A Hybrid Continuous-Discrete Model of Tumour-Induced Angiogenesis is Solved Numerically in Parallel and Performance Improvements Analysed

Paul M. Darbyshire

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

Email address:

rd@algenet.com

To cite this article:

Paul M. Darbyshire. A Hybrid Continuous-Discrete Model of Tumour-Induced Angiogenesis is Solved Numerically in Parallel and Performance Improvements Analysed. *European Journal of Biophysics*. Vol. 3, No. 5, 2015, pp. 27-37. doi: 10.11648/j.ejb.20150305.11

Abstract: The main aim of this paper is to investigate the potential performance improvements gained from a serial versus parallel implementation of the numerical solution to a system of coupled nonlinear PDEs describing tumour-induced angiogenesis. After applying a suitable finite difference scheme, the resulting hybrid continuous discrete model is solved based on a set of cellular rules defining endothelial cell movement towards a tumour. In addition, the model explicitly incorporates the processes of branching, anastomosis and cell proliferation. Parallel implementations are based on the CUDA programming model with a detailed look at efficient thread deployment and memory management. Results show substantial speedups for the CUDA C language against that of conventional high performance languages, such as C++. Such increased performance highlights the potential for simulating more complex mathematical models of tumour dynamics, such as vascularisation networks, tumour invasion and metastasis, leading to the potential for more rapid experimental results for a range of complex cancer models.

Keywords: Cancer Modelling, Tumour Angiogenic Factors (TAF), Tumour-Induced Angiogenesis, Anastomoses, Parallel Programming, Compute Unified Device Architecture (CUDA), Graphical Processing Unit (GPU), High Performance Computing (HPC)

1. Introduction

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*. The morphological events that are involved in angiogenesis have been highlighted through studies involving *in vivo* experiments, such as the chick chorioallantoic membrane and animal corneal models, and *in vitro* examination of endothelial cell migration and proliferation [1-5]. Whilst 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. Interestingly, one of the crucial milestones in solid tumour development is the so-called *angiogenic switch* i.e. the ability of a tumour to trigger the formation of its own vascular network by the secretion of chemical factors [6]. From these vascular networks, blood vessels provide essential nutrients and

oxygen throughout the body being comprised of an inner lining of closely assembled *endothelial cells* embedded stromal cells and the *extracellular matrix* (ECM). In healthy tissues, a series of biologic *on* and *off* switches regulates the process of angiogenesis. The main *on* switches are known as angiogenesis-stimulating growth factors and *off* switches as angiogenesis inhibitors. To monitor and supply sufficient amounts of essential nutrients to the surrounding tissues, blood vessels have hypoxia-induced sensors, or receptors that assist in vessel remodelling to adjust the blood flow accordingly. 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 [7-11].

Angiogenesis can be considered a complex biological phenomena and one that at a system 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 [12, 13]. 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 is a powerful method of testing hypotheses, confirming biological experiments, and simulating complex dynamics.

In the last decade *in silico* experiments focused on tumour growth have become more readily accepted by the biological community both as a means to direct new research and a route to integrate multiple experimental measurements in order to generate new hypotheses and testable predictions. More recently, the advantages of using parallel processing has highlighted the potential speedup gained from the numerical solution of complex mathematical models using high performance computing [14]. This recent shift has been partly driven by the emergence of new theoretical approaches, such as *hybrid* modelling [15]. Hybrid models integrate both continuous and discrete processes of biological phenomena on various temporal and spatial scales. These models represent cells as individual discrete entities and often use continuous nutrient concentrations to model cellular behaviour due their microenvironment. Indeed, the mixture of chemical factors in the microenvironment is the most important aspect regulating the behaviour of cells during angiogenesis. So, the cell centric nature of hybrid models naturally connects with cell biology and readily incorporates intra and extracellular phenomenon. The model presented in this paper is of a hybrid type in which a system of couple nonlinear partial differential equations (PDEs) describe the continuous chemical and macromolecular dynamics and a discrete cellular automata-like model controls cell migration and interaction of neighbouring endothelial cells.

2. Biological Description

Solid tumours generally undergo a period of avascular growth, after which they become dormant for a sustained period without access to a sufficient supply of essential nutrients, such as oxygen and glucose. Beyond a certain size

(~2 mm) diffusion alone is insufficient for the provision of such nutrients; the surface area to volume ratio is too low and as such the developing tumour begins to starve. In response to this state of hypoxia, cancer cells send out signals to cells of nearby blood vessels by secreting a number of chemicals, known collectively as *tumour angiogenic factors* (TAF) [16-18]. Such angiogenic factors are likened to the biologically relevant *vascular endothelial growth factor* (VEGF) that acts as an effective chemokine for endothelial cells. Once secreted, TAF diffuse into the surrounding tissue and sets up an initial steady-state concentration gradient between the tumour and any pre-existing vasculature. Endothelial cells situated in nearby parent vessels degrade their own basal lamina and begin migrating into the ECM [19, 20]. The degradation of the basal lamina leads to damage, and possible rupture, of the parent vessel basement membrane. Such damage allows fibronectin from the blood to leak from the parent vessel and diffuse into the surrounding tissue [21, 22]. Small capillary sprouts form from several endothelial cell clusters and begin to extend towards the tumour, directed by the motion of the leading endothelial cell at the sprout tip, until the finger-like capillaries reach a certain length. At this point, they tend towards each other, and form loops before fusing together in a process known as *anastomoses* [16, 17]. Following anastomoses, the primary loops start to bud and sprout repeating the process above and further extending the newly formed capillary bed (see Figure 1). Further sprout extension occurs when some of the endothelial cells on the sprout-wall begin to proliferate. Cell division is largely confined to a region just behind the cluster of mitotically inactive endothelial cells that constitute the sprout-tip. This process of sprout-tip migration and proliferation of sprout wall cells forms solid strands of endothelial cells within the ECM. As the sprouts approach the tumour, branching rapidly increases and produces a *brush border* effect [2, 4], until the tumour is finally penetrated [23]. Once a supply of essential nutrients reaches the tumour, through this newly formed blood circulatory system, it enters the phase of vascularisation. To support continued growth, the vascular system constantly restructures itself implying that angiogenesis is an on-going process, continuing indefinitely until the tumour is removed or destroyed. Indeed, angiogenesis also enables the tumour to spread to other parts of the body through the blood stream significantly increasing the probability of mortality from cancer due to metastasis.

Figure 1. The process of angiogenesis. A. Early capillary growth B. Development of a complex network of capillaries.

3. A Hybrid Model of Angiogenesis

Progress in mathematical modelling of angiogenesis has largely been driven by biological and clinical observations through *in vitro* and *in vivo* experiments [1-5]. The majority of mathematical models focus on the development of a set of spatial-temporal reaction-diffusion equations that describe essential nutrient concentrations. In general, such a method results in a system of coupled nonlinear partial differential equations (PDEs) that require a numerical solution subject to a number of observable (where possible) parameters. In this paper, we present a system of PDEs first developed by Anderson *et al* [24] that describe tumour induced angiogenesis. Moreover, the continuum approach, described by a set of coupled nonlinear PDEs, models the initial chemotactic response of endothelial cells to TAF and the hypotactic response of endothelial cells to fibronectin. Then, a discretised form of the PDEs leads to a biased random walk model which allows the individual paths of endothelial cells located at the sprout tips to be effectively traced.

3.1. The Continuous Model

Based on the above discussion, we assume that the motion of an endothelial cell (at or near a capillary sprout tip) is influenced by three factors, namely:

1. Random motility,
2. Chemotaxis in response to TAF gradients in the surrounding connective tissue stroma [14, 15], and
3. Haptotaxis in response to fibronectin gradients, also present in the surrounding tissue [16-18].

If we denote the endothelial cell density by n , the TAF concentration by c and the fibronectin concentration by f , then the complete system of scaled coupled nonlinear PDEs describing 1-3 can be written as [24]:

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

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

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

The chemotactic migration is characterised by the function $\chi(c)$ given by:

$$\chi(c) = \frac{\chi}{1 + \alpha c} \quad (4)$$

Table 1. Parameter descriptions and values used in the coupled nonlinear PDE model (1) – (4) [24].

Parameter	Description	Value
α	Decay factor	0.6
β	Fibronectin production coefficient	0.005
γ	Fibronectin degradation	0.1
D	Random motility diffusion coefficient	0.00035
η	Rates of TAF uptake	0.1
ρ	Haptotactic coefficient	0.34
χ	Chemotactic coefficient	0.38

$\chi(c)$ reflects the fact that chemotactic sensitivity generally decreases with increased TAF concentration. A description of each of the parameters, and their respective values, are shown in Table 1.

Our system is assumed to hold on a spatial (x, y) domain Ω (i.e. a region of tissue) with appropriate initial conditions; $c(x, y, 0)$, $f(x, y, 0)$ and $n(x, y, 0)$ as described below, and that tumour cells are confined within a domain $\Omega \in [0, 1] \times [0, 1]$ in which no-flux (Neumann) boundary conditions are imposed on $\partial\Omega$, the boundary of Ω . The no-flux boundary condition is given by:

$$(-D \nabla n + n(\chi(c) \nabla c + \rho \nabla f)) = 0 \quad (5)$$

For the initial conditions, we approximate a row of tumour cells by considering an initial TAF concentration, $c(x, y, 0)$ (maximum at $x = 1$) given by:

$$c(x, y, 0) = e^{-\frac{(1-x)^2}{\epsilon_1}}, \Omega \in [0, 1] \times [0, 1] \quad (6)$$

We take the initial concentration of fibronectin, $f(x, y, 0)$ (maximum at $x = 0$) to have the form:

$$f(x, y, 0) = k e^{-\frac{x^2}{\epsilon_2}}, \Omega \in [0, 1] \times [0, 1] \quad (7)$$

After the TAF has reached the parent vessel, the endothelial cells within the vessel develop into several cell clusters which eventually form sprouts. For simplicity, we assume that initially three clusters develop along the y -axis at $x \approx 0$, with the tumour located at $x = 1$ and the parent vessel of the endothelial cells at $x = 0$. In this case, the initial density of endothelial cells, $n(x, y, 0)$ is given by the following:

$$n(x, y, 0) = e^{-\frac{x^2}{\epsilon_3}} \sin^2(6\pi y), \Omega \in [0, 1] \times [0, 1] \quad (8)$$

Where $k, \epsilon_1, \epsilon_2$ and ϵ_3 are all positive constants.

3.2. The Discrete Model

In order to capture the complex morphological features of the developing capillary network, such as individual capillary sprouts, branching and anastomosis, the continuous model must be developed further. Cellular automata models are particularly useful for providing a foundation upon which we can develop a more detailed and precise biological model. The spatial movement of individual agents in cellular automata models are primarily governed by nearest-neighbour interactions and as such share some similarity with the discrete model we will present below. However, in general, the nearest-neighbour interactions for cellular automata models are based on phenomenological rules, whereas, in the discrete model presented here, the movement rules are based directly on a discretised form of the continuous model described above. The technique of tracing the path of an individual endothelial cell at a sprout tip was first proposed by Anderson *et al.* [25]. The method involves using standard *finite difference methods* (FDM) to discretise

the continuous model described in (1)-(4). Then, the resulting coefficients of the finite difference five-point stencil are used to generate the probabilities of movement of an individual EC in response to its local microenvironment. Stencil computations are those in which each node in a multi-dimensional grid is updated with a weighted average of neighbouring node values. These neighbours comprise the stencil, over which a large number of iterations across the array width generally leads to a successful numerical convergence. A schematic diagram of a finite difference five-point stencil scheme is shown in Figure 2.

Figure 2. A schematic diagram of a finite difference 5-point stencil scheme.

We first discretise the continuous model by approximating the two-dimensional domain $\Omega \in [0, 1] \times [0, 1]$ as a grid of nodes of size h , and time t by increments of size k . By applying a *forward* finite difference, the *fully-explicit* discretised version of the continuous model (1)-(4) can be written as:

$$n_{i,j}^{q+1} = n_{i,j}^q P_0 + n_{i+1,j}^q P_1 + n_{i-1,j}^q P_2 + n_{i,j+1}^q P_3 + n_{i,j-1}^q P_4 \quad (9)$$

$$f_{i,j}^{q+1} = f_{i,j}^q [1 - k\gamma n_{i,j}^q] + k\beta n_{i,j}^q \quad (10)$$

$$c_{i,j}^{q+1} = c_{i,j}^q [1 - k\eta n_{i,j}^q] \quad (11)$$

The coefficients P_0 - P_4 can be thought of as being proportional to the probabilities of endothelial cell movement i.e. the coefficient P_0 , is proportional to the probability of no movement, and the coefficients P_1 , P_2 , P_3 and P_4 , are proportional to the probabilities of moving left, right, up and down, respectively. These are written as:

$$P_0 = 1 - \frac{4kD}{h^2} + \frac{k\alpha\chi(c_{i,j}^q)}{4h^2(1+\alpha c_{i,j}^q)} \left[(c_{i+1,j}^q - c_{i-1,j}^q)^2 + (c_{i,j+1}^q - c_{i,j-1}^q)^2 \right] - \frac{k\chi(c_{i,j}^q)}{h^2} (c_{i+1,j}^q + c_{i-1,j}^q - 4c_{i,j}^q + c_{i,j+1}^q + c_{i,j-1}^q) + \frac{k\rho}{h^2} (f_{i+1,j}^q + f_{i-1,j}^q - 4f_{i,j}^q + f_{i,j+1}^q + f_{i,j-1}^q) \quad (12)$$

$$P_1 = \frac{kD}{h^2} - \frac{k}{4h^2} \left[\chi(c_{i,j}^q)(c_{i+1,j}^q - c_{i-1,j}^q) + \rho(f_{i+1,j}^q - f_{i-1,j}^q) \right] \quad (13)$$

$$P_2 = \frac{kD}{h^2} + \frac{k}{4h^2} \left[\chi(c_{i,j}^q)(c_{i+1,j}^q - c_{i-1,j}^q) + \rho(f_{i+1,j}^q - f_{i-1,j}^q) \right] \quad (14)$$

$$P_2 = \frac{kD}{h^2} - \frac{k}{4h^2} \left[\chi(c_{i,j}^q)(c_{i,j+1}^q - c_{i,j-1}^q) + \rho(f_{i,j+1}^q - f_{i,j-1}^q) \right] \quad (15)$$

$$P_3 = \frac{kD}{h^2} + \frac{k}{4h^2} \left[\chi(c_{i,j}^q)(c_{i,j+1}^q - c_{i,j-1}^q) + \rho(f_{i,j+1}^q - f_{i,j-1}^q) \right] \quad (16)$$

With grid spacing, $h = \Delta x = \Delta y$ and time step, $k = \Delta t$. (1) – (16) represents our *hybrid continuous-discrete* (HCD) model of tumour-induced angiogenesis. The exact forms of P_0 - P_4 are functions of both fibronectin and TAF concentrations at nearby neighbouring points of an individual endothelial cell. Therefore, the HCD model clearly describes the interactions between cells and their microenvironment. In addition, we explicitly incorporate the processes of branching, anastomosis and cell proliferation into the discrete model.

Figure 3. A schematic representation of branching at a sprout tip and looping of two capillary sprouts.

From Figure 2, we can see that the discrete model allows us to determine the endothelial cell density at grid node (i, j) and time $q + 1$, by averaging the cell density of the four neighbouring nodes at the previous time step q . We use the five coefficients P_0 - P_4 to generate the motion of an individual endothelial cell and assume that the motion of an individual endothelial cell located at the tip of a capillary sprout governs the motion of the whole sprout. This is not considered unreasonable since the remaining endothelial cells lining the sprout-wall are contiguous [4]. We further assume that each sprout tip has a probability, P_b of generating a new sprout (branching) and that this probability is dependent on the local TAF concentration. It is also reasonable to assume that the newly formed sprouts do not branch until there is a sufficient number of endothelial cells near their tip. We will assume that the density of endothelial cells required for branching is inversely proportional to the concentration of TAF, since new sprouts become much shorter as the tumour is approached [4]. Based on these assumption we can write down the following three cellular rules:

Rule 1: New sprouts must mature for a length of time ($\psi = 0.5$) [24] before branching,

Rule 2: Sufficient local space exists for a new sprout to form, and

Rule 3: Endothelial cell density, $n > n_b$, where $n_b \propto \frac{1}{c_{i,j}}$.

Figure 3 shows a schematic of the branching at a sprout tip and looping of two capillary sprouts.

3.3. Simulating the HCD Model

Each numerical simulation is based on an increasing size of array width resulting in finer grained uniform grids. This implies a discretisation of the HCD model on the unit square $\Omega \in [0,1] \times [0,1]$ with subsequent space steps h determined by the size of the array width. We use a constant iteration size of 5,000 time steps to allow for successful convergence of the numerical solution. We assume that there are initially five capillary sprout tips determined by endothelial cell initial conditions, of which three of them are discrete peaks located at the maximum-density of (8), the other two being placed between these three. Discretised versions of (6) and (7) are also used to provide initial conditions for the fibronectin and TAF concentrations, respectively. The no-flux boundary conditions are also incorporated into the model using a discretised version of (5) such that endothelial cells are contained within the uniform grid Ω . At each time step the numerical simulation involves solving the discrete model to generate the five coefficients P_0-P_4 , which are subsequently *normalised*. Using the values of these coefficients, a set of five probability ranges are determined based on the following:

$$R_0 = 0 \text{ to } P_0 \quad (17)$$

$$R_m = \sum_{n=0}^{m-1} P_n \text{ to } \sum_{n=0}^m P_n \quad (18)$$

Where $m = 1 \dots 4$. 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), or down (R_4). Each endothelial cell is therefore restricted to move to one of its four orthogonal neighbouring grid nodes or remain stationary at each time step. Parameters values as shown in Table 2 were used for all of the numerical simulations.

4. Implementation

Table 2. Parameter values used in the numerical simulations [24].

Parameter	Value
α	0.6
β	0.005
γ	0.1
D	0.00035
η	0.1
ρ	0.34
χ	0.38
k	0.75
ε_1	0.45
ε_2	0.45
ε_3	0.001

The main aim of this paper is to investigate the potential performance gained from a serial versus parallel implementation of a finite difference scheme to the numerical solution of the HCD model described above.

4.1. Hardware

The hardware used for the sequential C++ language implementation was a 4th generation Intel® Core™ i7-4790K CPU (4 core) processor running on Windows 8.1. 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 6.0 and tested on an Nvidia GeForce® GTX™ 780 GPU (2,304 core) card with compute capability 3.5.

4.2. Performance Benchmarks

The test platform uses the C++ programming language for the serial implementation and CUDA C programming model for the parallel version of the numerical solution to the HCD model. C++ is an obvious choice for professional development and one that is heavily adopted throughout the research community. C++ conveniently provides several powerful methods for allocating and managing memory making it extremely versatile when considering computational efficiency. C++ is also the natural choice for any CUDA enabled development since it relies heavily on extensions from the C language as a basis for its own language.

The actual performance improvement is 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 HCD model.

4.3. The CUDA Programming Model

A graphics processing unit (GPU) trades off fast single thread performance and clock speed for high throughput and streaming capabilities. The GPU consists of an array of highly threaded *multiprocessors* (MP); each having their own individual *streaming processors* (SP) that share control logic and instruction cache. This keeps the available area of each SM relatively small, and therefore more SMs can be packed per die, as compared to the limited number of available CPU cores. Over the past several years, the new *compute unified device architecture* (CUDA) developed by Nvidia has completely revolutionised numerical computation on the GPU. CUDA-driven applications run the sequential part of their workload on the CPU (the *host*), which is optimised for single-threaded performance, while accelerating parallel processing on the GPU (the *device*). A GPU is connected to the host through a high speed bus slot, typically PCI-express (PCI-e) in current high performance set ups. The GPU has its

own device memory, and is transferred between the GPU and host memories using programmed *direct memory access* (DMA), which operates concurrently between both the host and GPU. The device memory supports a very high memory bandwidth through the use of a wide data path e.g., a GPU with a 384-bit memory interface width allows twelve consecutive 32-bit words to be accessed from memory in a single cycle. However, since the GPU is a *coprocessor*, usually on a separate PCI-e card, data must first be copied explicitly from the host to device memory on the GPU. This can give rise to performance bottlenecks which are generally minimised by making intelligent use of memory management. Despite these drawbacks, GPUs are an extremely viable candidate for performing highly intensive computations that exhibit high levels of data parallelism.

4.3.1. Threads, Blocks, and Grids

The CUDA C 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, *asynchronously*. The SIMD architecture is an efficient mode of execution in which the same operation is

carried out on each data element at the same time i.e., *data parallel* computation. 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, such 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. Launching a CUDA kernel creates a grid of threads that all execute the kernel function. In its simplest form, the kernel is defined using the following CUDA C syntax [26, 27]:

```
global __kernel<<<GRIDSIZE,BLOCKSIZE>>> (...);
```

Threads are grouped into *blocks* and blocks are grouped into *grids* as shown schematically in Figure 4. There is a limit to the number of threads per block, on current GPUs a thread block may contain up to 1,024 threads. On the GPU, each MP is responsible for handling one or more blocks in a grid which is further divided into a number of SPs each handling one or more threads in a block. Threads are scheduled in groups of 32 parallel threads called *warps* and subsequently executed at the warp level.

Figure 4. A schematic representation of threads, blocks and grids.

Figure 5. A schematic diagram of block and thread indexing in CUDA.

Since all threads in a grid execute the same kernel function, they rely on unique block and thread IDs to distinguish them from each other and to identify the appropriate portion of data to process. For example, to use a 2D set of threads, we make use of the built-in variables `blockIdx.x` and `blockIdx.y` to give the block ID, `threadIdx.x` and `threadIdx.y` for the thread ID, and `blockDim.x` and `blockDim.y` for the block dimensions i.e. number of threads in a block. The following CUDA C syntax defines a 2D set of thread indices (*i, j*) [25, 26]:

```
int i = (blockIdx.x*blockDim.x)+threadIdx.x;
```

```
int j = (blockIdx.y*blockDim.y)+threadIdx.y;
```

Figure 5 shows a schematic diagram of block and thread indexing used in the CUDA programming model.

Threads are organised in a two-level hierarchy. At the top, a grid is organised into a 2D array of blocks. The number of blocks in each dimension is specified by the first parameter given in the kernel launch: `GRIDSIZE`. At the bottom level, all blocks of a grid are organised into a 3D array of threads. The number of threads in each dimension of a block is specified by the second parameter given in the kernel launch: `BLOCKSIZE`. Each parameter is a `dim3` CUDA C data type, which is essentially a struct with three fields' `.x`, `.y`, and `.z` all initialised to 1. Since grids are only a 2D array of block dimensions, the third field is often ignored; but still initialised to one. For example, a kernel could be launched with the following syntax [26, 27]:

```
dim3 GRIDSIZE(3, 2);
```

```
dim3 BLOCKSIZE(4, 3);
```

```
__global__ kernel<<<GRIDSIZE, BLOCKSIZE>>>(...);
```

Figure 6 shows a schematic diagram of the thread, block and grid architecture for the kernel launch based on the above CUDA C syntax.

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. Note that the CUDA programming model does not issue warnings or errors if thread bookkeeping is not optimised [26, 27].

4.3.2. Global and Shared Memory

The CUDA programming model makes available a number of accessible memories. One such memory is *global* (or *device*) memory which resides in the *dynamic random-access memory* (DRAM). Global memory is both read and write access and normally the largest of all available memory. However, access to global memory is relatively slow and the

need to continually transfer data between the host and device can lead to severe bottlenecks. A more useful type of available memory and with much faster access is *shared memory*. With shared memory, stored variables can be freely modified; the CUDA API creates copies of variables stored in shared memory for each block it launches and thereby allows every thread in that block shared access to those variables. Shared memory buffers also reside physically on the GPU thereby greatly improving the latency of access and per-block programmable management cache. Figure 7 shows a schematic diagram of the relationship between global and shared memory.

Figure 6. A schematic diagram of an example thread, block and grid architecture.

Figure 7. A schematic diagram of the relationship between global and shared memory.

When sharing data between threads using shared memory, it is necessary to be careful to avoid race conditions that can lead to undefined behaviour and incorrect results. Although threads in a block run logically in parallel, not all threads can execute physically at the same time. The CUDA C

`__syncthreads()` function allows us to synchronise and coordinate memory access by providing a barrier to execution [26, 27]. When a kernel function calls `__syncthreads()`, all threads in a block are held at the calling location until all other threads in the block reach the same location. This ensures that all threads in a block have executed that part of the kernel before they all move on to the next operation. Figure 8 shows a schematic diagram representing thread barrier synchronisation.

Figure 8. A schematic diagram of thread barrier synchronization.

4.3.3. The C++ and CUDA C Implementations

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:MAX_ITER do
    Determine coefficients:  $P_0, P_1, P_2, P_3, P_4$ 
    Determine probability ranges:  $R_0, R_1, R_2, R_3, R_4$ 
    Generate uniform random number [0, 1]
    Store results
    for j = 1:Nx-1 do
        for j = 1:Ny-1 do
            Update nodes:  $n_{i,j}^{q+1}, f_{i,j}^{q+1}, c_{i,j}^{q+1}$ 
        end for
    end for
    Swap pointers
end for
End clock()
Print results
Free() array memory

```

The CUDA programming model generally requires that

arrays use a single *contiguous* block of linear memory. So, rather than declaring a 2D array in C++, we use a single linear block of memory and reference it as if it were a 2D array. Such memory allocation uses *row-major order* which allows access to the individual array elements by determining the distance of each element from the start of the array. In the C++ programming language, the `calloc()` function allocates a block of memory, but unlike `malloc()`, all the allocated memory is cleared (set to 0). This is an ideal situation for initialising arrays for iterative procedures such as the one implemented here. Also, using `calloc()` can save some additional overhead of unnecessary calls to `memset()` which we would have to do if we were using `malloc()`. Thus, `calloc()` allocates space for an array of elements, initialises all of them to zero and returns a pointer to memory. `calloc()` also returns a null pointer if it cannot allocate a block of memory of adequate size. All memory is subsequently freed using `free()` at the end of code execution. The main body of the C++ implementation is shown in Algorithm 1.

Achieving a high-level of performance and optimisation using the CUDA programming model requires careful attention to detail [14, 28-30]. 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 `cuda Malloc()` and data is transferred between host and device memory using `cudaMemcpy()` depending on the data flow i.e. either `cudaMemcpy Host To Device` or `cudaMemcpy Device To Host`. Memory is subsequently freed after completion using `cuda Free()`. Making efficient use of available memory (e.g. shared memory) can reduce the amount of data that has to be physically transferred between host and device, which is typically the performance bottleneck. We implemented the CUDA C kernel with and without shared memory to assess the improvements in execution speed and performance. The algorithms for the implementation of the CUDA C kernel are shown in Algorithms 2 and 3. Algorithm 4 shows the main body of the CUDA C implementation.

Algorithm 2 CUDA C kernel (without shared memory)

```

Get current (i, j) thread index
if (i < Nx-1 && j < Ny-1 && i > 0 && j > 0) then
    Update nodes:  $n_{i,j}^{q+1}, f_{i,j}^{q+1}, c_{i,j}^{q+1}$ 
end if

```

Algorithm 3 CUDA C kernel (with shared memory)

```

Declare shared memory
Allocate shared memory
__synchronise()
Get current shared (si, sj) thread index
    if (si < BLOCKSIZE-1 && sj < BLOCKSIZE-1 && si > 0 && sj > 0) then
        Update nodes:  $n_{i,j}^{q+1}, f_{i,j}^{q+1}, c_{i,j}^{q+1}$ 
    end if

```

Unfortunately there is no global kernel barrier routine available in the CUDA programming model. For this reason, we have included a workaround using the cuda Thread Synchronisation() method within the kernel looping routine which acts as a barrier for synchronisation of the kernel calls.

Algorithm 4 CUDA C main body

```

Define model parameters
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 GRIDSIZE and BLOCKSIZE
Start clock()
    for q = 1:MAX_ITER do
        Determine coefficients:  $P_0, P_1, P_2, P_3, P_4$ 
        Determine probability ranges:  $R_0, R_1, R_2, R_3, R_4$ 
        Generate uniform random number [0, 1]
        Store results
        Launch kernel <<< GRIDSIZE, BLOCKSIZE, >>>
        cudaThreadSynchronise()
        Swap pointers
    end for
End clock()
Copy device arrays to host
Print results
Free() host array memory
cudaFree() device array memory

```

5. Results and Discussion

Table 3 shows the speedups ($\times$) achieved for the CUDA C, with and without shared memory, over that of the C++ implementation for an increasing width of array and subsequent number of grid nodes.

Tables 3 shows clearly that the CUDA programming model can be successfully applied to numerical solutions of the HCD model in order to dramatically increase overall performance based on execution time. Indeed, by just using the basic CUDA programming model without any memory optimisation we achieve impressive speedups of between $40\times$ and $74\times$ over conventional C++ for array widths of 1,000 and 2,000, respectively. By using shared memory, we can clearly see that once the full advantages of parallel processing are being employed (millions of computations), speedups in the range of $145\times$ are possible. That is, utilising shared memory gives us an improvement in performance of

around $2\times$ that of relying purely on global host to device data transfers. Figure 9 shows the comparison of the CUDA C implementations both with and without shared memory.

Table 3. Speedup ($\times$) of the CUDA C (with and without shared memory) over the C++ implementation.

Array Width	Grid Nodes	CUDA C (Without)	CUDA C (With)
100	10,000	1.27	1.22
500	250,000	14.83	21.78
1,000	1,000,000	40.41	74.88
2,000	4,000,000	73.85	145.02

Figure 9. Comparison of the speedups achieved using CUDA C with and without shared memory.

With more clever use of memory management there is potential for further performance improvement and code optimisation. Much of which relies more on a detailed understanding of the GPU hardware architecture rather than just the CUDA programming model. For example, consider warp size and the apparent number of threads running concurrently on an MP. In actuality, threads are running in both parallel and serial on the GPU. Within a warp, threads all have sequential indices such that there is a warp with indices 0 ... 31, then 32 ... 63 and so on up to the total number of threads in a block. The homogeneity of threads in a warp has a huge effect on performance. If all threads are executing the same instruction, then all the SPs in an MP can execute the same instruction in parallel. But if one or more threads in a warp is executing a different instruction from the others, then the warp has to be partitioned into groups of threads based on the instructions being executed, after which the groups are executed one after the other. This *serialisation* reduces throughput as threads become more and more *divergent* and *split* into smaller and smaller groups. It is therefore optimal to keep threads as homogenous as possible. Moreover, how the threads access global memory also affects throughput i.e., performance can be significantly improved if the GPU can *coalesce* several global addresses into a single burst access over the wide data bus that goes to external memory. Conversely, reading and writing separated memory addresses requires multiple accesses which can also slow down computations and hinder performance.

Shared memory, like all memories must be read into

registers before being operated on. Threads of a warp make requests from shared memory simultaneously using some form of access pattern of 32 addresses (since accesses are issued in warps of 32 threads). All threads can request a different address so that all 32 threads will get their data concurrently, if possible. To achieve high memory bandwidth for concurrent accesses, shared memory is divided into equally sized memory modules known as banks that can subsequently be accessed simultaneously. Any memory load or store of n addresses that spans n distinct memory banks can be accessed simultaneously, providing an effective bandwidth that is n times as high as the bandwidth of a single bank. Moreover, a bank conflict can occur when any memory access pattern fails to distribute efficiently across banks available in memory, during which access is achieved serially, losing all the advantages of a parallel architecture. Often bank conflicts can be avoided by using a shared memory buffer through clever use of memory padding and data manipulation.

The authors are currently investigating many of the performance issues described above to further improve numerical simulations based on the CUDA programming model. Indeed, such understanding will greatly help in tackling more complex numerical simulations allowing us to fully exploit the advantages of using parallel processing on high performance computers for cancer modelling.

On a more biological note, it is worth pointing out that work presented in this paper has dealt with individual endothelial cells migrating from a single parent vessel. In reality, a tumour can be penetrated, and subsequently fed, from a number of different vascular networks emanating from a variety of different parent vessels (see Figure 10). Such an array of vascular networks implies that each vasculature may respond differently to anti-angiogenesis treatments, for example some may be totally inefficient for drug delivery whilst others may be optimal. This would result in a highly heterogeneous distribution of drug to the tumour, with some regions being totally destroyed by strong concentrations of cytotoxic drugs and others becoming hypoxic and potentially releasing further angiogenic factors. Indeed, computational methods developed here will allow us to rapidly quantify and assess the ability of different patterns of vasculature networks to transport blood, nutrients, and chemotherapeutic drugs into the tumour.

Recently, mathematical and computational models have been developed for the purpose of identifying new therapeutic targets and clinically relevant approaches for either inhibiting or stimulating angiogenesis. Such endeavours include the identification of putative drug candidates or drug combination strategies and the engineering of vascularised tissue structures *ex vivo*. Future work by the authors will involve incorporating flow dynamics through complex 3D networks, taking into account factors, such as blood pressure, shear stress and viscosity in order to understand in more detail the biological implications for tumour invasion and growth and also identify potentially viable chemotherapeutic treatment strategies.

Figure 10. A schematic diagram representing a more complex tumour-induced angiogenesis scenario in which several vascular networks penetrate the tumour from a number of different surrounding parent vessels.

6. Conclusions

In this paper we have presented a comparison between implementations of a serial C++ and parallel CUDA C approach to solving a hybrid continuous-discrete model of tumour-induced angiogenesis. The main aim was to show that a numerical solution to a system of coupled nonlinear PDEs, such as those frequently encountered in computational biophysics, could benefit from being treated in a parallel manner. It was shown that the CUDA programming model which dramatically improved performance quantified by the execution time of a 2D finite difference implementation, resulting in speedups in the range of $145\times$ over that of their serial counterpart when employing data access through shared memory. Clearly, when challenged with more complex biological phenomenon, such as hemodynamics and chemotherapy drug targeted systems, the use of parallel processing is clearly an invaluable tool. The authors have already begun implementing more complex 3D numerical solutions of vascular tumour growth, which include the processes of metastasis and tissue invasion, with promising early results. Indeed, it is assumed that the implementation of more advanced numerical schemes that can be handled in a massively parallel manner will provide an extremely productive computational platform. With such tools, clinicians and oncologists will have the ability to develop new treatment scenarios and simulate these in a realistic environment a gain rapid results, which will substantially reduce pre-clinical trial costs and associated time constraints.

References

- [1] Arnaoutova, I. and Kleinman, H. K. In vitro angiogenesis: Endothelial cell tube formation on gelled basement membrane extract. *Nature Protocols*, 5, 628–635. 2010.

- [2] Gimbrone MA, Cotran RS, Leapman SB, Folkman J: Tumor growth and neovascularization: An experimental model using the rabbit cornea. *Journal of the National Cancer Institute*. 52, 413–427. 1974.
- [3] Folkman, J. and Haudenschild, C. Angiogenesis in vitro. *Nature*, 288, 551–556. 1980.
- [4] Muthukkaruppan, V. R., Kubai, L., Auerbach, R. Tumorinduced neovascularization in the mouse eye. *Journal of the National Cancer Institute*, 69, 699–705. 1982.
- [5] Jain RK, Schlenger K, Höckel M. and Yuan F: Quantitative angiogenesis assays: Progress and problems. *Nature Medicine* 3: 1203–1208. 1997.
- [6] Hanahan, D and Folkman, J. Patterns and emerging mechanisms of the angiogenic switch during tumorigenesis. *Cell*, 86, 353–364. 1996.
- [7] 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.
- [8] Ferrara, N. and Kerbel, R. S. Angiogenesis as a therapeutic target. *Nature*, 438 967–974. 2005.
- [9] Carmeliet, P. Angiogenesis in life, disease and medicine. *Nature*, 438: 932–936. 2005.
- [10] 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.
- [11] 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.
- [12] Peirce, S. M. Computational and mathematical modeling of angiogenesis. *Microcirculation*, 15(8), 739–751. 2008.
- [13] 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.
- [14] 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.
- [15] Rejniak A. K. and Anderson A.R.A. Hybrid models of tumor growth. *Wiley Interdisciplinary Reviews: Systems Biology and Medicine*, 3(1), 115–125. 2011.
- [16] Paweletz, N. and M. Knierim M. Tumor-related angiogenesis. *Critical Reviews in Oncology and Hematology*, 9, 197–242. 1989.
- [17] Paku, S. and N. Paweletz. First steps of tumor-related angiogenesis. *Laboratory Investigation*, 65, 334–346. 1991.
- [18] Schor S. L., Schor A. M., Brazill G. W. The effects of fibronectin on the migration of human foreskin fibroblasts and Syrian hamster melanoma cells into three-dimensional gels of lattice collagen fibres. *Journal of Cell Science*, 48, 301–314, 1981.
- [19] Bowersox J. C. and Sorgente N. Chemotaxis of aortic endothelial cells in response to fibronectin. *Cancer Research* 42, 2547–2551. 1982.
- [20] Quigley J. P., Lacovara J., and Cramer E. B. The directed migration of B-16 melanoma-cells in response to a haptotactic chemotactic gradient of fibronectin. *Journal of Cell Biology* 97, A450–451. 1983.
- [21] Stokes C. L., Lauffenburger D. A., and Williams S. K. Migration of individual microvessel endothelial cells: stochastic model and parameter measurement. *Journal of Cell Science*, 99: 419–430. 1991.
- [22] Stokes C. L., Rupnick M. A., Williams S. K., and Lauffenburger D. A. Chemotaxis of human microvessel endothelial cells in response to acidic fibroblast growth factor. *Laboratory Investigation*, 63, 657–668, 1991.
- [23] Stokes C. L. and Lauffenburger D. A. Analysis of the roles of microvessel endothelial cell random motility and chemotaxis in angiogenesis. *Journal of Theoretical Biology*, 152, 377–403. 1991.
- [24] Anderson, A. R. A. and Chaplain, M. Continuous and discrete mathematical models of tumour-induced angiogenesis, *Bulletin of Mathematical Biology*, 60, 857–900. 1998.
- [25] 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.
- [26] Nvidia Corporation. *CUDA C programming guide*. Version 6.0. 2014.
- [27] Cheng, J., Grossman, M and McKercher, Ty. Professional CUDA C Programming. Wrox. 2014.
- [28] 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.
- [29] 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.
- [30] Cecilia, J. M., Garcia, J. M. and Ujaldon, M. CUDA 2D stencil computations for the Jacobi method. Springer, 173–183. 2012.