Computational Biology and Bioinformatics
2015; 3(5): 65-73
Published online August 25, 2015 (http://www.sciencepublishinggroup.com/j/cbb)
doi: 10.11648/j.cbb.20150305.11
ISSN: 2330-8265 (Print); ISSN: 2330-8281 (Online)

A System of Coupled Nonlinear Partial Differential Equations Describing Avascular Tumour Growth Are Solved Numerically Using Parallel Programming to Assess Computational Speedup

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 System of 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, 2015, pp. 65-73. doi: 10.11648/j.cbb.20150305.11

Abstract: The challenging issues of cancer prevention and cure lie in the need for a more detailed knowledge of the internal processes and mechanisms of tumour growth. We present a mathematical model of avascular tumour growth formulated in a system of coupled nonlinear PDEs. The interaction between the surrounding tissue and cell motility of the developing tumour are also included to more realistic replicate an *in-vivo* environment. The mathematical model is solved using finite difference methods and implemented in the C programming language. The CUDA programming framework is then introduced to allow a parallelisation of the sequential C implementation. Results show a dramatic Speedup of around 26x that of conventional implementations in C. Such increased computational efficiency clearly highlights the possibility of improvements in the numerical simulation of more complex mathematical models of 2D and 3D tumour growth, such as angiogenesis and vascularisation. Parallelisation of such models can greatly facilitate researchers, clinicians and oncologists by performing time-saving *in-silico* experiments that have the potential to highlight new cancer treatments and therapies without the need for the use of valuable resources associated with excessive pre-clinical trials.

Keywords: Avascular Tumour Growth, Multicellular Spheroids (MCS), Parallel Programming, Compute Unified Device Architecture (CUDA), Graphical Processing Unit (GPU)

1. Introduction

Solid tumours usually undergo a period of avascular growth, after which they become dormant for a sustained period without access to a sufficient supply of essential nutrients (e.g. oxygen and glucose) to continue to proliferate. If the quiescent tumour eventually invades the surrounding tissue, a network of blood vessels can develop through the process of angiogenesis. With the tumour now having access to a rich supply of nutrients provided through its own blood supply, as well as other growth promoting factors, it enters into full vascularisation. While the difference between cancerous and healthy regions are apparent in the avascular stage, this difference is less clear during vascular growth where the tumour becomes aggressive and spreads to other parts of the body through the blood stream via metastasis. On the other hand, the avascular tumour is generally considered to be a solid mass, growing through mitosis and thought, at this early stage of development, to be non-invasive to the surrounding healthy tissue [1]. Understanding such a complex mechanism greatly facilitates the knowledge required to address the proliferation of fully vascular tumours. The very early stages of tumour growth are often undetectable due their small mass size. However, avascular tumour growth is relatively easy to replicate *in vitro*. Such observations strongly suggests that very early stage solid tumours remain approximately spherical as they grow. Indeed, multicellular spheroids (MCS) have been widely used has models of *in vitro* avascular tumour growth from which a deeper insight into tumour heterogeneity can be gained for many years [2-4].

A typical MCS is composed of three distinct regions. A thin outer rim (a few hundred μm thick) of proliferating cells in contact with a rich supply of nutrients that surrounds a thicker quiescent band of dormant cells. These quiescent cells, although not proliferating, are not dead but lay dormant awaiting the necessary nutrients so that they can carry on dividing through mitosis. Thus quiescence is a reversible state [5]. The final region is an inner core of necrotic cells starved of vital nutrients, forming a central mass of cell debris. At some stage during early growth, proliferation and necrosis reach an equilibrium and the avascular tumour reaches a *limit size* which is thought to be around 1-3 mm in diameter consisting of several million cells [6]. Figure 1 shows the histology of human colon adenocarcinoma HT29 MCS at two levels of magnification. The three phase structure of the MCS is clearly apparent with the inner necrotic core surrounded by a quiescent region subsequently enclosed by an outer rim of proliferating cells.

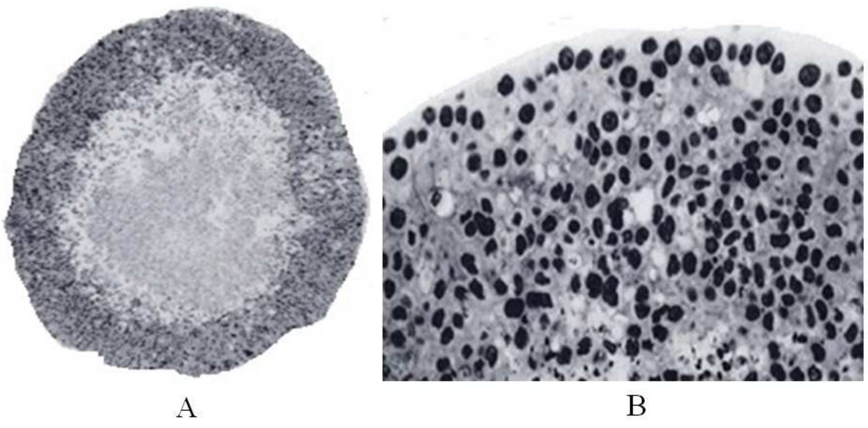

Figure 1. *Histology of human colon adenocarcinoma HT29 MCS [Diameter 1.4 mm]. A. Central section of spheroid of 1415 µm diameter after 18 days in culture demonstrating a viable rim of cells surrounding an extensive necrotic core (x 60). B. High magnification (x 310) showing the structural arrangement of MCS in the viable rim of approximately 225 µm thickness (Sutherland et al., 1986).*

Progress in mathematical modelling of avascular tumour growth has largely been driven by biological and clinical observations through *in vitro* and *in vivo* experiments, biopsies and autopsies. The majority of mathematical models focus on the development of a set of spatial-temporal reaction-diffusion equations that describe nutrient concentrations coupled with population growth, inhibition factors, and cell motility. 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 Sherratt in [7] and Sherratt and Chaplain in [8] that describe avascular tumour growth within a closely-packed cell population model. Sherratt and Chaplain in [8] proposed a more realistic model that additionally takes into account the contact between other cells that naturally surround a tumour, such as those found in the epithelium. Indeed, the most common types of cancer, such as breast, lung, prostate, and colon are carcinomas that develop within the layers of epithelial tissues. The model developed here should be extremely useful in further understanding the formation of avascular tumour growths *in-vivo*.

2. Mathematical Model of Avascular Tumour Growth

When considering closely-packed cell populations such as those found in the epithelium, we need to consider the effect of reduced cell motility through natural contact with other neighbouring cells. Such a phenomenon, well documented in many types of cells [9], is known as *contact inhibition of migration*. Of course, contact inhibition will not prevent a tumour from growing in size, but it will have a significant effect on the overall growth dynamics of the tumour. As previously discussed, the model proposed here by Sherratt and Chaplain in [8] is oriented towards an *in vivo* rather than *in vitro* environment, and crucially allows for nutrient supply from underlying tissue. Also, rather than assuming that proliferating, quiescent and necrotic cell regions have distinct compartments, we consider the transition between states has a gradual process [10]. Furthermore, if we are not assuming separate compartments, then we must formulate the model in terms of continuous cell densities. We denote these densities; $p(x, t)$, $q(x, t)$, and $n(x, t)$ for proliferating, quiescent, and necrotic cells, respectively. In addition, since the tumour is assumed to be growing in the epithelium, its growth will naturally be inhibited by the surrounding epithelial cells, denoted $s(x, t)$. These cells will themselves be motile, and will divide at a rate that depends on the nutrient concentration $c(x, t)$. In order to include contact inhibition in the random motility of tumour cells, the overall viable cell flux is fractionated evenly between the proliferating, quiescent and surrounding cell densities. This is based on the assumption that the three cell populations have equal motility (obviously necrotic cells have no motility). The system of coupled nonlinear PDEs that describe this system can be written as (referred to from now as Model 1):

$$\frac{\partial p}{\partial t} = \nabla.\left(\frac{p}{p+q+s}\nabla(p+q+s)\right) + g(c)p(1-p-q-n-s) - f(c)p$$

$$\frac{\partial q}{\partial t} = \nabla.\left(\frac{q}{p+q+s}\nabla(p+q+s)\right) + f(c)p - h(c)q$$

$$\frac{\partial s}{\partial t} = \nabla.\left(\frac{s}{p+q+s}\nabla(p+q+s)\right) + g(c)s(\Gamma - p - q - n - s)$$

$$\frac{\partial n}{\partial t} = h(c)q$$

And the nutrient concentration, c is given by:

$$c = \frac{c_0\,\gamma[1-\alpha(p+q+s+n)]}{\gamma+p}$$

Where α and Γ are dimensionless parameters and c_0 is the nutrient concentration in the absence of a tumour cell population. A cell density of one corresponds to a completely closely-packed cell population. In the direction of the core of the tumour, we assume that a subset of proliferating cells, with limited access to essential nutrients, will become quiescent at a rate $f(c)$ and some quiescent cells, which are totally starved of nutrients, will undergo necrosis at a rate $h(c)$. We further assume that the rate (per cell) of entry into quiescence $f(c)$ is larger than the rate of necrosis $h(c)$, at any given nutrient level, so that $f(c) > h(c)$. In addition, we assume the growth rate $g(c)$ of the proliferating cells is proportional to the concentration $c(x, t)$ of nutrients and limited by the neighbouring effects of the total cell population. The nutrients are assumed to pass through the surface of the tumour and diffuse into the interior through the intracellular space sufficiently fast enough that the local nutrient concentration $c(x, t)$ can be approximated by a quasi-steady state. The functional forms of $f(\cdot)$, $g(\cdot)$ and $h(\cdot)$ along with the chosen model parameters are shown in Table 1. Obviously, the type of functional form and parameter values will affect cell density in each region, as well as the overall speed of tumour growth.

Table 1. *Functional forms and parameter values used in the numerical solution to Model 1.*

Function forms	Parameter values
$f(c) = \frac{1}{2}[1 - \tanh(4c - 2)]$	c0 = 1
$g(c) = 1 + 0.2c$	α = 0.9
$h(c) = \frac{1}{2}f(c)$	γ = 10
	Γ = 0.4

The following initial and zero-flux Neumann boundary conditions are imposed (no boundary conditions are required for n).

Initial conditions:

$$p(x,0) = 0.01e^{-0.1x}$$

$$q(x,0) = 0$$

$$n(x,0) = 0$$

$$s(x,0) = \Gamma[1 - 0.01e^{-0.1x}]$$

Neumann boundary conditions:

$$\frac{\partial p}{\partial x} = \frac{\partial q}{\partial x} = \frac{\partial s}{\partial x} = 0$$

For systems of coupled nonlinear PDEs like Model 1, finite difference methods (FDM) are the dominant approach to finding a suitable numerical solution. In this paper, Model 1 is solved numerically using an explicit finite difference (EFD) scheme as discussed in the next section.

2.1. Explicit Finite Difference Scheme

There are three main kinds of FDMs in common use; implicit, explicit and Crank-Nicolson. In this paper, we implement the explicit scheme as it is the most parallelisable of the three methods. Although the EFD scheme is widely used since it is relatively easy to apply, its computational complexity can grow dramatically with increasing accuracy. The finite-difference scheme generally involves producing a set of discrete numerical approximations to the partial derivative, often in a *time-stepping* manner. In this way, explicit time-marching methods, such as EFD can be thought of in terms of being naturally parallel. As well as the discretised model, the EFD scheme also requires both the initial and boundary conditions. Initial conditions determine the state of the system at $t = 0$. Whilst boundary conditions define the behaviour at the edges, such as the Neumann boundary conditions relating to zero-flux at the boundaries of the avascular tumour imposed in our Model 1.

To use a FDM to approximate the solution to Model 1, we must first discretise across a relevant spatial-temporal domain by dividing it into a uniform grid. Then, Model 1 can be solved *explicitly* since it is possible to find the value of any inner node n +1 from the value of preceding neighbouring nodes at n. At the endpoints $j = 0$ and $j = J$, we are on the edges of the grid, so we simply apply the given boundary conditions. At every time step, we calculate the value of each node on the grid based on the discretised system. The grid spacing is chosen for numerical stability, and we have to be mindful that the EFD scheme is only conditionally stable. That is, the EFD is known to be numerically stable and convergent only with a suitable choice of model parameters.

Although other more advanced numerical methods besides the EFD scheme are available, such as the alternating direct implicit (ADI) method, each method has its own advantages and disadvantages in terms of implementation complexity, numerical stability and convergence. In this paper, we decided to use the EFD scheme since the form of Model 1 allowed for a suitably complex discretisation requiring numerous calculations that could be usefully parallelised and with the correct choice of parameters both numerically stable and convergent.

2.2. The Discretisation of Model 1

Note that Model 1 may be written as:

$$\frac{\partial p}{\partial t}=\frac{\partial}{\partial x}\left(\frac{p}{p+q+s}\frac{\partial(p+q+s)}{\partial x}\right)+g(c)p(1-p-q-n-s)-f(c)p$$

$$\frac{\partial q}{\partial t}=\frac{\partial}{\partial x}\left(\frac{q}{p+q+s}\frac{\partial(p+q+s)}{\partial x}\right)+f(c)p-h(c)q$$

$$\frac{\partial s}{\partial t}=\frac{\partial}{\partial x}\left(\frac{s}{p+q+s}\frac{\partial(p+q+s)}{\partial x}\right)+g(c)s(\Gamma-p-q-n-s)$$

$$\frac{\partial n}{\partial t}=h(c)q$$

So, using a forward finite difference approximation for the time derivative and a central difference approximation for the spatial derivative (FTCS), gives:

$$p_j^{n+1}=p_j^n+\Delta t\left[u_j^n+g\left(c_j^n\right)p_j^n\left(1-p_j^n-q_j^n-n_j^n-s_j^n\right)-f\left(c_j^n\right)p_j^n\right]$$

$$q_j^{n+1}=q_i^n+\Delta t[v_i^n+f(c_i^n)p_i^n-h(c_i^n)q_i^n]$$

$$s_j^{n+1}=s_j^n+\Delta t\left[w_j^n+g\left(c_j^n\right)s_j^n\left(\Gamma-p_j^n-q_j^n-n_j^n-s_j^n\right)\right]$$

$$n_j^{n+1}=n_j^n+\Delta t\left[h\left(c_j^n\right)q_j^n\right]$$

Where

$$u_j^n=\frac{\left(p_{j+1}^n-p_{j-1}^n\right)r_j^n\left(r_{j+1}^n-r_{j-1}^n\right)+4p_j^nr_j^n\left(r_{j+1}^n-2r_j^n+r_{j-1}^n\right)-p_j^n\left(r_{j+1}^n-r_{j-1}^n\right)^2}{4(\Delta x)^2\left(r_j^n\right)^2}$$

$$v_j^n=\frac{\left(q_{j+1}^n-q_{j-1}^n\right)r_j^n\left(r_{j+1}^n-r_{j-1}^n\right)+4q_j^nr_j^n\left(r_{j+1}^n-2r_j^n+r_{j-1}^n\right)-q_j^n\left(r_{j+1}^n-r_{j-1}^n\right)^2}{4(\Delta x)^2\left(r_j^n\right)^2}$$

$$w_j^n=\frac{\left(s_{j+1}^n-s_{j-1}^n\right)r_j^n\left(r_{j+1}^n-r_{j-1}^n\right)+4s_j^nr_j^n\left(r_{j+1}^n-2r_j^n+r_{j-1}^n\right)-s_j^n\left(r_{j+1}^n-r_{j-1}^n\right)^2}{4(\Delta x)^2\left(r_j^n\right)^2}$$

$$c_i^n=\frac{c_0\gamma\left[1-\alpha\left(p_j^n+q_j^n+n_j^n+s_j^n\right)\right]}{\left(\gamma+p_j^n\right)}$$

$$r_j^n=p_j^n+q_j^n+s_j^n$$

Δx and Δt refer to the time steps and grid spacing, respectively. We partition the x-axis into intervals of length Δx and t-axis into intervals of length Δt. The (x-t)-plane is divided into a uniform grid with lines parallel to $0t$, defined by:

$$x_j=j\Delta x, j=0,1,2,\dots,J$$

And by lines parallel to $0x$ defined by:

$$t_n=n\Delta t, n=0,1,2,\dots,T$$

So, p_i^n (for example) refers to the density of the proliferating cells at the nth time interval and jth spatial position.

3. Implementation

The main aim of this paper is to investigate potential computational efficiency and Speedup of algorithm execution moving from a serial to parallel platform. This will be quantified based on analysing the Speedup of an implementation of an EFD scheme for a system of coupled nonlinear PDEs describing avascular tumour growth.

3.1. Hardware

A microprocessor contains a central processing unit (CPU) called a core that performs arithmetic and logic operations at high speeds. A single-core processor performs one operation at a time, but can efficiently switched between different tasks, seemingly executing many computations simultaneously. A quad-core processor, by contrast, has four CPUs on a single chip and executes four separate operations in parallel, greatly enhancing compute capability. Moreover, CPU cores are generally designed to work well with single-threaded applications. To improve thread-performance the CPU core employs an architecture that exploits the potential for parallel instruction. That is, each CPU core supports *scalar* and *single instruction multiple data* (SIMD) operations so that the execution of multiple operations per cycle are allowed. However, such architecture restricts the size and complexity of the processor limiting the number of cores that can be integrated on a single die. In contrast, the graphics processing unit (GPU) trade off fast single thread performance and clock speed for high throughput. The GPU consists of an array of highly threaded *streaming multiprocessors* (SM) with each having their own individual *streaming processors* (SP) that share control logic and instruction cache. Each SM consists of a single fetch unit and eight scalar units. So that each instruction is retrieved and executed in parallel on all eight scalar units over four cycles for 32 data elements (a *warp*) [11]. This keeps the available area of each SM relatively small, and therefore more SMs can be packed per die, as compared to the number of CPU cores.

The hardware used for the sequential C 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 program was also developed in Microsoft® Visual Studio 2012 using CUDA version 7.0 and tested on an Nvidia GeForce® GTX™ 780 GPU card with compute capability 3.5. Table 2 gives a more detailed specification of the CPU and GPU hardware.

Although a quad core processor has multiple CPUs, they share other components, such as random access memory (RAM). Memory bandwidth, the speed at which the processor chip accesses data in RAM, can become a *bottleneck* when all the processors need to access the same information and store data. For this reason, a quad core rarely performs at exactly four times that of a single core, but instead typically runs between two and four times. From Table 2, note that the Core™ i7 provides a memory bandwidth of 25.6 GB/s, while the GTX™ 780 provides a bandwidth of 288.4 GB/s resulting in a very useful peak bandwidth ratio of ~11.3x.

Table 2. *CPU and GPU hardware specifications.*

	Intel® Intel® Core™ i7-4790K (CPU)	**Nvidia GeForce® GTX™ 780 (GPU)**
Clock speed (GHz)	4.00	0.863
# of cores	4	2,304
Memory bandwidth (GB/s)	25.6	288.4

3.2. Performance Benchmarks

The test platform will make use of C for the sequential implementation and the CUDA programming framework for the parallel implementation of the EFD scheme. The C language is an obvious choice for professional development and a language that is heavily adopted throughout the computational biology community. One of the most important functions of any programming language is to provide facilities for managing memory and the objects that are stored in memory. C provides several powerful methods of allocating and managing memory making it an extremely versatile especially when considering computational efficiency and increased speed of code execution. C is also the natural choice for any CUDA enabled development since it relies itself on extensions from the C language as a basis for its own implementation.

We are fully aware that it is possible to implement C in a parallel context, indeed, the Gauss-Seidel red black method (GSRB) is a FDM for solving systems of coupled nonlinear PDEs in parallel that can greatly enhance the speed of sequential code execution. Nevertheless, performance here will be established on multicore processors executing sequential code in C that will subsequently be adapted to an equivalent parallel algorithm under the CUDA programming framework. The actual platform performance will be based on the execution time of each algorithm implementation and subsequent Speedup.

3.2.1. Amdahl's Law

The theoretical Speedup S of an algorithm is given by Amdahl's law [12]:

$$S = \frac{T(1)}{T(n)}$$

Where $T(n)$ is the execution time when using n processors. Here, the execution time is the difference between two clock statements in each of the main algorithms. One placed at the start, and the other at the end of the *kernel* looping routine (including the device to host transfer in CUDA). Thus, execution time represents the time taken to complete the entire process of a single simulation of the numerical solution to Model 1.

3.2.2. Floating Point Operations

We can estimate the likely Speedup of our C and CUDA implementations by calculating the *floating point operations per second* (FLOPS). FLOPS 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 FLOPS (GFLOPS) every time its internal clock ticks. These clock ticks are called *cycles* measured by the processor clock speed. It is important to note that there is quite a difference between *single-precision* and *double-precision* FLOPS. A processor that is capable of many single-precision GFLOPS may only be capable of a small fraction of that many double-precision calculations. For the Core™ i7-4790K CPU (4 core) processor Intel® assume the following simple multiplication formula to determine the CPU GFLOPS:

$$clock\ speed\ x\ cores\ x\ FLOPS\ per\ clock\ cycle\ x\ MAD\ instructions$$

Where MAD are the number of Multiply-Add instructions per clock cycle as per the processor specifications. Therefore, CPU GFLOPS are given by:

$$4\ x\ 4\ x\ 2\ x2 = 64$$

i.e. 64 GFLOPS (double precision float point)

Here we have assumed two double precision floating point numbers per clock cycle. For single precision, we need to double the number i.e. 128 GFLOPS (single precision float point). For GPU GFLOPS, we have similarly:

$$0.863\ x\ 2304\ x\ 2 = 3977$$

i.e. 3977 GFLOPS (single precision floating point)

Based on these values, the estimated Speedup and peak performance between the CPU and GPU implementations should be in the region of 31.3x. Of course, this value is very subjective and reliant on numerous other factors, such as correct code optimisation, full usage of MAD instructions, efficient use of memory, etc. and therefore should only be used as a guideline.

3.3. The CUDA Programming Framework

For some time the usual method for improving performance on the CPU was to simply increase the processor clock speed. Since then, high performance computing has provided dramatic improvements in computational efficiency by gradually increasing the number of processor cores. Indeed, the majority of computers today have at least four or more cores per die allowing multicore processing capabilities and parallel implementations effortlessly. In a similar development driven path, early GPUs were initially designed for producing colour texture coordinates matching pixels on the screen using a programmable arithmetic unit knows as a *pixel shader*. Since the arithmetic being performed on the input colours and textures was completely controlled by the programmer, it was soon realised that these input 'colours' could effectively be *any* data type. If the inputs were considered numerical data, the pixel shaders could be programmed to perform arbitrary computations on this other type of data. The new compute unified device architecture (CUDA) developed by Nvidia completely revolutionised computation on the GPU. The CUDA programming framework included a unified *shader pipeline* that allows each arithmetic logic unit (ALU) on the processor to be marshalled by a program intending to perform general-purpose computations (see Figure 2).

GPU-accelerated applications run the sequential part of their workload on the CPU, which is optimised for single-threaded performance, while accelerating parallel processing on the GPU. However, since the GPU is a coprocessor usually on a separate PCI-express card, data must first be explicitly copied from the system memory to global memory on the GPU. For this reason, performance bottlenecks are often minimised by making intelligent use of *memory bandwidth*. Despite these drawbacks, GPUs are an extremely viable candidate for performing highly intensive computations that exhibit high levels of parallelism.

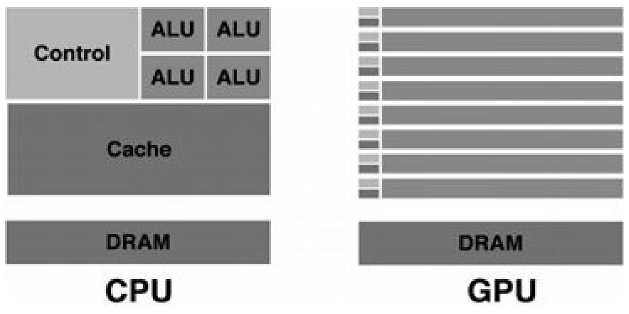

Figure 2. *A schematic of the CPU vs. GPU architecture [11].*

3.3.1. Threads, Blocks, Grids and Memory

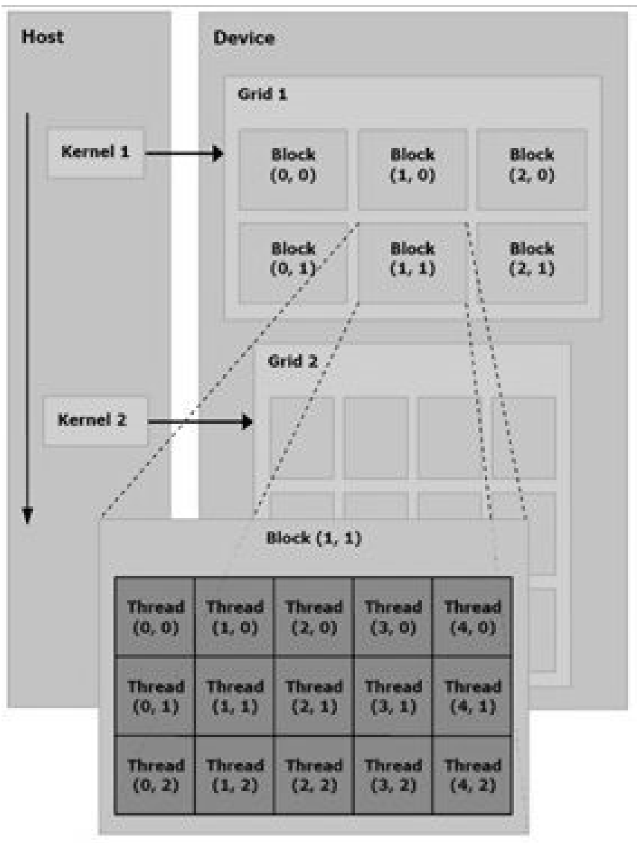

Figure 3. *Schematic of the arrangement of threads, blocks, and grids between the host and device [11].*

The CUDA programming framework provides an API for programmers that exposes the underlying GPU architecture, which is a collection of single instruction, multiple data (SIMD) processors capable of switching between thousands of *threads*. CUDA further extends C by allowing the programmer to define C functions known as *kernels*, that are executed N times concurrently by N different CUDA threads.

A kernel is defined using the __global__ declaration specifier and the number of CUDA threads that execute the kernel is specified using a <<<GRIDSIZE, BLOCKSIZE>>> execution configuration syntax. Each thread that executes the kernel is given a unique thread id that is accessible within the kernel through the threadIdx variable. In CUDA, the threads are grouped into blocks and the blocks are grouped into grids (see Figure 3). There is a limit to the number of threads per block, on current GPUs, a thread block may contain up to 1,024 threads. A thread block size of 256 threads, although arbitrary, is a common choice [11].

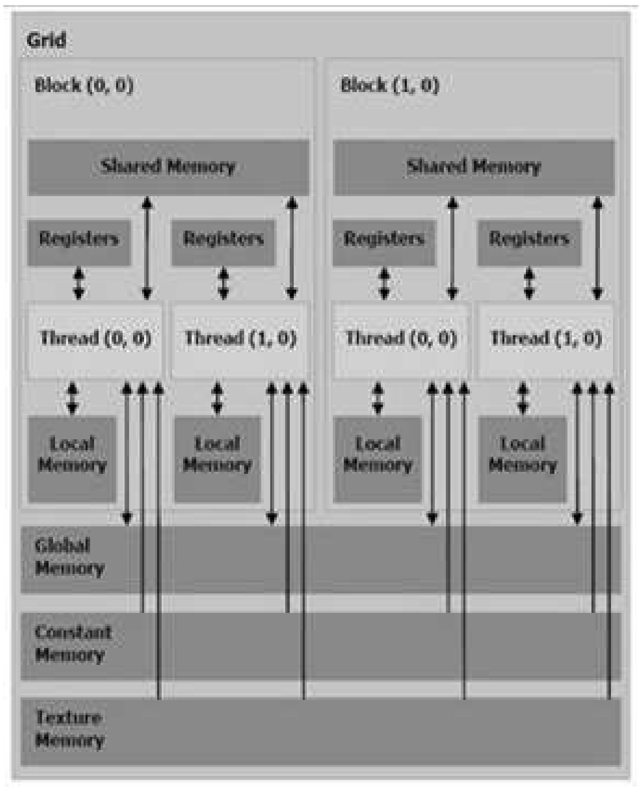

Figure 4. Schematic of the arrangement of memory within grids and blocks [11].

Threads within a block can cooperate by sharing data through *shared memory* and by synchronising their execution to coordinate memory access. More precisely, one can specify synchronisation points in the kernel by calling the __syncthreads() function [11]. Indeed, CUDA makes available several different types of accessible memory options. For instance, one very practical type of memory is *constant memory*. Constant memory is used for data that will not change during the execution of a kernel and in some situations can reduced memory bandwidth. CUDA also provides access to *shared memory*. With this type of memory it is possible to modify variables resident in the shared memory. CUDA treats variables in shared memory differently to standard variables. That is, CUDA creates copy of the variables for each block it launches on the GPU and thereby allows every thread in that block shared access to the memory. This is extremely useful since a major drawback of current GPU vs. CPU implementations is the need to continually transfer data between host and device. The correct usage of shared memory along with synchronisation can greatly alleviate some of these efficiency problems. Also, threads cannot see or modify the copied variable that can be seen in the other blocks and so provides a favourable mechanism by which threads within a block can communicate and collaborate on workloads. Furthermore, shared memory buffers reside physically on the GPU thereby greatly improving the latency of access and per-block programmable management cache. Other types of available memory include g*lobal memory*; the slowest of the memory available but the largest in size, and *texture memory* (see Figure 4).

3.3.2. The C and CUDA Implementations

Algorithm 1. *C implementation for the EFD scheme.*

```
1:    Define model parameters
2:    Declare pointers
3:    Initialise array memory
4:    Set initial conditions
5:    Start clock()
6:    for n = 1:Nt do
7:    for j = 1:Nx do
8:    Update nodes $p_i^{n+1}$, $q_i^{n+1}$, $s_i^{n+1}$, $n_i^{n+1}$
9:    end for
10:   end for
11:   End clock()
12:   Print results
13:   Free memory
```

The CUDA programming model 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 using the C calloc function. calloc also initialises all elements to zero and subsequently returns a null pointer if it cannot allocate a linear block of adequate size. The algorithm developed to implement the C program for the EFD scheme is shown in Algorithm 1.

Moving from C to CUDA requires additional coding, as well as some manipulation of the kernel. Note that under the CUDA programming framework, we refer to the CPU as the *host* and the GPU as the *device*. With CUDA, 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. Basically, there are three steps that are essential to the successfully execution of a kernel on the GPU. Firstly, data must 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 then be 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(). As already mentioned, each block of threads has access to shared memory. Making the correct use of this memory can reduce the amount of data that has to be transferred from global memory, which is typically the performance bottleneck in many GPU vs. CPU algorithms [11]. The algorithm for the implementation of the CUDA kernel for the EFD scheme is shown in Algorithm 2.

Algorithm 2. *CUDA kernel for the EFD scheme.*

```
1:  Get current thread index
2:  if (i<N) then
3:      Update nodes $p_j^{n+1}$, $q_j^{n+1}$, $s_j^{n+1}$, $n_j^{n+1}$
4:  end if
5:  Synchronise threads
```

Algorithm 3 shows the main CUDA implementation for the EFD scheme.

Algorithm 3. *CUDA main implementation for the EFD scheme.*

```
1:  Define model parameters
2:  Declare host and device pointers
3:  Allocate host and device memory
4:  Set initial conditions
5:  Copy host arrays to device
6:  Declare GRIDSIZE and BLOCKSIZE
7:  Start clock()
8:  for n = 1:Nt do
9:      for j = 1:Nx do
10:         Launch kernel<<<GRIDSIZE,BLOCKSIZE;>>>
11:     end for
12:     Synchronise threads
13: end for
14: End clock()
15: Copy device arrays to host
16: Print results
17: Free memory
```

4. Results and Discussion

Table 3 shows the execution time for the C and CUDA implementations EFD scheme solution to Model 1.

Table 3. *Performance results for C and CUDA implementations and subsequent Speedup of CUDA over C.*

Iterations	Speedup
100,000	1.03
500,000	1.63
1,000,000	3.04
5,000,000	22.5
10,000,000	29.2

Table 3 shows that at low iterations there is practically no difference between the implementations in C or CUDA. In fact, this is entirely what we would expect since the power of the GPU does not show itself unless it is sharing the majority of the workload and handling millions (and billions) of calculations. Indeed, at such low iterations the bottleneck between transferring data between the host and device global memory clearly hinders any computational improvement in performance. However, when considering iterations in the several millions, CUDA far outstrips C in execution time and efficiency. Indeed, the GPU showed a Speedup of an average 25.9x that of the sequential C implementation and very close to that of the estimated peak performance of 31.3x Speedup based on the GFLOPS calculation.

Whilst we only considered a 1D implementation, we have shown that the CUDA programming framework can be extremely valuable in dramatically increasing execution time and efficiency. Moreover, when challenged with more complex systems of coupled nonlinear PDEs, for example in studying the processes of angiogenesis in breast cancer or vascular brain tumours, the need for parallel algorithms is essential. Indeed, it is assumed that the implementation of more advanced numerical schemes, that can be considered massively parallel, will provide an extremely productive computational methodology. The authors have already begun implementing more complex 2D and 3D numerical solutions on the CUDA programming framework with very promising initial results. Moreover, greater biological insight and clinical knowledge can be gained from such implementations.

5. Conclusions

Obtaining a numerical solution to a system of coupled nonlinear PDEs can be a daunting computational task, even in 1D. The aim of this paper was to show that a numerical solution to a system of coupled nonlinear PDEs, such as those frequently encountered in computational biology, could benefit from parallelisation. A suitable platform was shown to be Nvidia's CUDA programming framework which substantially improved computational efficiency and execution time in a 1D implementation of an EFD scheme, returning Speedups around 26x that of conventional methodologies.

The modelling of avascular tumours is often seen as a first step towards the development of more complex models of later stage tumour growth, such as angiogenesis and vascularisation. Mathematical and computational modelling is playing an increasingly important role in helping biomedical researchers in understanding the different aspects of solid tumour dynamics. *In-silico* experiments and simulations, such as those performed in this paper, give researchers, clinicians and oncologists the tools and opportunity to observe effects of different treatments on cancerous cells in realistic time frames. This will inevitably lead to more rapid improvements in effectual therapeutic strategies as well as aiding in the discovery of new forms of treatment. Personalised healthcare, adapted to patient specific symptoms, could be used in therapy planning by suggesting irradiation regions adapted to growth dynamics or optimal temporal distribution of chemotherapy through computer simulation. In fact, getting new chemotherapy and other anti-cancer drugs into pre-clinical trials is relying more and more on supportive evidence from - *in-silico* experiments. The ability to use advanced computational methods to simulate the virtual stages of pre-clinical trials greatly reduces time to market and laboratory resources. If we consider the billions of dollars spent on cancer research and that lost in unnecessary and time consuming clinical trials, the case for *in-silico* experiments that make use of advancements in computational technology, is a first port of call for any research establishment or pharmaceutical company dedicated to fighting diseases such as cancer.

References

[1] Araujo, R. and McElwain, D. A history of the study of solid tumor growth: the contribution of mathematical modelling. *Bulletin of Mathematical Biology*, 66. 2004.

[2] Sutherland, R.M. and Durand, R. E. Growth and cellular characteristics of multicell spheroids. *Recent Results in Cancer Research* 95, 24-49. 1984.

[3] Sutherland, R. M., Sordat, B., Bamat, J., Gabbert, H., Bourrat, B., Mueller-Klieser,W. Oxygenation and differentiation in multicellular spheroids of human colon carcinoma. *Cancer Research*, Vol. 46, 5320–5329. 1986.

[4] Sutherland, R. M. Cell and environment interaction in tumour microregions: the multicell spheroid model. *Science*, Vol. 240, 177-184. 1988.

[5] Freyer, J. P. and Schor, P. L. Regrowth of cells from multicell tumour spheroids. *Cell and Tissue Kinetics*, 20, 249. 1987.

[6] Orme, M. and Chaplain M.A.J. A mathematical model of vascular tumor growth and invasion. *Mathematical and Computational Modelling*, 23. 1996.

[7] Sherratt, J. A. Wave front propagation in a competition equation with a new motility term modelling contact inhibition between cell populations. *Proceedings of the Royal Society of London*, A456, 2365–2386. 2000.

[8] Sherratt, J. A and Chaplain M.A.J. A new mathematical model for avascular tumour growth. *Journal of Mathematical Biology*, Vol. 43, pp291–312. 2001.

[9] Huttenlocher, A., Lakonishok, M., Kinder, M., Wu, S., Truong, T., Knudsen, K. A. and Horwitz, A. F. Integrin and cadherin synergy regulates contact inhibition of migration and motile activity. *Journal of Cell Biology* 141, 515-526. 1998.

[10] McElwain, D. L. S. and Pettet, G. J. Cell migration in multicell spheroids: swimming against the tide. *Bulletin of Mathematical Biology*, 55, 655–674. 1993.

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

[12] Amdahl, G. M. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. *AFIPS Conference Proceedings* (30): 483–485. 1967.