Cell Biology
2015; 3(3): 38-49
Published online October 28, 2015 (http://www.sciencepublishinggroup.com/j/cb)
doi: 10.11648/j.cb.20150303.11
ISSN: 2330-0175 (Print); ISSN: 2330-0183 (Online)

Performance Optimisations for a Numerical Solution to a 3D Model of Tumour-Induced Angiogenesis on a Parallel Programming Platform

Paul M. Darbyshire

Department of Computational Biophysics, Algenet Cancer Research, Nottingham, UK

Email address:
rd@algenet.com

To cite this article:
Paul M. Darbyshire. Performance Optimisations for a Numerical Solution to a 3D Model of Tumour-Induced Angiogenesis on a Parallel Programming Platform. *Cell Biology*. Vol. 3, No. 3, 2015, pp. 38-49. doi: 10.11648/j.cb.20150303.11

Abstract: The challenging issues of cancer prevention and cure lie in the need for a more detailed knowledge of the dynamic processes and mechanisms of cellular behaviour and tumour growth dynamics. In this paper we extend a previous 2D parallel implementation of a continuous-discrete model of tumour-induced angiogenesis to the more realistic 3D case. In particular, we look in-depth at available performance optimisation techniques to further improve the computational method and explore in more detail the hardware architecture. Recent evidence clearly indicates that GPU-accelerated computing can greatly facilitate researchers, clinicians and oncologists by performing time-saving *in-silico* experiments that have the potential to assist in quantifying cellular parameters, highlight model features, and help explore new cancer treatments and therapies.

Keywords: Tumour-Induced Angiogenesis, Compute Unified Device Architecture (CUDA), Graphical Processing Unit (GPU), High-Performance Computing (HPC)

1. Introduction

Over the last decade, *high-performance computing* (HPC) has evolved dramatically, in particular because of the accessibility to graphics processing units (GPUs) and the emergence of GPU-CPU *heterogeneous* architectures, which have led to a fundamental shift in parallel programming. Finite difference methods (FDM), such as those developed here, are the first port of call for solving complex biological phenomenon described by nonlinear partial differential equations (PDEs). However, they require intensive computational resources which generally lead to significant and time-consuming expense. The advantages of explicit time-stepping in FDM over many other types of solutions lend themselves well to exploitation in a completely *data-parallel* context. In such cases, GPUs can be used to greatly accelerate numerical simulations and offer an extremely valuable computational technique for tackling such problems. The *compute unified device architecture* (CUDA) programming model is especially well-suited to address problems that can be expressed as data-parallel computations.

In a previous paper the authors developed a 2D finite difference approximation to a hybrid continuous-discrete model of tumour-induced angiogenesis [2]. The numerical solution was implemented in both C++ and CUDA C to assess the performance benefits of porting from a serial to a parallel programming platform. Results indicated a dramatic increase in execution time between the two implementations and also highlighted a range of potential performance improvements available through more advanced data manipulation and memory management techniques. In this paper, the authors develop a 3D finite difference approximation to the same hybrid continuous-discrete model and implement some of these advanced features with a view to highlighting the potential benefits of modelling cellular and cancer dynamics whilst also highlighting the possibilities for developing new advanced clinical research tools based on GPU-accelerated applications. Indeed, in the last decade *in silico* trials focussed on simulating the different processes of solid tumour growth have become more readily accepted by the clinical and oncology community. The advantages of using GPU-accelerated programs and HPC continually highlight the potential performance improvements in solving complex mathematical models of biological phenomenon in this way [1, 2].

In order to progress from the relatively harmless avascular phase to the potentially lethal vascular state, solid tumours must induce the growth of new blood vessels from existing

ones, a process known as *angiogenesis*. While early models of angiogenesis were focused on accurately replicating key observed behaviours during the process, more recent models have been able to test specific hypotheses and suggest useful strategies for antiangiogenic drug development. A key mechanism of antiangiogenic therapy is to interfere with the process of blood vessel growth and literally starve the tumour of its blood supply. Indeed, a new class of cancer treatments that block angiogenesis have recently been approved and available to treat cancers of the colon, kidney, lung, breast, liver, brain, ovaries and thyroid [3-7]. Angiogenesis is without doubt a complex biological phenomena and one that at a cellular level is dynamic, spatially heterogeneous, frequently non-linear, and spans many orders of magnitude, both spatially and temporally. Mathematical and computational models of vascular formation have generated a basic understanding of the processes of capillary assembly and morphogenesis during tumour development and growth [8, 9]. However, by the time a tumour has grown to a size whereby it can be detected by clinical means, there is a strong likelihood that it has already reached the vascular growth phase and developed its own blood circulatory network. For this reason, a thorough understanding of the behavioural processes of angiogenesis is essential. The development of realistic mathematical and computational models of cancer dynamics is a powerful method of testing hypotheses, confirming biological experiments, and simulating complex behaviour. The model presented here is of a hybrid nature in which a system of couple nonlinear partial differential equations (PDEs) describing continuous chemical and macromolecular dynamics and a discrete cellular automata-type model controls cell migration and their interaction with neighbouring cells [10]. The main objective of the paper is to extend the work developed in [2] to the numerical solution of the hybrid continuous-discrete model describing tumour-induced angiogenesis to the more realistic 3D case. We also wish to address ways in which parallel performance can be optimised by making use of the explicit GPU hardware architecture and CUDA programming model.

2. A Continuous-Discrete Model of Tumour-Induced Angiogenesis

2.1. The Continuous Model

For a more detailed treatment of the biological aspects of tumour-induced angiogenesis as well as a more rigorous mathematical proof, readers are directed to [2, 10] and references therein. Here we simply summarise the main mathematical development so as to focus on the main issues of the paper. If we denote the endothelial cell density by n, the TAF and fibronectin concentration by c and f, respectively the complete system of scaled coupled nonlinear PDEs describing tumour-induced angiogenesis can be written as [2, 10]:

$$\frac{\partial n}{\partial t} = D\nabla^2 n - \nabla \cdot (\chi(c) n \nabla c) - \rho \nabla \cdot (n \nabla f) \quad (1)$$

$$\frac{\partial f}{\partial t} = \beta n - \gamma n f \quad (2)$$

$$\frac{\partial c}{\partial t} = -\eta n c \quad (3)$$

A description of each of the parameters, and their respective values, can be found in [2, 10]. Our system is assumed to hold on a 3D spatial domain Ω (i.e., a volume of tissue) with appropriate initial conditions; $c(x, y, z\ 0)$, $f(x, y, z\ 0)$ and $n(x, y, z\ 0)$ [2, 10]. The tumour cells are assumed to be confined within a domain $\Omega \in [0,1]^3$ in which *no-flux* (Neumann) boundary conditions are imposed on the boundaries of Ω (see Figure 1).

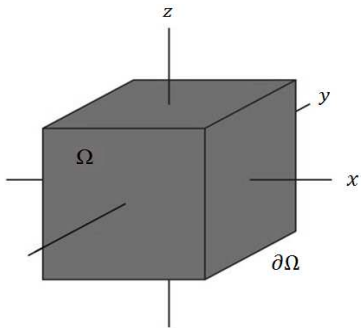

Figure 1. *A schematic diagram of the 3D spatial domain Ω and boundary ∂Ω.*

2.2. The Discrete Model

The technique of tracing the path of an individual endothelial cell at a sprout tip was first proposed by Anderson *et al.* [11]. The method involves using standard FDM to discretise the continuous model described in (1)-(3) over a 3D uniform grid. Then, the resulting coefficients of the finite difference seven-point stencil are used to generate the probabilities of movement of an individual endothelial cell in response to its local microenvironment. 3D stencil computations are those in which each node in a 3D grid is updated with a weighted average of the six neighbouring node values. Two schematic diagrams of a 3D finite difference seven-point stencil are shown in Figure 2.

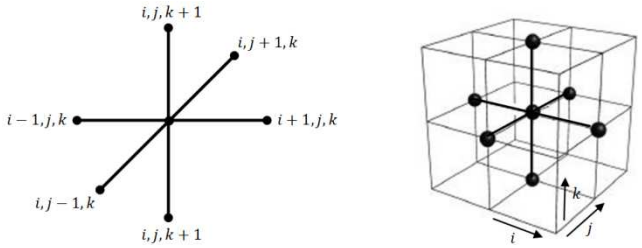

Figure 2. *Schematic diagram of the finite difference 7-point 3D stencil.*

We first discretise the continuous model by approximating the 3D domain $\Omega \in [0,1]^3$ on a uniform grid of node length, width and depth h, and time t by increments of size k. By applying a *forward* finite difference scheme, the *fully-explicit* discretised version of the continuous model can be obtained.

For illustration purposes, the endothelial cell discretisation is shown below:

$$n_{i,j,k}^{q+1} = n_{i,j,k}^{q}P_0 + n_{i+1,j,k}^{q}P_1 + n_{i-1,j,k}^{q}P_2 + n_{i,j+1,k}^{q}P_3 + n_{i,j-1,k}^{q}P_4 + n_{i,j,k+1}^{q}P_5 + n_{i,j,k-1}^{q}P_6 \quad (4)$$

The coefficients P_0–P_6 can be thought of as being proportional to the probabilities of endothelial movement. That is, the coefficient P_0, is proportional to the probability of no movement, and the coefficients P_1, P_2, P_3, P_4, P_5, and P_6, are proportional to the probabilities of moving left, right, up and down, out of and into the plane, respectively. The exact forms of P_0–P_6 are functions of both fibronectin and TAF concentrations at nearby neighbouring points of an individual endothelial cell [2, 10].

Each numerical simulation is based on an increased size of array width i.e., a finer grained uniform 3D grid. We use a constant iteration size of 1,000 time steps to allow for an adequate convergence of the numerical solution. At each time step, the numerical simulation involves solving the discrete model to generate the seven coefficients P_0–P_6. Based on the values of these coefficients, a set of seven probability ranges are determined and then a uniform random number is then generated on the interval [0, 1], and, depending on the range into which this value falls, the current individual endothelial cell will remain stationary (R_0), move left (R_1), right (R_2), move up (R_3), down (R_4), out of (R_5), or into the plane (R_6). The complete set of parameter values used for the numerical simulation can be found in [2, 10].

3. Implementation

3.1. The Kepler GK110 Architecture

The GPU is specialised for computer-intensive, highly data parallel computations allowing more transistors to be devoted to data processing rather data caching and flow control. More specifically, the GPU is especially well-suited to address problems that can be expressed such that the same algorithm is executed on many data elements in parallel, with high arithmetic intensity i.e., the ratio of arithmetic operations to memory operations. Since the algorithm is executed on many data elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches. The recent rollout of the Nvidia *Kepler GK110* architecture marked a significant milestone in the evolution of GPU-accelerated computing. By offering much higher processing power than previous architectures and by providing new methods to optimise and increase parallel workload execution on the GPU, the Kepler GK110 has further revolutionised HPC. Each of the Kepler GK110 streaming multiprocessor (SMX) units feature 192 single-precision CUDA cores, and each core has fully pipelined floating-point and integer *arithmetic logic units* (ALU). Figure 3 shows the major differences between the CPU and GPU architectures in terms of ALU, cache and *dynamic random access memory* (DRAM) layout [12].

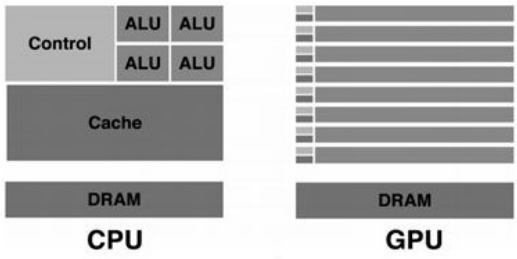

Figure 3. *A schematic of the CPU vs. GPU architecture [12].*

Applications running on Kepler GK110 can also take advantage of the increased number of registers available to each thread to increase instruction level parallelism. The Kepler GK110 features a large dedicated L2 cache memory, double the amount of L2 available with previous architectures. The L2 cache is the primary point of data unification between the SMX units, servicing all load, store, and texture requests and providing efficient, high speed data sharing across the GPU [12]. Table 1 shows some specifications for the hardware architecture in the Kepler GK 110 architecture.

Table 1. *Specifications for the Kepler GK 110 architecture [12].*

Specification	Value
Warp size	32
Threads/multiprocessor	2,048
Threads/block	1,024
Global memory	3 GB
L2 cache memory	1,536 KB
Constant memory	64 KB
Read-only data cache	48 KB

The SMX schedules threads in groups of 32 parallel threads known as *warp*s. Each SMX features four warp schedulers and eight instruction dispatch units, allowing four warps to be issued and executed concurrently [12]. Figure 4 shows a schematic of how warps are scheduled in the Kepler GK110 architecture.

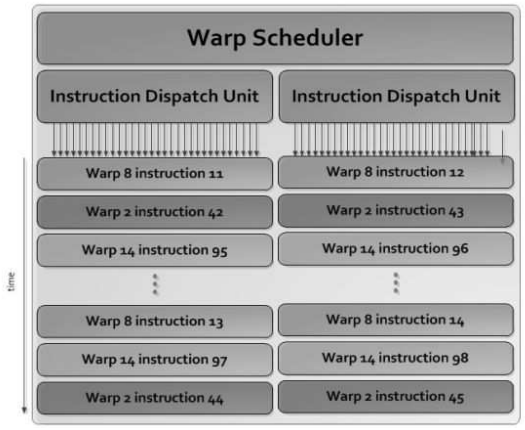

Figure 4. *A schematic of warp scheduling in the Kepler GK 110 architecture [12].*

The Kepler GK110 memory hierarchy is organised similarly way to earlier architectures as shown in Figure 5 and also enables compiler-directed use of an additional new

cache for read-only data (see Table 1).

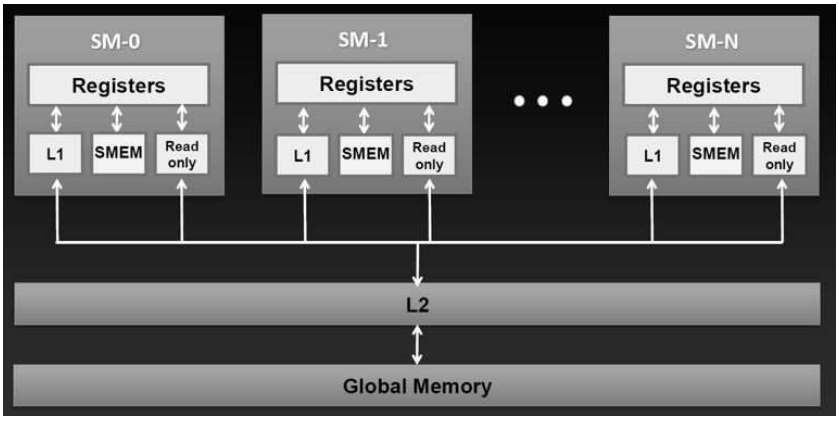

Figure 5. *Hierarchal organisation of memory in the Kepler GK 110 architecture.*

When writing parallel programs, it is often necessary to communicate values between parallel threads. The typical way to do this in the CUDA programming model is to use *shared mem*ory. However, the Kepler GK110 architecture introduced a way to directly share data between threads that are part of the same warp i.e., threads in a warp can read other registers by using a new instruction called SHFL, or shuffle. Firstly, it is possible to use the shuffle instruction to free up shared memory to be used for other data. Secondly, the shuffle instruction is faster than shared memory since it only requires one instruction versus three for shared memory (write, synchronise, read). Another potential performance advantage for shuffle is that relative to older architectures, shared memory bandwidth has doubled on Kepler devices and the number of cores has increased by 6×; therefore, the shuffle instruction provides another means to share data between threads and keep the cores busy with memory accesses that have low latency and high bandwidth.

3.2. Hardware Specifications

The hardware used for the serial C++ implementation is a fourth generation Intel® Quad Core™ i7-4790K 4GHz CPU processor. The C++ implementation was developed and compiled in Microsoft® Visual Studio 2012. The CUDA C++ program was also developed in Microsoft® Visual Studio 2012 using CUDA version 7.0 and tested on an Nvidia GeForce® GTX™ 780 GPU based on the Kepler GK110 architecture with Compute Capability 3.5. The Compute Capability describes the features of the hardware and reflects the set of instructions supported by the device as well as other specifications, such as the maximum number of threads per block and the number of registers per multiprocessor. Moreover, hardware design, number of cores, cache size, and supported arithmetic instructions are different for different versions of Compute Capability. Higher compute capability versions are supersets of lower (i.e., earlier) versions, so they are backward compatible. The operating system for both configurations is Windows 8.1. Table 2 shows some hardware specifications for the Nvidia GeForce® GTX™ 780 GPU.

Table 2. *GPU hardware specifications.*

Specification	Nvidia GeForce® GTX™ 780
GPU clock speed	0.863 GHz
Memory clock rate	3.004 Ghz
CUDA cores	2,304
Memory interface	384-bit
Peak performance*	3.98 Tflops
Memory bandwidth	288.4 GB/s
* Peak single-precision floating-point performance	

3.2.1. Memory Bandwidth

Bandwidth is usually used to describe the highest possible amount of data transfer per unit time, while *throughput* can be used to describe the rate of any kind of information or operations carried out per unit time, such as, how many instructions are completed per cycle. Limited memory bandwidth can become a serious *bottleneck* to GPU performance and while a GPU typically has far greater memory bandwidth than a CPU, maximising the use of this bandwidth is still a critical issue. If an algorithm spends more time computing than transferring data, then it may be possible to overlap these operations and completely hide the *latency* associated with transferring data. On the other hand, if the algorithm spends less time computing than transferring data, it is important to minimise transfer between the CPU and GPU. In general, whilst performing code optimisation, it is important to determine how the application compares to theoretical limits. Theoretical memory bandwidth can be calculated using:

$$\text{memory clock rate} \times \text{bus interface width} \times \text{data rate} \quad (5)$$

Since the Kepler 110 architecture relies on the graphics double data rate random access memory type i.e., GDDR5, the theoretical memory bandwidth for the GTX™ 780 GPU card is 288.4 GB/s $(= 3.004 \times (384/8) \times 2)$. Note that bus interface width has been converted to bytes.

3.2.2. Instruction Throughput

Two types of floating-point numbers are typically used in algorithms, single-precision *floats* and double precision *doubles*. Single precision requires 32 bits (4 bytes) of storage

and has an accuracy around 7 decimal places. Double precision requires 64 bits (8 bytes) and achieves an accuracy around 16 decimal places. Such large discrepancies between the two types of numbers have a significant impact on numerical simulations. That is, nine decimal places of information are lost when using only single-precision, and when implementing iterative procedures, such as FDM, this can introduce large errors. Results obtained using double-precision calculations will frequently differ from the same operation performed using single-precision arithmetic due to rounding issues. Therefore, it is important to be sure to compare values of like precision and to express the results within a certain tolerance rather than assuming them to be exact. Indeed, we can estimate a lower-bound to the performance of our CUDA C implementation by estimating the (*giga*) *floating point operations per second* (Gflops). Gflops are a measure of processing speed, equal to the number of operations the CPU and GPU can perform per second. In general, a processor can do a certain number of Gflops every time its internal clock ticks (or *cycle*). It is important to note that there is quite a difference between *single-precision* and *double-precision* Gflops. A processor that is capable of many single-precision Gflops may only be capable of a small fraction of that many double-precision calculations. We assume the following general formula to determine the number of Gflops for our CPU and GPU processors, given by:

$$\text{clock speed} \times \#\ \text{cores} \times \text{flops per clock cycle} \qquad (6)$$

For the GTX™ 780 GPU card, we get 3977 Gflops ($= 0.863 \times 2304 \times 2$) i.e., 3.98 Tflops single-precision. With the GK110 architecture, double-precision performance is fixed at 1/24 that of single-precision performance i.e., 166 Gflops double-precision. Based on these values, the estimated performance improvement between serial and parallel implementations, in terms of Gflops calculations alone, should be at least in the region of 31×. Note that, in addition to accuracy, the relative conversion between double and floating point numbers (and vice versa) can also have a detrimental effect on performance.

We can also measure the performance of an algorithm in terms of its *compute to memory access* ratio (CMA). Many numerical algorithms, such as FDM, have a very low CMA of around 1.0, implying there is a read or write to memory for every floating-point operation. For the GTX™ 780 GPU card, which has a memory bandwidth of 288.4 GB/sec. At single precision (4 bytes) the maximum transfer rate will be 72.1 Gflops. With a CMA of 1.0, this gives a calculated flop rate of 72.1 Gflops, far less than the theoretical maximum of 3.5 Tflops. In order to achieve optimal memory bandwidth, it is vital to ensure that memory Is effectively managed, which when correctly managed, can lead to substantial increases in data transfer rates, and is vital for delivering performance that is close to the theoretical maximum.

In a GPU, a SMX relies on thread-level parallelism to maximise *utilisation* of its functional units. Utilisation is therefore directly linked to the number of resident warps. The number of clock cycles between an instruction being issued and being completed is defined as instruction *latency*. Full compute resource utilisation is achieved when all warp schedulers have an eligible warp at every clock cycle. This ensures that the latency of each instruction can be hidden by computation from other warps. Whilst bandwidth is usually used to describe the highest possible amount of data transfer per unit time, while *throughput* can be used to describe the rate of any kind of information or operations carried out per unit time, such as, how many instructions are completed per cycle. Another useful performance metric is the ratio of instructions to bytes. For the GTX™ 780 GPU card, the theoretical ratio is 13.8 instructions: 1 byte (= 3.98/288.4) i.e., if an application issues more than 13.8 instructions for every byte accessed, then it is bound by arithmetic performance. However, most GPU-accelerated workloads, are bound by memory bandwidth.

4. The CUDA Programming Model

The CUDA programming model involves running code on two different platforms concurrently; a *host* system (the CPU) and a *device* (the GPU). While GPUs are frequently associated with graphics, they are also powerful arithmetic engines capable of running thousands of lightweight threads in parallel. This capability makes them well suited to computations that can leverage parallel execution. Nowadays, modern GPUs can support up to 2,304 active threads concurrently per multiprocessor. So, for a GPU with 12 multiprocessors, this leads to more than 27,000 concurrently active threads. Threads on a CPU are generally heavyweight entities. The operating system must swap threads on and off CPU execution channels to provide multithreading capability. Context switches (i.e., when two threads are swapped) are subsequently slow and expensive. On GPUs, threads are extremely lightweight. In a typical system, thousands of threads are queued up for work in sets of 32 threads each (i.e., warps). If the GPU must wait on one warp of threads, it simply begins executing work on another. Since separate *registers* are allocated to all active threads, no swapping of registers or other state need occur when switching among GPU threads. Resources stay allocated to each thread until it completes its execution. In short, CPU cores are designed to minimise latency for one or two threads at a time, whereas GPUs are designed to handle a large number of concurrent, lightweight threads in order to maximise throughput. The host system and the device each have their own distinct attached physical memories. As the host and device memories are separated by the *PCI Express* (PCIe) bus, data in the host memory must be communicated across the bus to the device memory. Such continually data transfers usually result in memory *bottlenecks* which can lead to serious performance issue when developing GPU-accelerated applications.

The CUDA programming model provides an *application program interface* (API) that exposes the underlying GPU architecture; a collection of *single instruction, multiple data*

(SIMD) processors capable of executing thousands of *threads* in parallel. A version of SIMD used by GPUs is the *single instruction, multiple threads* (SIMT) architecture in which multiple threads execute an instruction sequence. In CUDA C, an instruction sequence is written into a specific function known as a *kernel* that can be executed on a device N times in parallel by N different CUDA threads, *asynchronously*. Unlike a C function call, all CUDA kernel launches are asynchronous so that control returns to the CPU immediately after the CUDA kernel is invoked. An *execution configuration* defines both the number of threads that will run the kernel plus their arrangement in a 1D, 2D, or 3D computational grid. In its simplest form, the kernel is defined using the following CUDA C syntax [13, 14]:

__global__ *kernel*<<<dimGrid, dimBlock>>>();

Threads are grouped into *blocks* and blocks are grouped into *grid*s as shown schematically in Figure 6. There is a limit to the number of threads per block, for the Kepler GK110 architecture a thread block may contain up to 1,024 threads. On the GPU, each multiprocessor is responsible for handling one or more blocks in a grid which is further divided into a number of streaming processors each handling one or more threads in a block.

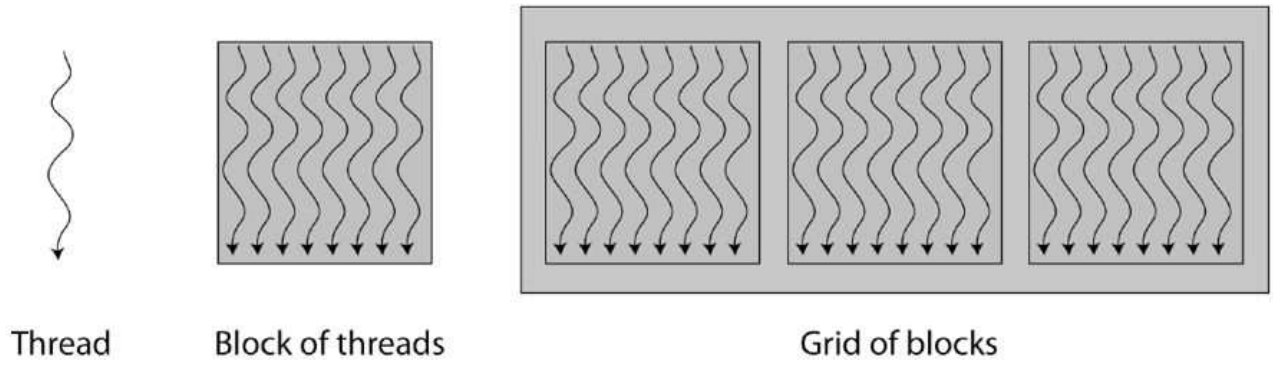

Figure 6. *A schematic representation of threads, blocks and grids.*

In general we want to size our blocks and grids to match data requirements and simultaneously maximise *occupancy*. Occupancy measures the efficiency to which we assign how many threads are active at any one time. The major factors influencing occupancy are efficient memory allocation and thread block size. Clearly, thread block size should always be a multiple of 32, since threads are scheduled in warps. For example, if we have a block size of 50 threads, the GPU will still issue commands to 64 threads and this would just be waste of resources. It is often necessary to try and size blocks based on the maximum numbers of threads and blocks corresponding to the Compute Capability of the GPU. The theoretical occupancy is the ratio of active warps to the maximum warps for a SMX. Each multiprocessor on a device has a set of N registers available for use by CUDA thread programs. These registers are a shared resource that is allocated amongst thread blocks executing on a multiprocessor. The CUDA compiler attempts to minimise register usage to maximise the number of thread blocks that can be active simultaneously. If a program tries to launch a kernel for which the registers used per thread times the block size is greater than N, the launch will fail. Varying the size of the thread block is a standard optimisation to find the best occupancy rates. Moreover, high occupancy rates help to hide the latency in accessing global memory.

A block is 1D, 2D, or 3D with the maximum size of the x, y, and z dimensions being 1,024, 1,024, and 64, respectively, such that $x \times y \times z \leq 1{,}024$ i.e., the maximum number of threads per block. Blocks are subsequently organised into a 1D, 2D or 3D grid with the maximum size of the x, y, and z dimensions being 2^{31}-1, 65,535, and 65,535, respectively. An example schematic of a block and grid set up is shown in Figure 7. There are also a maximum of 65,536 registers available per block.

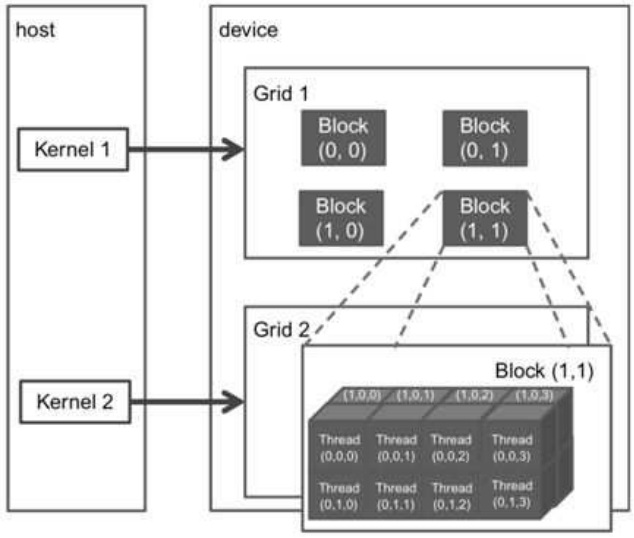

Figure 7. *An example CUDA thread grid and block.*

5. Performance Optimisation

Optimising the performance of CUDA applications most often involves optimising data accesses which includes the appropriate use of the various available *memory spaces* (see Figure 8) of the GPU architecture. Indeed, appropriate use of these memory spaces can have significant performance implications for almost every CUDA application.

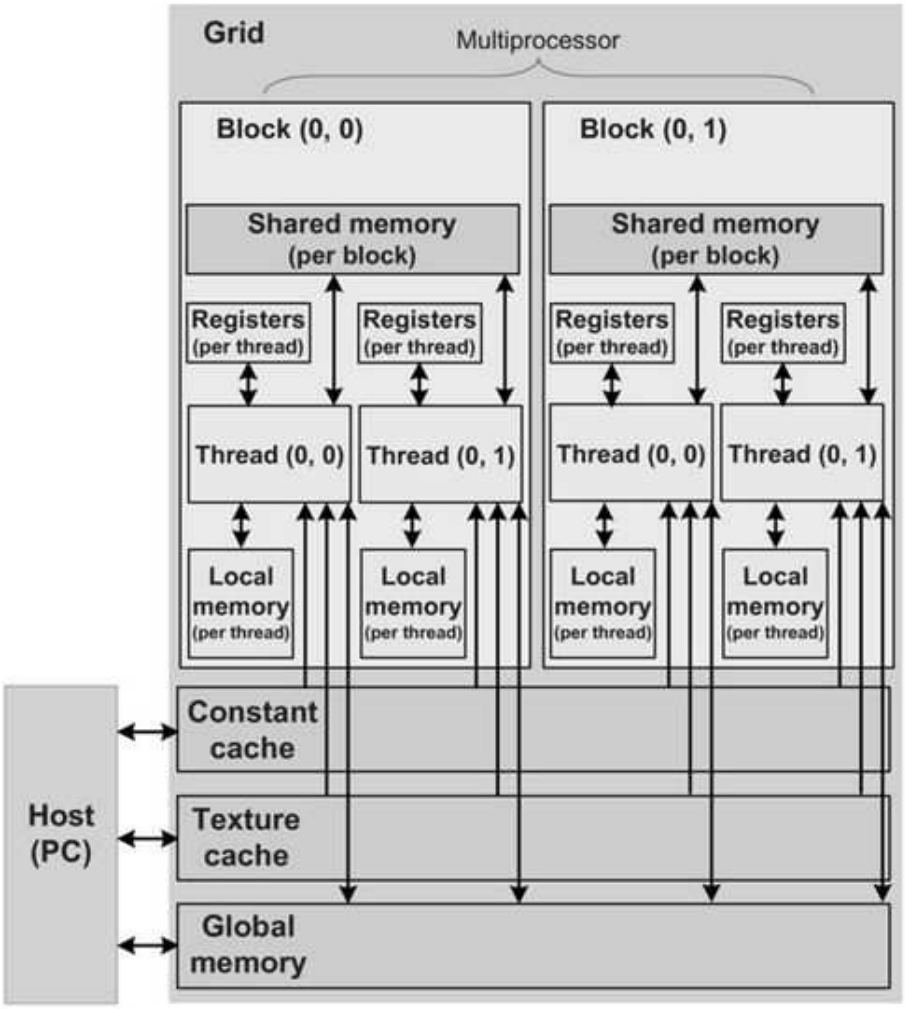

Figure 8. *Schematic of the arrangement of available memory spaces.*

Note that Figure 8 includes blocks labelled *local memory* within the multiprocessor. Local memory implies *local in the scope of each thread*. It is a memory abstraction, not an actual hardware component. In actuality, local memory gets allocated in global memory by the compiler and delivers the same performance as any other global memory region. The local and global memory spaces are *not cached* which means each memory access to global (or local) memory generates an explicit memory access.

5.1. Global Memory Coalescing

The latency in accessing global memory can be considerable. Although the bandwidth of global memory seems high, around 200-300 GB/s, it is very slow compared to the Tflop performance capability of a typical GPU. Global memory is implemented with *dynamic random access memories* (DRAM) using a parallel access process i.e., each time a memory location is accessed, a number of other memory locations (that include the requested location) are also accessed. If an application utilises data from consecutive accessed locations before accessing other locations, the DRAM can achieve near peak global memory bandwidth. Therefore, global memory delivers the highest memory bandwidth only when the global memory accesses can be *coalesced*. The performance penalty for *non-coalesced* memory operations varies according to the size of the data type (e.g., 4-bytes). Each active block is split into SIMD groups of threads; warps. Each warp contains the same number of threads i.e., *warp size*, which are executed by the multiprocessor in a SIMD manner. This means each thread within a warp is *broadcast* the same instruction from the instruction store, which directs the thread to perform some operation or manipulation of local and/or global memory. Active warps are *time-sliced*; the thread scheduler periodically switches from one warp to another to maximise the use of the multiprocessor's hardware resources (see Figure 9). The order of execution of the warps within a block and of blocks themselves is undefined, which means they can occur in any order. Moreover, all threads in a warp execute the same instruction. When all threads in a warp execute a load instruction, the hardware checks if the threads are accessing consecutive memory locations. Ideally, thread 0

accesses location n, thread 1 accesses location $n + 1$, ..., thread 31 accesses location $n + 31$, then all accesses are coalesced and combined into one single contiguous access.

Consider the case when the warp scheduler requests 32, *aligned* consecutive 4-byte words from global memory. Each memory address will fall into 4 segments of 128 bytes each as shown in Figure 10.

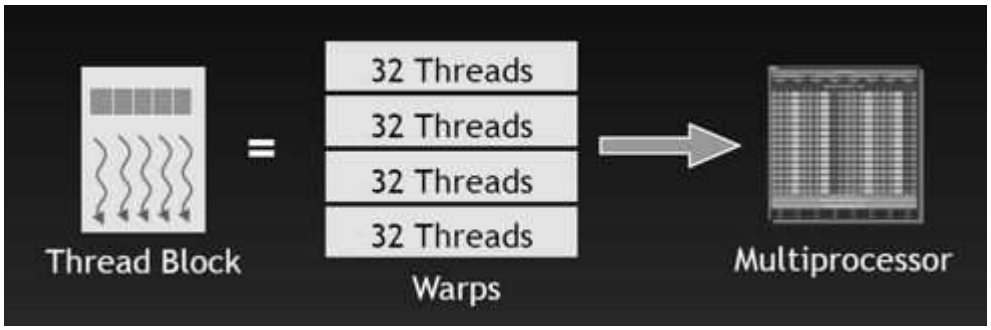

Figure 9. *A schematic diagram of thread scheduling.*

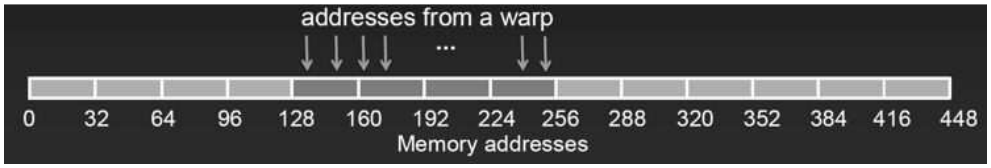

Figure 10. *A schematic diagram showing consecutive global memory access.*

The consecutive alignment scenario in Figure 10 will result in 100% coalesced global memory access. Now consider the case when the warp scheduler requests 32, *permutated* consecutive 4-byte words from global memory as shown in Figure 11. Each memory address will still fall into 4 segments of 128 bytes and also achieve 100% coalesced global memory reads.

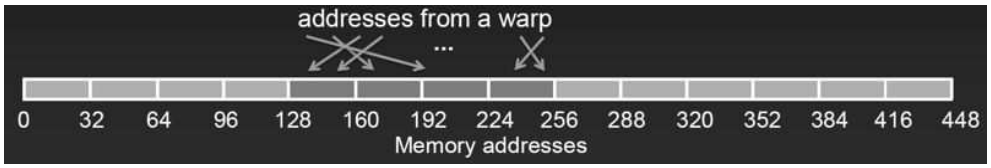

Figure 11. *A schematic diagram showing permutated global memory access.*

Figure 12 shows the case when memory addresses are misaligned consecutive 4-byte words. In this case, each memory address now falls into at most 5 segments of 160 bytes which results in a lower utilisation of global memory reads (80%).

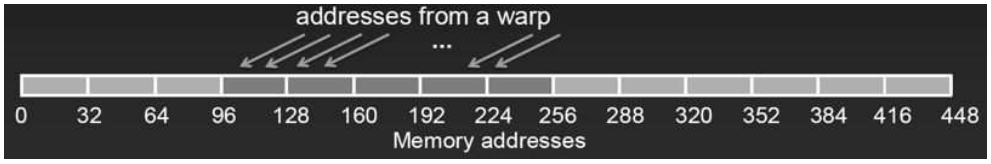

Figure 12. *A schematic diagram showing misaligned global memory access.*

Finally, consider the case when memory addresses are *scattered* as shown in Figure 13. This results in N segments of $N \times 32$ bytes and a severe non-coalesced global memory access.

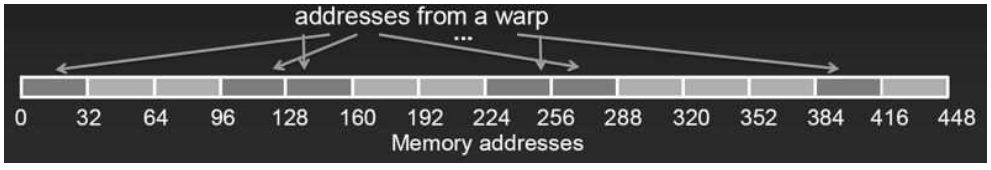

Figure 13. *A schematic diagram showing misaligned global memory access.*

It is therefore extremely important to aim for perfect address coalescing i.e., optimised *address patterns*. A warp will generally access a contiguous region of memory so it is necessary to avoid scattered access patterns or those with large *strides* between threads.

5.2. Array Flattening

In general memory allocated dynamically on the GPU cannot use 2D array indexing like you would using C++ i.e., a 2D array declared as: $M[i][j]$, must first be *flattened* into 1D linear array of memory in which each element is indexed from the beginning of the array by determining an *offset*, $M[\text{offset}]$ dependent on indices i, j, and the array width. In general, such memory allocation utilises *row-major order*. Figure 14 shows an example of how a 2D array is flattened into a 1D representation using a row-major order offset.

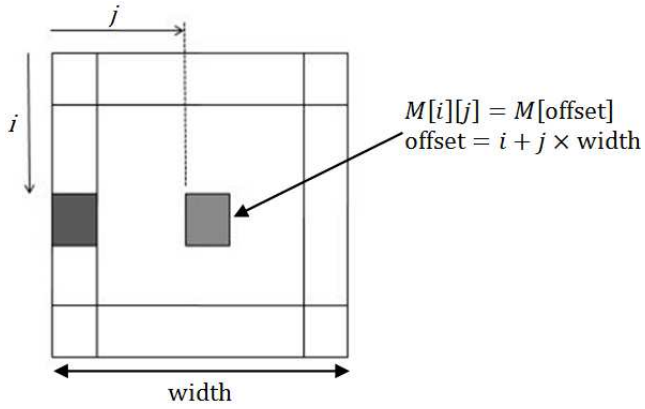

Figure 14. *An example of using an offset to flatten a 2D array into 1D.*

Using linear arrays removes a great deal of the complexity associated with transferring a double pointer (i.e., ** or [][]) array between the host and device. In essence, a nested *deep copy* operation is required in the copy sequence from host to device, such that linearising (or flattening) the data allows it to be referenced using only a single pointer (*). For a 3D array, declared as: $M[i][j][k]$, a similar offset is calculated but now dependent on indices i, j, k and the array width and depth as shown in Figure 15.

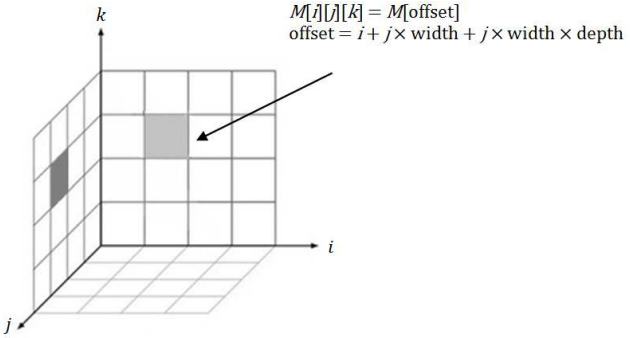

Figure 15. *An example of using an offset to flatten a 3D array into 1D.*

5.3. Texture Cache

Together with memory coalescing in global memory transfers, exploiting shared memory is a further key optimisation. Reusing data stored in shared memory is far more efficient than repeatedly loading from global memory, as long as it can be used efficiently within a thread block. However, with the increases in caching levels for the Kepler GK110 architecture, this has become less critical for certain types of algorithm [12]. For example, *texture memory* provides a surprising aggregation of capabilities including the ability to cache global memory (separate from register, global, and shared memory) and dedicated interpolation hardware separate from the thread processors. Texture memory also provides a way to interact with the display capabilities of the GPU. Since optimised data access is very important to GPU performance, the use of texture memory can (in the right circumstances) provide a large performance increase. The best performance will be achieved when the threads of a warp read locations that are *spatially local*. Moreover, designed primarily for graphics applications, textures are used more generally to maximise memory bandwidth in applications where global memory reads do not satisfy coherency constraints but nonetheless exhibit a high degree of spatial locality [14].

The Kepler GK110 architecture enables applications to utilise texture cache when reading from global memory without actually using the texture reference or texture object APIs. This is done using LDG instruction which is like a global load, except that data is transported through the texture cache instead of the regular L1/L2 cache hierarchy. To allow access to such *bindless textures*, pointers to global memory must be decorated with __const__ and __restrict__ qualifiers. The whole point of __restrict__ is to tell the compiler that two or more pointer arguments will never overlap in memory. Two pointers *alias* if the memory to which they point overlaps, so in an ideal situation we require no redundant memory accesses as a result of pointer aliasing. By decorating a pointer with the restrict property, the programmer is promising the compiler that any data written to through that pointer is not read by any other pointer with the __restrict__ property. In other words, the compiler does not have to worry that a write to a restrict pointer will cause a value read from another restrict pointer to change. Pointer aliasing is something developers need to be extremely aware of on both the GPU and CPU for which proper use can significantly improve performance and code optimisation [13, 15].

5.4. Constant Memory

Constant memory is read only and cached on-chip and has only one read port, but can *broadcast* data from this port across a warp. This means that constant memory access is effective when all threads in a warp read the same address, but when threads in a warp read different addresses the reads are *serialised*. Since constant memory is cached, a read from constant memory costs one memory read from device memory *only* on a cache miss; otherwise, it just costs one read from the constant cache. That is, since constant memory is cached, consecutive reads of the same address will not incur any additional memory traffic. Constant memory is declared using the __constant__ keyword and must be declared outside of the main body of the program and the kernel function. Constant cache is written to only by the host and subsequently initialised in the main body of the program using cudaMemcpyToSymbol(). Constant memory is perfect

for coefficients and other data that are used uniformly across threads [13, 15].

6. C++ and CUDA C Algorithms

The main body of the C++ implementation is shown in Algorithm 1.

Algorithm 1 *C++ main body*

```
Define model parameters
Declare pointers
Initialise array memory
Set initial conditions: n(x, y, 0), f(x, y, 0), c(x, y, 0)
    Start clock()
    for q ← 1 … T do
        Determine coefficients: P0, P1, P2, P3, P4, P5, P6
        Determine probability ranges: R0, R1, R2, R3, R4, R5, R6
        Generate uniform random number [0, 1]
        Store results
        for k ← 1 … Nz-1 do
            for j ← 1 … Ny-1 do
                for i ← 1 … Nx-1 do
                    Update nodes: n_{i,j,k}^{q+1}, f_{i,j,k}^{q+1}, c_{i,j,k}^{q+1}
                end for
            end for
        end for
        Swap pointers
    end for
    End clock()
Print results
Free() array memory
```

Achieving a high-level of performance and optimisation using the CUDA programming model requires careful attention to detail [1, 2, 16-18]. Porting from C++ to CUDA C involves additional coding, as well as some efficient manipulation of the kernel function in respect of thread deployment and memory management. Within the CUDA programming model, CUDA C code is required to initialise memory on the device, and to deal with the transfers of data to the device and back to the host after the kernel execution has completed. In general, there are three steps that are essential to the successfully execution of a kernel on the GPU. Firstly, data must be initialised and transferred from the host to the device global memory. Once the data is on the GPU, the kernel is executed `N` times and launches the required number of `N` threads for the device. When all threads have completed execution (enforced through synchronisation) data is transferred back to the host from the device. In the CUDA programming model, device memory is typically allocated using cudaMalloc()and data is transferred between host and device memory using cudaMemcpy depending on the data flow i.e., either cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost. Memory is subsequently freed after completion using cudaFree(). Making efficient use of available memory (e.g. texture cache, constant memory) can reduce the amount of data that has to be physically transferred between host and device, which is typically the performance bottleneck. The algorithms for the implementation of the CUDA C kernel and the main body are shown in Algorithms 2 and 3.

Algorithm 2 *CUDA C kernel*

```
Get current (i, j, k) global indices and array offset
Determine coefficients: P0, P1, P2, P3, P4, P5, P6
Determine probability ranges: R0, R1, R2, R3, R4, R5, R6
Generate uniform random number [0, 1]
Store results
    if (i < Nx - 1 && j < Ny - 1 && j < Nz - 1 && i > 0 && j > 0 && k > 0) then
        Update nodes: n_{i,j,k}^{q+1}, f_{i,j,k}^{q+1}, c_{i,j,k}^{q+1}
    end if
```

Algorithm 3 *CUDA C main body*

```
Declare global __constants__
Declare host and device pointers
Allocate host and device memory
Set initial conditions: n(x, y, 0), f(x, y, 0), c(x, y, 0)
Copy host arrays to device
Declare dimGrid and dimBlock
Start cudaEventRecord()
    for q ← 1 … T do
        Launch kernel <<< dimGrid, dimBlock >>>
        cudaThreadSynchronise()
        Swap pointers
    end for
Stop cudaEventRecord()
Copy device arrays to host
Print results
Free() host array memory
cudaFree() device array memory
```

The actual performance improvement is be based on the execution time of each of the C++ and CUDA C implementations. Here, the execution time is the difference between two clock statements in each of the C++ and CUDA C algorithms. One placed at the start, and the other at the end of the main looping routine (including the memory transfer in CUDA C). Thus, execution time represents the time taken to complete the entire process of a single simulation of the numerical solution to the hybrid continuous-discrete model. With CUDA C, it is important to remember that calls to kernels are asynchronous. Therefore, to accurately measure the elapsed time for a particular call or sequence of CUDA calls, it is necessary to synchronise the CPU thread with the GPU by calling cudaDeviceSynchronize() immediately before starting and stopping the CPU timer. cudaDeviceSynchronize() blocks the calling CPU thread until all CUDA calls previously issued by the thread are completed. The CUDA event API provides calls that create and destroy events, record events (timestamp), and convert timestamp differences into a floating point value i.e., milliseconds (ms) with a resolution of approximately ½ ms. cudaEventRecord() is used to place the start and stop events into the default stream i.e., stream 0. The device will record a timestamp for the event when it reaches that event in the stream. The cudaEventElapsedTime() function returns the time elapsed between the recording of the start and stop events.

7. Results and Discussion

Usually we require a host function to verify the results from the kernel to check that both versions are indeed producing the same answer. This usually achieved by setting the execution configuration to <<<1, 1>>>, so that the kernel is forced to run with only one block and one thread. This emulates a sequential implementation. In addition, this is very useful for verifying that numeric results are *bitwise* exact from one simulation to another, especially if encountering order of operations issues. Table 3 shows the magnitude of speedup (×) of the CUDA C implementation over that of C++ based on execution time and for a range of block dimensions up to the maximum allowable threads per block i.e., 1,024. Table 4 shows average occupancy, memory bandwidth, and instruction throughput for the range of block dimensions.

Table 3. *Speedup (×) of CUDA C over the C++ implementation.*

	Block Dimensions (x, y, z)				
Grid Size/ Speedup (×)	$(4 \times 4 \times 4)$	$(16 \times 16 \times 4)$	$(16 \times 4 \times 4)$	$(16 \times 8 \times 4)$	$(64 \times 4 \times 4)$
$100 \times 100 \times 100$	19.8	27.9	32.7	31.4	26.9
$200 \times 200 \times 200$	21.1	37.6	40.8	39.5	32.0
$300 \times 300 \times 300$	22.8	40.5	43.5	42.3	39.4
$400 \times 400 \times 140$	23.7	44.1	46.9	45.5	40.3

Table 4. *Average occupancy, bandwidth, and throughput.*

	Block Dimensions (x, y, z)				
	$(4 \times 4 \times 4)$	$(16 \times 16 \times 4)$	$(16 \times 4 \times 4)$	$(16 \times 8 \times 4)$	$(64 \times 4 \times 4)$
# Threads	64	1,024	256	512	1,024
Avg. Occupancy (%)	48.3	90.8	92.4	91.6	84.6
Avg. Bandwidth (GB/s)	113.4	128.6	162.1	157.8	99.2
Avg. Throughput (Gflops)	28.3	32.1	40.5	39.5	24.8

Table 3 & 4 show that the optimal block dimensions are $(16 \times 4 \times 4)$ resulting in the best performance in terms of execution speed, bandwidth and throughput. Notice that average memory bandwidth and throughput are only 56% that of their theoretical peak values which suggests there are likely further areas of optimisation that need to be investigated. However, it is not that surprising since theoretical values are rarely achieved in reality. An optimal choice of execution configuration can often lead to performance benefits at the expense of a higher occupancy. For example, a very low occupancy such as the one obtained with block dimension $(4 \times 4 \times 4)$ is clearly a bad allocation of resources and will generally lead to poor performance. However, the $(64 \times 4 \times 4$ configuration resulted in a high occupancy but the lowest of throughput. So, it is not necessarily the case that the highest occupancy always results in optimum performance. Moreover, it is fair to say that algorithm optimisation is an exhaustive process i.e., involving identify an opportunity for optimisation, apply and testing, verify the speedup achieved, and repeating. It is not necessary for a programmer to spend large amounts of time memorising the bulk of all possible optimisation strategies prior to achieving reasonable speedups. Instead, strategies can be applied incrementally as they are understood. As we have seen, optimisations can be applied at various levels, from overlapping data transfers with computation all the way down to fine-tuning floating-point operations. The available *profiling tools* are invaluable for guiding this process, as they can help suggest a next-best course of action for the developer's optimisation efforts and provide references into the relevant portions of the optimisation section of this guide. When attempting to optimise CUDA C applications, it pays to know how to measure performance accurately and to understand the role that bandwidth plays in performance measurement.

Further areas of improved performance and code optimisation are currently being explored by the authors, including methods such as *data prefetching* and improvements to the *instruction mix*. Data prefetching involved masking the loading of data from global memory to register by overlapping data access and computation. Instruction mix optimisation is where code is refactored to maximise the number of floating-point operations as opposed to addressing and branching. An example would be *loop unrolling*, which decreases loop iterations whilst increasing the number of floating-point calculations per iteration. These optimisations are again based on maximising the available memory bandwidth.

8. Conclusions

The main objective of this paper was to extend previous work by implementing a hybrid continuous-discrete model describing tumour-induced angiogenesis in the more realistic 3D case. We also addressed ways in which performance can be optimised by making use of the GPU hardware architecture and the CUDA programming model. Indeed, CUDA has proved to be a natural programming model to deploy applications in a modern massively-parallel (i.e., highly-threaded) environment. We consider several orders of magnitude performance increase over existing technology a disruptive change that can dramatically alter various aspects of the techniques and methods applied to computational biology. For example, computational tasks that previously would have taken a year can now complete in a few days, hour long computations suddenly become interactive being completed in seconds. Moreover, new technology will allow the tractability of previously intractable complex real-time processing tasks.

Data-parallel processing maps data elements to parallel

processing threads. Many applications that process large data sets can use a data-parallel programming model to speed up computations. For example, 3D rendering, large sets of pixels and vertices are mapped to parallel threads. Similarly, image and media processing applications such as post-processing of rendered images, video encoding and decoding, image scaling, stereo vision, and pattern recognition can map image blocks and pixels to parallel processing threads. In fact, many algorithms can be accelerated by data-parallel processing, from general signal processing or physics simulation to computational biology. In terms of clinical research, developing realistic highly-complex cancer simulations using such technologies will lead to new therapy strategies, optimised drug delivery systems, interactive computer simulations of dynamic oncological process, and other advanced cancer treatment possibilities. Radiation therapies with ion beams can precisely target cancerous tumours, while leaving surrounding healthy tissue unharmed. Such targeted therapy leads to less invasive surgery, shorter hospital stays and speedier recovery times. The drawback is that conventional ion accelerators tend to be huge in both size and cost. This puts them beyond the budget for most medical facilities. Creating live computational simulations and adding physical phenomena to the algorithms was once considered impossible. Now, with the parallel processing power of GPUs, it is no longer impossible. Moreover, the race is on to understand how cell mutation causes cancer, which kills hundreds of thousands worldwide each year and is the second leading cause of death in the U.K. Research is focused on anti-cancer drug discovery pipelines using advanced molecular dynamics simulations powered by GPUs. Enormous gains in computing power are enabling a new framework for drug discovery utilising computer simulations to capture various shapes of a tumour suppressor, the protein called p53, known as the guardian of the genome because it is a key regulator of cell growth and development in normal cells. Indeed, computer simulations capture not just how proteins are built, but how they function inside the body. Such simulations running on GPU-accelerated computers can reveal new binding sites that may help cancer researchers create new drugs to help reactivate p53 when it mutates and ceases to function correctly. Clearly, the continued development of parallel processed computer simulations using GPUs is of paramount importance. Such technologies can clearly aid in the understanding of complex mathematical models and underlying biological processes, whilst also helping to uncover the elusive cure for cancer.

References

[1] Darbyshire, P. M. Coupled Nonlinear Partial Differential Equations Describing Avascular Tumour Growth Are Solved Numerically Using Parallel Programming to Assess Computational Speedup. Computational Biology and Bioinformatics. Vol. 3, No. 5, 65-73. 2015.

[2] Darbyshire, P. M. The Numerical Solution of a Hybrid Continuous-Discrete Model of Tumour-Induced Angiogenesis is Implemented in Parallel and Performance Improvements Analysed. European Journal of Biophysics. Vol. 7, No. 4, 167-182. 2015.

[3] Albini, A., Tosetti, A. F., Li, W. V., Noonan, D. M. and Li, W. W. Cancer prevention by targeting angiogenesis *Nature Reviews Clinical Oncology* 9, 498-509. 2012.

[4] Ferrara, N. and Kerbel, R. S. Angiogenesis as a therapeutic target. *Nature*, 438 967–974. 2005.

[5] Carmeliet, P. Angiogenesis in life, disease and medicine. *Nature*, 438: 932–936. 2005.

[6] Bouard S. de, Herlin, P. and Christensen, J. G. Antiangiogenic and anti-invasive effects of sunitinib on experimental human glioblastoma. *Neuro-Oncology*, Vol. 9, No. 4, 412–423. 2007.

[7] Norden, A. D, Drappatz, J. and Wen P. Y. Novel antiangiogenic therapies for malignant gliomas. *The Lancet Neurology*, Vol. 7, No. 12, 1152–1160. 2008.

[8] Peirce, S. M. Computational and mathematical modeling of angiogenesis. *Microcirculation*, 15(8), 739–751. 2008.

[9] M. Scianna, M., Bell. C. and Preziosi L. A review of mathematical models for the formation of vascular networks. *Oxford Centre for Collaborative Applied Mathematics*. 2012.

[10] Anderson, A.R.A. and Chaplain, M. Continuous and discrete mathematical models of tumour-induced angiogenesis, *Bulletin of Mathematical Biology*, 60, 857-900. 1998.

[11] Anderson, A., B. D. S. Sleeman, I. M. Young and B. S. Griffiths. Nematode movement along a chemical gradient in a structurally heterogeneous environment: II. Theory. *Fundamental and Applied Nematology*, 20, 165–172. 1997.

[12] NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. Whitepaper. NVIDIA Corporation. 2012.

[13] Nvidia Corporation. *CUDA C programming guide*. Version 6.0. 2014.

[14] CUDA C BEST PRACTICES GUIDE. NVIDIA Corporation. 2015.

[15] Cheng, J., Grossman, M and McKercher, Ty. Professional CUDA C Programming. Wrox. 2014.

[16] Venkatasubramanian, S. and Vuduc, R. W. Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems. In *Proceedings of the Association of Computing Machinery International Conference on Supercomputing*, New York. 2009.

[17] Amorim, R., Haase, G., Liebmann, M. and Weber dos Santos, R. Comparing CUDA and OpenGL implementations for a Jacobi iteration. In *Proceedings of High Performance Computing and Simulation Conference*, Berlin. 2009.

[18] Cecilia, J. M., Garcıa, J. M. and Ujaldon, M. CUDA 3D stencil computations for the Jacobi method. Springer, 173-183. 2012.