUs and CUDA," Computer Science, Massey University, Tech. Rep. CSTN-096, 2010, accepted for and to appear in Concurrency and Computation: Practice and Experience.

[94] K. A. Hawick, "Erlang and distributed meta-computing middleware," Computer Science, Massey University, Tech. Rep. CSTN-092, 2009.

[95] K. A. Hawick and D. P. Playne, "Automatically Generating Efficient Simulation Codes on GPUs from Partial Differential Equations," Computer Science, Massey University, Tech. Rep. CSTN-087, 2010, submitted to Springer J. Sci. Computing.

[96] K. A. Hawick, A. Leist, D. P. Playne, and M. J. Johnson, "Comparing intra- and inter-processor parallelism on multi-core cell processors for scientific simulations," Computer Science, Massey University, Tech. Rep. CSTN-102, 2009.