Investigación

# Programación en paralelo GPGPU del método en diferencias finitas FTCS para la ecuación del calor

# FTCS finite difference scheme GPGPU parallel computing for the heat conduction equation

Vicente Cuellar Moro, Miguel Martín Stickle
y Manuel Pastor Pérez

Revista de Investigación

Volumen IV, Número 1, pp. 107–126, ISSN 2174-0410
Recepción: 7 Ago'13; Aceptación: 10 Mar'14

1 de abril de 2014

**Resumen**

En el presente artículo se muestran las ventajas de la programación en paralelo resolviendo numéricamente la ecuación del calor en dos dimensiones a través del método de diferencias finitas explícito centrado en el espacio FTCS. De las conclusiones de este trabajo se pone de manifiesto la importancia de la programación en paralelo para tratar problemas grandes, en los que se requiere un elevado número de cálculos, para los cuales la programación secuencial resulta impracticable por el elevado tiempo de ejecución. En la primera sección se describe brevemente los conceptos básicos de programación en paralelo. Seguidamente se resume el método de diferencias finitas explícito centrado en el espacio FTCS aplicado a la ecuación parabólica del calor. Seguidamente se describe el problema de condiciones de contorno y valores iniciales específico al que se va a aplicar el método de diferencias finitas FTCS, proporcionando pseudocódigos de una implementación secuencial y dos implementaciones en paralelo. Finalmente tras la discusión de los resultados se presentan algunas conclusiones.

**Palabras Clave:** Computación en paralelo, GPU, Diferencias finitas, ecuación del calor.

**Abstract**

In this paper the advantages of parallel computing are shown by solving the heat conduction equation in two dimensions with the forward in time central in space (FTCS) finite difference method. Two different levels of parallelization are consider and compared with traditional serial procedures. We show in this work the importance of parallel computing when dealing with large problems that are impractical or impossible to solve them with a serial computing procedure. In the first section a summary of parallel computing approach is presented. Subsequently, the forward in time central in space (FTCS) finite difference method for the heat conduction equation is outline, describing how the heat flow equation is derived in two dimensions and the particularities of the finite

difference numerical technique considered. Then, a specific initial boundary value problem is solved by the FTCS finite difference method and serial and parallel pseudo codes are provided. Finally after results are discussed some conclusions are presented.

**Keywords:** Parallel computing, GPU, finite differences, heat equation.

# 1. Parallel computing using graphic processing unit

## 1.1. Introduction

Traditionally, computer software has been written for serial computation. To solve a problem, an algorithm is constructed and implemented as a serial stream of instructions. These instructions are executed on a central processing unit (CPU) on one computer. Only one instruction may execute at a time. After that instruction is finished, the next is executed.

Parallel computing, on the other hand, uses multiple processing elements simultaneously to solve a problem. This is accomplished by breaking the problem into independent parts so that each processing element can execute its part of the algorithm simultaneously with the others. The processing elements can be diverse and include resources such as a single computer with multiple processors, several networked computers, specialized hardware, or any combination of the above.

Although the diversity of parallel machines is too big to describe them here, they can be classified into two big groups. On one side we have the distributed memory systems, on the other the shared memory systems. The distributed memory systems are those in which every computing unit has a private memory which only he can access directly. Shared memory systems are those in which each computing node can access all the memory of the system. It is important to note that these systems are sometimes combined as the computing unit in a distributed system is usually a parallel machine on its own, whether a multicore processor or any other type, which in itself is a system of the shared memory kind.

This distinction between models is critical as it describes how each system handles the fundamental problem of parallel computing, which is the data locality problem. Except in the simplest algorithms, often called “embarrassingly parallel problems” parallel machines have to deal with the access to information between its computing units, as some units need information calculated by different units.

Shared Memory Systems can have simplified solution compare to the distributed ones, as the whole system memory is available to every unit it is just a matter of synchronization and coherence rather than communication. Shared memory systems need to care only about avoiding race conditions over the data, they have to make sure the calculations of unit are completed before other units access that information. The main limitation of the Shared Memory Systems is the poor scaling, to go beyond certain sizes they need to be arranged as nodes in a Distributed Memory Model.

Distributed Memory Systems have additional needs, they need to establish and manage the communication of the information across the different nodes to solve the problem. This means additional complexity and requires a careful design not only of the system itself but also in the algorithms to avoid the communications from becoming a bottleneck.

The GPU is in itself a Shared Memory parallel machine. In general a multi-GPU system would constitute a Distributed Memory System, although it must be indicated that latest developments in some of the GPU languages as CUDA can make a multi-GPU system appear to the programmer as a Shared memory one at the cost of some performance loss.

Nowadays a GPU (graphics processing unit) can be used together with a CPU to accelerate general-purpose scientific and engineering applications. Among the different existing procedure to perform parallel computing, CPU + GPU is a powerful combination because CPUs consist of a few cores optimized for serial processing, while GPUs consist of thousands of smaller, more efficient cores designed for parallel performance. In this section the principal aspects concern the General Purpose Graphic Processing Unit (GPGPU) for parallel computing are outline.

## 1.2. Massively parallel Computing in Graphic Processors.

Simply known as GPU Computing or GPGPU (General Purpose Graphic Processing Unit) is a relatively new field in the parallel programming landscape that uses graphic processors for its calculations.

The usual applications of GPGPU include all sort of physical simulations, image and video processing, biological and chemical analysis, applications that require many linear algebra operations, MRI image reconstruction, and all sorts of financial calculations.

The closer model of computers that existed so far was vector computers, although the differences between vectors computers and GPUs are substantial, the algorithms designed for vector computers can be easily translated into a GPU version.

In practice GPU Computing brings a supercomputer performance to low cost desktop computers, enabling thousands of researchers and scientist to run simulations that were out of the reach before for purely economic reasons. Additionally the programming models for GPU Computing are simpler to deal with, so the development costs have gone down as well as the infrastructure costs.

The first apparent big difference between GPU computing and "traditional" parallel computing is the huge amount of execution threads in GPU computing. While conventional parallel computing rarely goes beyond few hundred threads, GPU computing goes several orders of magnitude beyond, in some cases launching kernels with millions of threads, although in reality only few thousands will run concurrently at any given time.

Execution threads in a GPU can be considered to be "lightweight", their creation and management take few resources ("zero overhead scheduling") in contrast with their CPU equivalents.

The development of programming languages for GPU computing is very recent. In 2001 Nvidia Corporation develops the first GPU that can easily be programmed. Shortly after that Thompson et al (2002) write the first academic paper exploring the use of GPU as vector computers for arithmetic floating point calculations. Ian Buck et al (2004)[5] created Brook, the first GPGPU language, as an extension of C. Shortly after that the first proprietary language emerges, CUDA in 2006. OpenCL the main open language is released in 2008 by the Khronos Group.

### 1.3. GPU Languages. CUDA.

CUDA is probably the most advanced and developed language, currently in its 5.0 version with a pretty solid 5.5 version as release candidate. CUDA development cycle brings a new major version every year. It is currently accessible as language extensions to C/C++ (Nvidia) and Fortran (Portland Group), as well as the compiler-directive language OpenACC. Nevertheless there are bindings and tools to connect CUDA with almost any language.

The latest architecture for GPU computing with CUDA is the Kepler one, it is a 28nm process, the most powerful card on said architecture is the Tesla K20X, having 2688 stream processors (CUDA cores) grouped in 14 SMX (Streaming multiprocessors). Even when we are running our calculations on a modest Geforce GT640 with only 2 SMX (384 sps) the algorithm provides substantial speedups compared to the serial version.

The CUDA GPUs are able to work with three different levels of parallelism. At the bottom level we find the execution thread, which will be executed by a Streaming Processor (SP). Above the single thread we have the execution block, which consists of a group of threads whose number go from one to tens of thousands (although rarely is less than 32 threads), a Block will be executed by the SPs grouped in a Streaming Multiprocessor (SMX), in the latest architecture the SMX holds 192 SP, every SMX is capable of running several execution blocks concurrently if it has enough hardware resources to do so. Finally there is a third level of parallelism, although is not frequently used, which is the streams, each stream can schedule many GPU functions (kernels) even memory transfer operations to and from the CPU, which depending on the hardware resources may run concurrently, streams are especially useful in multi-GPU configurations usually in combination with MPI (message passing interface).

Compared to other GPGPU languages CUDA provides both simplicity and depth, it can provide a great control of the hardware without requiring too many lines of code as opposed to OpenCL. CUDA enjoys a strong support from Nvidia, in form of many libraries, very extensive documentation and code examples available for free, despite CUDA proprietary nature. In the present article we focus only in the first two levels of paralelism, leaving for the third level a mere indication of how to scale the problem for multi-GPU environments.

## 2. Finite Difference Method for the Heat Conduction Equation

### 2.1. Introduction

There are many physical phenomena that are able to be properly represented by partial differential equations (PDEs). For instant heat flow through metallic bar, evolution of water wave motion or the static deformation of an elastic bar.

In general, a PDE in several spatial variables $x,y,z$, and time $t$ is an equation $G$ involving an unknown function $u = u(x,y,z,t)$, sometimes called the state variable, and some of its partial derivatives

$$G\left(x,y,z,t,u,u_x,u_y,u_z,u_t,u_{xx},...\right) = 0, \quad (x,y,z) \in \Omega, \ t \in I \tag{1}$$

where $I$ is a given interval of time, often define as positive time $t \geq 0$, and $\Omega$ is a region in the space. A PDE model or an initial boundary value problem is a PDE supplemented with initial and boundary conditions that specify the state initially and on the boundary.

By a solution of the PDE (1) it is meant a function $u = u(x,y,z,t)$ defined on the space-time domain $(x,y,z) \in \Omega,\ t \in I$ that satisfies, upon substitution, the equation (1) identically on the domain.

In most practical cases coming from engineering practice PDEs are almost always solved numerically on a computer because most real world problems are too complicate to be solved analytically. By a numerical method we generally mean that the continuous PDE problem is replaced by a discrete problem that can be solved on a computer in finitely many steps. The result is a discrete solution known approximately at only finitely many points.

In this section we focus attention on the forward in time central in space (FTCS) finite difference method for the heat conduction equation. First we will set up a PDE model for the heat flow in three and two dimensions. As the heat equation describes a diffusive-like behaviour the finite difference procedure to solve the diffusion equation is outline subsequently.

## 2.2. Heat Flow in Three Dimensions

Many PDE models come from a basic balance law or conservation law. A conservation law, whose foundation goes back to Lavoisier's principal [2], is just a mathematical formulation of the basic fact that the rate at which a quantity changes in a given domain must equal the rate at which the quantity flows across the boundary plus the rate at which the quantity is created or destroy within the domain.

To set up a PDE model for the heat flow in three dimensions let $\Omega$ be a region in space where heat is flowing, and let $u = u(x,y,z,t)$ be the temperature at time $t$ at the point $(x,y,z)$ in $\Omega$. We assume that the region is homogeneous and is characterized by a constant specific heat $c$ and a constant density $\rho$. Now let *B* be an arbitrary sphere contained in $\Omega$ (Figure 1).

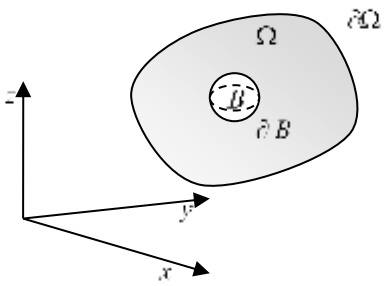


*Figure 1. Arbitrary sphere B contained in a region $\Omega$. $\partial B$ denotes the boundary of B*

We shall apply an energy balance principal to *B* that requires [1] that rate of change of the total amount of heat energy in *B* must equal the rate at which heat flows into *B* across its

boundary plus the rate at which heat energy is generated by any sources in *B*. The total amount of heat in a small volume element $dV = dxdydz$ is $c\rho u dV$, and thus the total amount of heat energy in $B$ is given by three-dimensional integral

$$\int_B c\rho u dV \tag{2}$$

We assume that the heat sources (or sinks) are given by a point function $f = f(x,y,z,t)$, where $fdV$ is the rate at which heat is generated in $dV$; thus, the rate at which heat is generated is the whole of *B* is

$$\int_B f dV \tag{3}$$

Next, we introduce the heat flux vector $\boldsymbol{\phi} = \boldsymbol{\phi}(x,y,z,t)$; its direction corresponds to the direction of heat flow at position $(x,y,z)$ at time $t$. The rate at which heat flows across a surface element $dA$ oriented by a unit vector $\mathbf{n}$ is $\boldsymbol{\phi}\cdot\mathbf{n}\,dA$ (Figure 2). Consequently, the net rate that heat energy flows across the boundary of *B*, denoted by $\partial B$, is the surface integral

$$\int_{\partial B} \boldsymbol{\phi}\cdot\mathbf{n}\,dA \tag{4}$$

So, the conservation law or equivalently the energy balance law [1] is

$$\frac{d}{dt}\int_B c\rho u dV = -\int_{\partial B} \boldsymbol{\phi}\cdot\mathbf{n}\,dA + \int_B f dV \tag{5}$$

Now making use of the divergence theorem, under sufficient differentiability requirement on the vector field $\boldsymbol{\phi}$, we can write (5) as

$$\frac{d}{dt}\int_B c\rho u dV = -\int_B \operatorname{div}(\boldsymbol{\phi}) dA + \int_B f dV \tag{6}$$

Bringing the time derivative under the integral on the left side and finally rearrange all the terms under one volume integral we get

$$\int_B \left(c\rho u_t + \operatorname{div}(\boldsymbol{\phi}) - f\right) dV = 0 \tag{7}$$

As this balance law must hold for every sphere *B* in $\Omega$, the integrand must vanish [1], giving the partial differential equation

$$c\rho u_t + \operatorname{div}(\boldsymbol{\phi}) = f \tag{8}$$

For all $t$ and all $(x,y,z) \in \Omega$. The equation (8) is the local form of the conservation law. It still contains two unknowns, the scalar temperature $u$ and the vector heat flux $\boldsymbol{\phi}$. A constitutive relation can be postulate to connect the two. One such relation is the three dimensional version of Fick's law, or Fourier's heat conduction law, which states, consistent with the laws of thermodynamics, that heat flows down the gradient, or in symbols,

$$\boldsymbol{\phi} = -K\operatorname{grad}(u) = -K(u_x, u_y, u_z) \tag{9}$$

where the proportionality constant $K$ is the thermal conductivity. Substituting (9) in (8) and using the identity $\mathrm{div}\left(\mathrm{grad}\left(u\right)\right) = u_{xx} + u_{yy} + u_{zz}$ we obtain the heat equation for the temperature $u = u\left(x,y,z,t\right)$ in three dimensions:

$$c\,\rho\,u_t - K\left(u_{xx} + u_{yy} + u_{zz}\right) = f \tag{10}$$

Introducing the Laplacian of $u$, denoted by $\Delta u$, we finally get

$$u_t - \kappa\Delta u = \frac{1}{c\,\rho} f \tag{11}$$

where the constant $\kappa = K/\left(c\,\rho\right)$ is called the diffusivity. If there are no sources, $f\left(x,y,z,t\right) = 0$, then $u = u\left(x,y,z,t\right)$ satisfies the standard heat equation

$$u_t = \kappa\Delta u \tag{12}$$

Of course, these calculations can be made in two dimensions in order to describe heat flow in the plane. The two-dimensional analogue of (11) has the same expression where now $u = u\left(x,y,t\right)$, $f = f\left(x,y,t\right)$, and $\Delta u = u_{yy} + u_{zz}$.

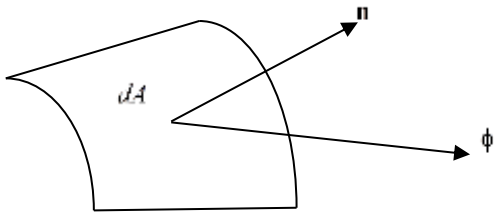


*Figure 2. Heat flux through a surface element dA oriented by its unit normal vector* $\mathbf{n}$

The auxiliary conditions to complete the PDE model for heat conduction includes an initial temperature condition

$$u\left(x,y,z,0\right) = u_0\left(x,y,z\right),\ \ \left(x,y,z\right) \in \Omega \tag{13}$$

and conditions on the boundary. There are two main types of boundary conditions that often occur in heat conduction problems. We can prescribe the temperature on the boundary of $\Omega$, that is,

$$u\left(x,y,z,t\right) = g\left(x,y,z,t\right),\ \ \left(x,y,z\right) \in \partial\Omega,\ \ t > 0 \tag{14}$$

where $g$ is given; this condition is called Dirichlet condition. We could also specify the heat flux on the boundary

$$-K\,\mathrm{grad}\left(u\right)\cdot\mathbf{n} = l\left(x,y,z,t\right),\ \ \left(x,y,z\right) \in \partial\Omega,\ \ t > 0 \tag{15}$$

where $l$ is given; this condition is called Neumann condition.

## 2.3. Forward in Time Central in Space Finite Difference Method

In order to illustrate the forward in time central in space (FTCS) finite difference method we applied it to the following one dimension diffusion equation problem

$$u_t = Du_{xx},\ 0 < x < 1,\ t > 0 \tag{16}$$

$$u(0,t) = u(1,t) = 0,\quad t > 0 \tag{17}$$

$$u(x,0) = f(x),\ 0 < x < 1 \tag{18}$$

The general idea of the finite difference method is to replace the partial derivatives in the equation by their difference-quotient approximations and then let the computer solve the resulting difference equation. The forward in time central in space scheme is a specific choice for the difference-quotient approximation [2,3].

The first step consist in discretize the region of space-time where we want to obtain a solution. In this case the region is $0 \le x \le 1,\ 0 \le t \le \mathrm{T}$. We have put a bound on time because in engineering practice we only solve a problem up until a finite time. Discretizing means defining a lattice of points, also known as nodes, in this space-time region by

$$x_j = jh,\quad t_n = nk,\quad j = 0,1,...,J;\ n = 0,1,...,N \tag{19}$$

where the fix numbers $h$ and $k$ are the spatial and temporal step sizes, respectively. Here, $h = 1/J$ while $k = T/N$. The integer $J$ is the number of subintervals in $0 \le x \le 1$ and the integer $N$ is the number of time steps to be taken (Figure 3). At each node $(x_j, t_n)$ of the lattice we seek an approximation, which we call $U_j^n$, to the exact value $u(x_j, t_n)$ of the solution. Note that the superscript $n$ refers to time, and the subscript $j$ refers to space. We regard $U_j^n$ as two dimension array, or matrix, where $n$ is a row index and $j$ is a column index.

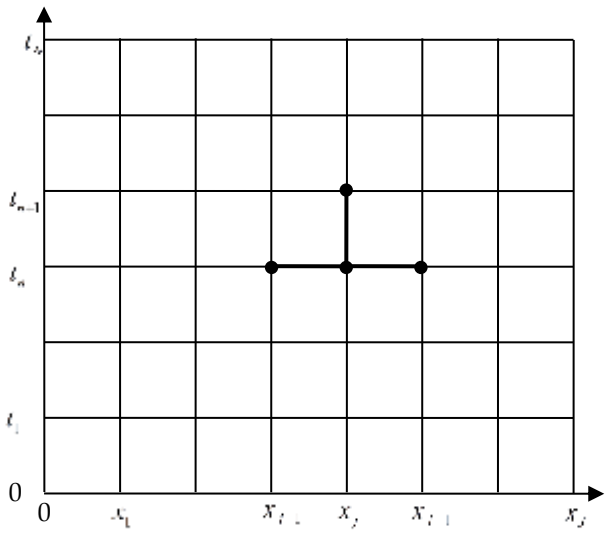


*Figure 3. Discrete lattice and nodes involved in the FTCS finite difference method*

To obtain equations for the $U_j^n$, we replace the partial derivatives in the PDE by their difference approximation. From calculus, using the Taylor series expansion in $t$, we might get [1, 2] the difference quotient

$$u_t\left(x_j,t_n\right)\approx\frac{u\left(x_j,t_{n+1}\right)-u\left(x_j,t_n\right)}{k} \tag{20}$$

And by the Taylor series expansion in $x$,

$$u_{xx}\left(x_j,t_n\right)\approx\frac{u\left(x_{j-1},t_n\right)-2u\left(x_j,t_n\right)+u\left(x_{j+1},t_n\right)}{h^2} \tag{21}$$

The first of these formulas is the usual forward in time difference approximation for a first derivative, and the second is central in space difference approximation for a second derivative. Making use of (20) and (21), the PDE (16) at the point $\left(x_j,t_n\right)$ is replace by the difference equation

$$\frac{U_j^{n+1}-U_j^n}{k}=D\frac{U_{j-1}^n-2U_j^n+U_{j+1}^n}{h^2} \tag{22}$$

Or, upon solving for $U_j^{n+1}$,

$$U_j^{n+1}=U_j^n+\frac{Dk}{h^2}\left(U_{j-1}^n-2U_j^n+U_{j+1}^n\right) \tag{23}$$

The difference equation (23) gives the approximate solution at the node $\left(x_j,t_{n+1}\right)$ in terms of approximations at the three earlier nodes $\left(x_{j-1},t_n\right)$, $\left(x_j,t_n\right)$, $\left(x_{j+1},t_n\right)$, see Figure 3. These four points geometric arrangement form the stencil for the FTCS finite difference method in one dimension. Now we see how to fill up the lattice approximate values. We know the values at $t=0$ from the initial condition (18). That is

$$U_j^0=f\left(x_j\right),\ j=0,1,...,J \tag{24}$$

We can observe that if $j=0$ then $U_{-1}^n$ appears in equation (23) and for $j=J$ then $U_{J+1}^n$ also enters in equation (23). However, following (19) there are no nodes place at $j=-1$ neither at $j=J+1$. To solve this drawback [3] we consider the boundary conditions (17), namely we only consider equations in (23) with subscript $j$ running from 1 to $J-1$ while

$$U_0^n=0,\ \ U_J^n=0,\ n=1,2,...,N \tag{25}$$

The difference formula (23) with $j=1,...,J-1$ can now be applied at all the interior lattice points, beginning with the values at the $t=0$ level, to compute the values at the $t=t_1$ level, then using those to compute the values at the $t=t_2$ level, and so on. Therefore, using the difference equation (23), we can march forward in time, continually updating the previous temperature profile.

A finite difference scheme like (23) is called explicit scheme because it permits the explicit calculation of an approximation at next time step in terms of values at a previous time step. Because of the error, called truncation error, that is present in (23) due to replacing derivatives by differences, the scheme is not always accurate. If the time step $k$ is too large, then highly

inaccurate results will be obtained. It can be shown [3] that for a one dimension problem we must fulfil the stability condition

$$\frac{kD}{h^2} \leq \frac{1}{2} \tag{26}$$

for the scheme to work; this condition limits the time step. This is why the forward scheme (23) is known to be conditionally stable. To obtain a method that is unconditionally stable we should consider an implicit difference method. However, these schemes involved the solution of a linear system to advance in time. The solution of algebraic linear systems scales poorly with the problem size, usually as an $O(n^3)$ problem, therefore in many cases it is often avoided when the objective is performance on large problems. From now on we will focus on the forward in time central in space scheme (23).

## 3. Case of application. Heat diffiusion in two dimensions

The case of application consistes in the followin initial boundary value problem:

$$u_t = \frac{1}{16}\left(u_{yy} + u_{zz}\right) \tag{27}$$

$$u(0,y,t) = u(1,y,t) = 0, \quad u(x,0,t) = u(x,1,t) = 0, \quad t > 0 \tag{28}$$

$$u(x,y,0) = sin(2\pi x)sin(2\pi y), \quad (x,y) \in \Omega \tag{29}$$

Where $\Omega$ is the unit square in *xy*-plane. By the second section of the present work, this PDE model might be regarded as a heat conduction problem in two dimensions where the diffusivity has been set to $1/16$ with no heat source. Homogeneous Dirichlet boundary condition has been considered while initial temperature is given by(29).

After applying separation of variables method [4] to the initial boundary value problem define by (27)-(29) we get the following analytical solution

$$u(x,y,t) = e^{-0.5\pi^2 t} sin(2\pi x)sin(2\pi y) \tag{30}$$

The same PDE model is numerically solved by the forward in time central in space (FTCS) finite difference method up until $T = 1\,\text{s}$. To make a significant comparison between serial and different parallel computing implementations, the number $J$ of subintervals for the spatial discretization, equal for *x* and *y* directions, varies in powers of two from $32$ to $512$. The number of time steps is set to $N = 100000$. Regarding the stability condition for two dimension case, being $h = 1/J$ and $k = T/N$, we get $\frac{k\alpha}{h^2} < \frac{1}{4}$. Therefore stable numerical simulations are performed for each value of $J$ [3].

The FTCS finite difference method in two dimensions is associated with a five-point stencil. Therefore, calculation of every grid node value at time $t_{n+1}$ requires reading the values of its four neighbour nodes at time $t_n$, $(x_i, y_{j-1}, t_n)$, $(x_i, y_{j+1}, t_n)$, $(x_{i-1}, y_j, t_n)$, $(x_{i+1}, y_j, t_n)$ and the value at time $t_n$ of the node itself $(x_i, y_j, t_n)$, as it is shown in Figure 4.

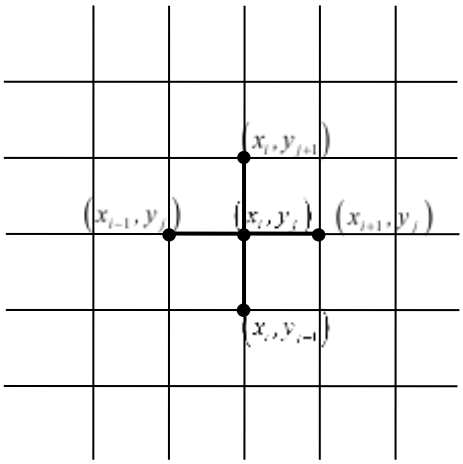


*Figure 4. FTCS finite difference stencil scheme in two dimensions*

Bearing in mind this five-point stencil structure, a serial pseudo code solution would contain code like:

**Algorithm1** Simulation Loop

1: **for all** spatial step *i* in *x* direction **do**
2: **for all** spatial step *j* in *y* direction **do**
3: read node value *U* in position $(x_i, y_j)$ at time step $t_n$
4: read neighbour node values *U* for $(x_i, y_{j-1})$, $(x_i, y_{j+1})$ $(x_{i-1}, y_j)$, $(x_{i+1}, y_j)$ at time step $t_n$
5: calculate node value *U* for $(x_i, y_j)$ at time step $t_{n+1}$
6: write result into memory.
7: **end for**
8: **end for**
9: **for all** nodes $(x_i, y_j)$ that belong to boundary
10: apply boundary conditions at time step $t_{n+1}$
11: **end for**

For a naive GPU implementation the pseudo code solution would contain code like:

**Algorithm2** Simulation Loop

The kernel is launched with one thread per grid node $(x_i, y_j)$

1: read node value $U$ in position $(x_i, y_j)$

2: read neighbour node values $U$ for $(x_i, y_{j-1})$, $(x_i, y_{j+1})$, $(x_{i-1}, y_j)$ $(x_{i+1}, y_j)$ at time step $t_n$

3: calculate node value $U$ for $(x_i, y_j)$ at time step $t_{n+1}$

4: write result into memory.

Another kernel is launched with one thread per boundary node

5: apply boundary conditions at time step $t_{n+1}$

As we can see the most aparent change in the pseudo code comes from the fact that instead of having a nested loop to traverse the grid in the $x$ and $y$ directions, in the parallel version we simply launch a kernel (GPU subroutine) with as many threads as grid nodes. Although not all the threads run concurrently, high level of paralelism is achieved. Moreover the GPU will schedule and execute the threads in a manner that while some threads are waiting for the values they need from memory, other threads are being executed, this is called "latency hidding" wich is another reasons GPUs can run so fast calculations. Effective latency hidding requires launching more threads than Streaming Processors are available at the GPU. The particular tunning of the parameters like threads per block and total number of threads must be adapted for every particular algorithm.

We can improve over the naive version when we consider the second level of parallelism as we can use the on-chip shared memory that every SMX has. In the second version we start using better the GPU hardware, in particular as we are working at the SMX level we can employ the on-chip shared memory.

Shared memory presents one of the best opportunities to obtain a large increase in performance if the algorithm can work within the restriction imposed by its limited size.

Fetching values from shared memory is much faster than reading global memory. With shared memory we have a latency of around 8 cycles, while uncached global memory has a latency of up to 600 cycles. That is almost two orders of magnitude.

The main obstacle in the use of shared memory is its limited size. In the best of cases we have available around 48 Kb of shared memory per SMX which means around 6.000 values in double precision or 12.000 in single precision.

The benefit of using shared memory is better understood if we consider that in the case of this stencil operation we can reduce the memory bandwidth in use to a fifth of the naive implementation.

Any GPU subroutine can be classified from a performance point in two rough groups. "Memory Bandwidth Bound" or simply Memory-Bound are algorithms whose performance is limited by the latencies of memory Access, that is the number of floating point operations they execute cannot compensate for the time each thread has to wait to fetch the values is using, in other words a faster clock rate in the GPU would not yield the result any faster.

"Computationally Bound" kernels in the other hand, have enough computations so the clock rate in the SP is the limiting factor, in this case increasing the clock frequency would result in the increase of performance. A given implementation can be either memory or computationally bound, as the computation threshold is usually several times higher that would be ideally the limit we would like to reach.

Most implementations, specially the naive ones, are "Memory-Bound". The main reason for that is how the hardware is built. As the computational power of a chip depends of the number of transistors in the chip that means is directly proportional to the "effective computational area" of the chip, while the bandwidth is proportional to the perimeter of the chip. This may change in the future with the advent of 3D-stack chips, but for the time being is the main challenge to current chip architectures.

So our second implementation tries to reduce the amount of times we need to fetch values from global memory, as it can be seen that we are reading five times every value from global memory, one for the calculation of the node himself and another four for the calculation of its neighbours.

The new implementation is now divided in two fundamental phases, the first one consist of loading the node value into shared memory, the second one consists in reading the necessary values from shared memory to perform the calculation.

There are two very important aspects to consider in this implementation. First given the limitation of shared memory size we have to divide the grid into "tiles" that fit shared memory, this also requires some overlap between the tiles so all the threads have access to all the necessary values. This introduces a small redundancy as the "halo" threads are only used for memory loads and not for calculation.

The second aspects is a mere practical one, we need to introduce a synchronization barrier at block level. We need to make sure that all the threads belonging to the same block have finished loading their node values into shared memory before we make the calculation.

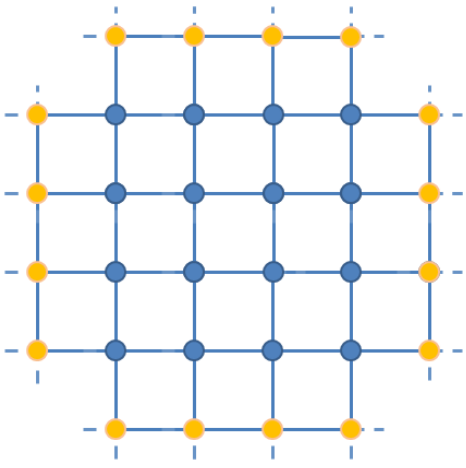

*Figure 5. Simplified tile division of the grid for each SMX. Calculation nodes are in blue while "halo" nodes are yellow.*

**Algorithm3** Simulation Loop

The kernel is launched with as many halo-overlapping blocks as necessary.

One thread per grid node $(x_i, y_j)$ in the block + enough threads for the "halo" nodes.

1: read node value $U$ in position $(x_i, y_j)$

2: store node value in shared memory

3: **synchronization barrier** at block level

4: read neighbour node values $U$ for $(x_i, y_{j-1})$, $(x_i, y_{j+1})$, $(x_{i-1}, y_j)$, $(x_{i+1}, y_j)$ at time step $t_n$

5: calculate node value $U$ for $(x_i, y_j)$ at time step $t_{n+1}$

6: **if** node is not in halo **do**

7: write result into global memory

8: **end if**

Another kernel is launched with one thread per boundary node

9: apply boundary conditions at time step $t_{n+1}$

## 4. Results and Discussion

In Figure 6 the numerical temperature profile for $t = 0,\ 0.25,\ 0.5,\ 1\ s$ obtained with the FTCS for $J = 32$ and $N = 1000$ is presented. We observed how the maximum and minimum temperatures, attained at time $t = 0$, are $u = 1$ and $u = -1$, respectively. As the time integration proceeds this temperature profile tends to dump out, leading to an almost zero temperature in the whole domain at time $t = 1s$.

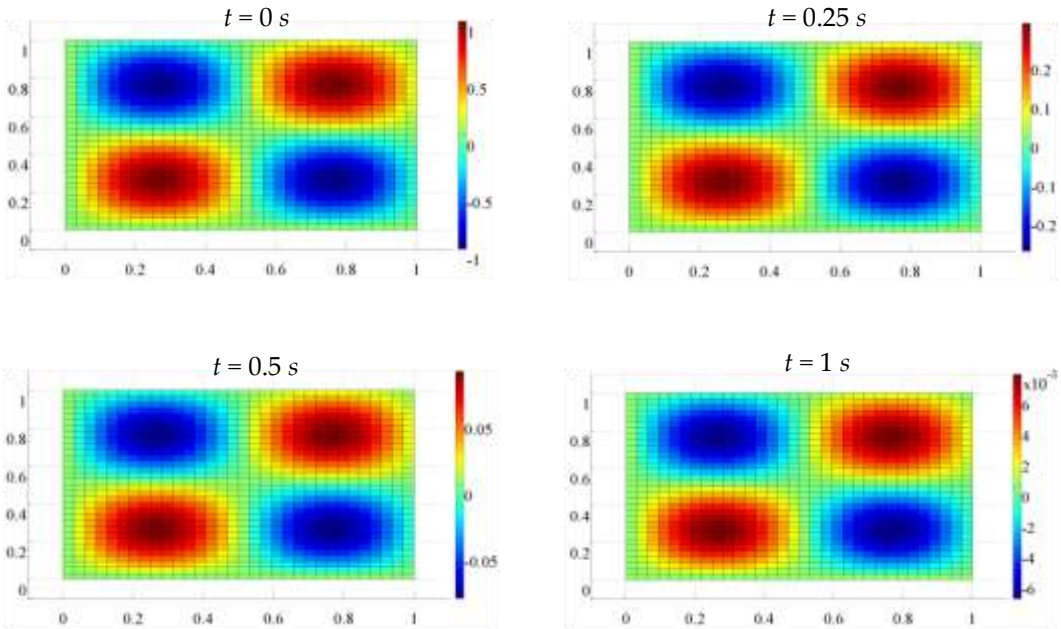


*Figure 6. Temperature profile for times* $t = 0,\ 0.25,\ 0.5,\ 1\ s$

In order to verify that numerical solutions are reproducing adequately the heat conduction phenomena, the approximate solution achieve with FTCS scheme is compared with exact solution (30). To this end the error profile for $t = 0,\ 0.25,\ 0.5,\ 1\ s$, measure as the absolute value of the difference of the approximate and the exact solution is displayed in Figure 7 for $J = 32$ and $N = 1000$

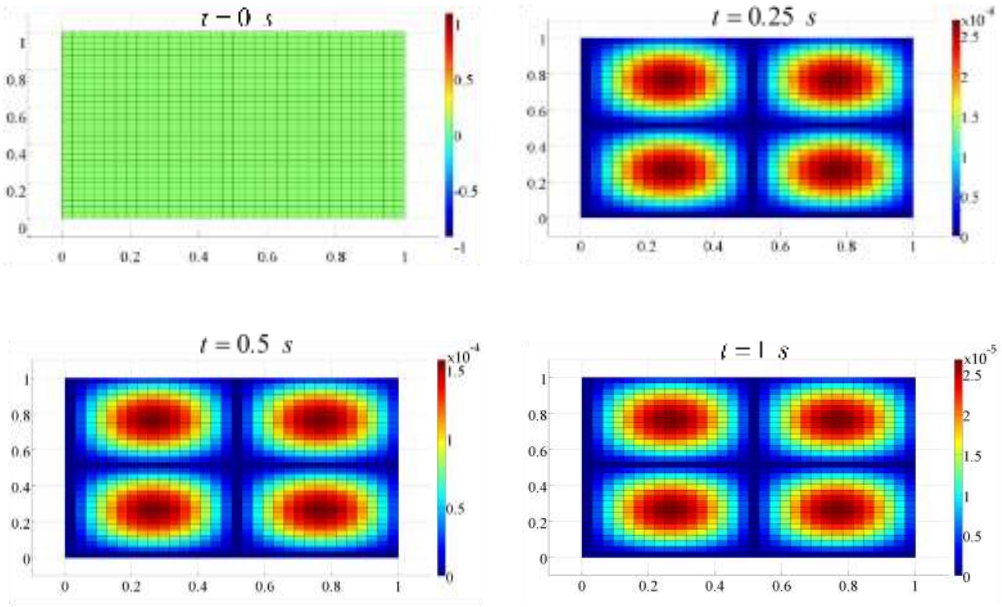


*Figure 7. Absolute error profiles for times* $t = 0,\ 0.25,\ 0.5,\ 1\ s$

In this last figure it is clear that numerical and analytical solutions are almost coincident. Obviously at initial time no difference exists. As the time integration proceeds, some error appears in the numerical solution, being higher at nodes (0.25 , 0.25), (0.25 , 0.75), (0.75 , 0.25), (0.75 , 0.75), which are the nodes where maximum and minimum temperature is attained.

Once it has been shown numerical solutions reproduce correctly the heat conduction problem in 2D, running time comparisons for the different implementations are analyzed subsequently.

We decided to run the parallel version in the last three GPU architectures developed by Nvidia. These are, Tesla, Fermi and Kepler which is the latest one. Although we decide to report only the two most diverse cards the Quadro FX3800 and the Geforce GT640, as the results for the GT420 card didn't add any interesting insights that were not provided by the other two (Table 1)

*Table 1.Hardware used in the comparison.*

| Processor | Arquitecture Name | Chipset | Process | Clock Mhz | Memory Band. Gb/sec | SMX | SP | Power Draw (W) |
|---|---|---|---|---|---|---|---|---|
| Intel Xeon | Bloomfield | W3520 | 45 nm | 2660 | | | | 130 |
| Quadro FX3800 | Tesla | GT200 | 55 nm | 600 | 51.2 | 24 | 192 | 108 |
| Geforce GT640 | Kepler | GK107 | 28 nm | 728 | 28.5 | 2 | 384 | 50 |

We also provide the speedup per streaming multiprocessor to account the difference between each card, as the Tesla one was a high end card at the time of its release (Quadro FX3800) while the other two are low-end cards (Geforce GT420 and GT640).

In Figure 8 we can observed the running time in milliseconds for different number of subintervals for the spatial discretization and the different implementations considered. It is clear that an important running time improvement is achieved when parallel instead of serial implementation is considered.

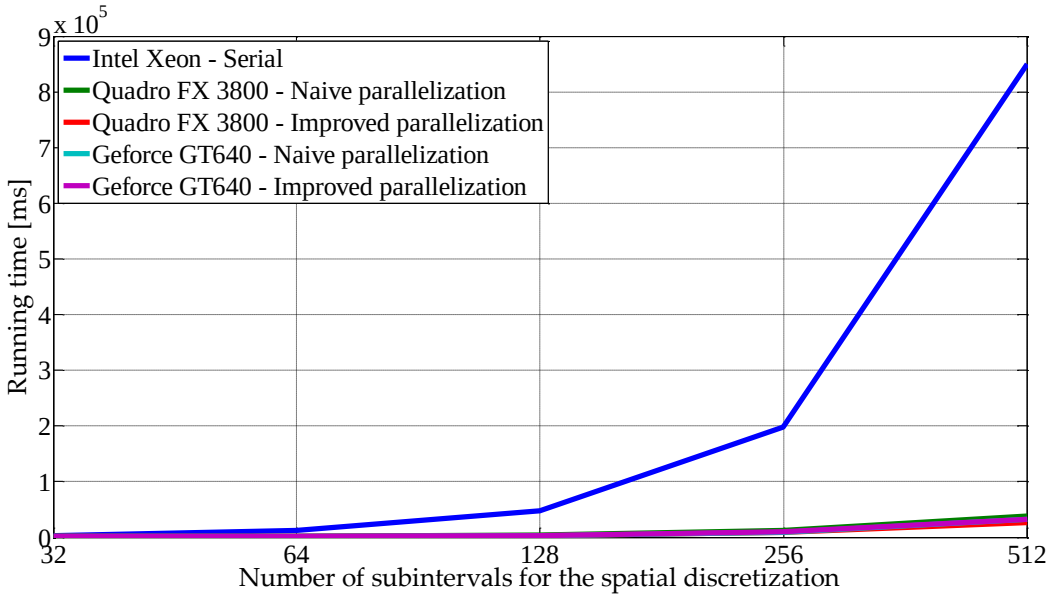


*Figure 8. Running times in milliseconds.*

A different way to cluster data of Figure 8 is to consider the relative speed up between serial and parallel computations. This is shown in Table 2. Again, it is clear the important speed up achieve when parallel instead of serial implementation is considered.

*Table 2. Speed up against serial version*

| Algorithm | Processor | 32x32 | 64x64 | 128x128 | 256x256 | 512x512 |
|---|---|---|---|---|---|---|
| Naive parallel. | FX3800 | 1.4x | 5.2x | 11.8x | 16.8x | 22.5x |
| Improved parallel. | FX3800 | 1.7x | 6.0x | 13.8x | 22.0x | 31.7x |
| Naive parallel. | GT640 | 2.0x | 7.1x | 16.9x | 21.5x | 26.7x |
| Improved parallel. | GT640 | 2.0x | 7.3x | 17.1x | 20.9x | 26.3x |

From Table 2 it can be observed how time scales with the grid size. In principle this is an $O(n^2)$ problem, meaning the execution time (per time step) should scale roughly with the square of the grid size (spatial steps), so a 64 x 64 grid should take four times as much as the 32 x 32 grid. This is observed in the serial version of the algorithm, with some irregularities maybe caused by cache sizes.

In the GPU implementations we see a better scaling up to sizes of 512 x 512. This can be observed in Table 3. In the case of the smaller grids it seems to scale almost free of cost up to 128 x 128. The reason for this advantage has to do with the way GPUs operate. To hide latencies properly the GPU needs grids with a lot of blocks and threads. In the small problems we only launch few blocks as the block size is 256 treads organized as 16 x 16 tiles. That means the 32 x 32 problem only gets four blocks in the naive implementation. As the size of the problem continues to increase we use all of the scheduling advantage of the GPU and execution times and problem size scale at the expected rate.

*Table 3. Scaling of execution times per time step compared to the previous problem size.*

| Algorithm | Processor | 32x32 | 64x64 | 128x128 | 256x256 | 512x512 |
|---|---|---|---|---|---|---|
| Serial | Xeon | | 4.00 | 4.00 | 4.13 | 4.29 |
| Naive parallel. | FX3800 | | 1.12 | 1.75 | 2.89 | 3.21 |
| Improved parallel. | FX3800 | | 1.12 | 1.73 | 2.59 | 2.98 |
| Naive parallel. | GT640 | | 1.11 | 1.68 | 3.25 | 3.46 |
| Improved parallel. | GT640 | | 1.10 | 1.70 | 3.37 | 3.42 |

If we analyze the results only considering parallel implementation, we observed that the improved version performs slightly worse than the naive implementation for the Fermi and

Kepler architectures (Figure 9). The reason is very simple. In this algorithm the use of shared memory is limited while Kepler and Fermi include caches which in practice is the same memory used as shared. So the naive implementation is in practice using the same memory access but launching a bit less blocks and threads. Additionally the naive implementation suffers no delay from the synchonization barriers.

That is not to say the improve algorithm is worse, whith a different memory access pattern (a differente type of stencil) or being the problem a 3D instead of a 2D, we would see much better results. If shared memory had to hold intermediate values like in reductions algorithms we would see greater performance improvements using the shared memory

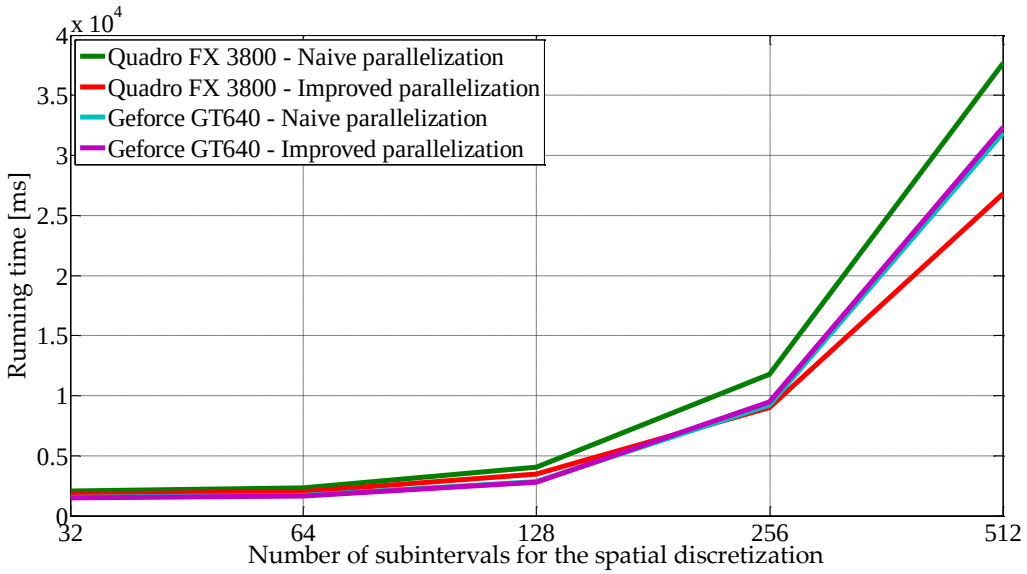


*Figure 9. Running times in milliseconds for all the GPU implementations.*

Another good comparison is the speedup per Streaming Multiprocessor (Table 4). The number of SMX seems to be more or less constant between architectures, with high end cards having between 13 to 15, or even twice that if they are dual cards. In this case we can clearly see that the kepler architecture achieves more than 27x speedup per SMX over the serial version, so even if we don't account for the bandwith improvement, which is also a big difference between low end and high end cards, we could see speedups exceding the 380x with the high end cards of the Kepler architecture such as the Geforce Titan or the K20 cards.

*Table 4. speed up per SMX against serial version*

| Algorithm | Processor | 32x32 | 64x64 | 128x128 | 256x256 | 512x512 |
|---|---|---|---|---|---|---|
| Naive parallel. | FX3800 | 0.1x | 0.2x | 0.5x | 0.7x | 0.9x |
| Improved parallel. | FX3800 | 0.1x | 0.2x | 0.6x | 0.9x | 1.3x |
| Naive parallel. | GT640 | 1.0x | 3.6x | 8.5x | 10.8x | 13.3x |
| Improved parallel. | GT640 | 1.0x | 3.6x | 8.5x | 10.4x | 13.1x |

Finally if we adjust the speedup with the ratio of power draw between the CPU and the GPU we obtain a table (Table 5) that brings the attention to the energy efficiency of the new architectures of GPUs. The performance adjusted for energy consumption is simply fantastic, these cards can be the building block of low cost, low energy cosumption clusters, there are already hardware solutions like the Carma and Kayla development kits that point into this direction. This is from our point of view one of the most interesting results, very good speedups with very cheap hardware at very low operation costs.

*Table 5. Speedup adjusted for energy consumption.*

| Algorithm | Processor | 32x32 | 64x64 | 128x128 | 256x256 | 512x512 |
|---|---|---|---|---|---|---|
| Naive parallel. | FX3800 | 1.7 | 6.2 | 14.2 | 20.2 | 27.1 |
| Improved parallel. | FX3800 | 2.0 | 7.2 | 16.6 | 26.4 | 38.0 |
| Naive parallel. | GT640 | 5.1 | 18.5 | 44.0 | 55.9 | 69.3 |
| Improved parallel. | GT640 | 5.2 | 18.9 | 44.4 | 54.3 | 68.3 |

## 5. Concluding remarks

A PDE model of a heat conduction problem in two dimensions has been numerically solve by the forward in time central in space (FTCS) finite difference method. Three different implementations have been developed one under serial computing and the other two under parallel procedure. The parallel implementations have been run over Quadro FX3800 and Geforce GT640 GPU architectures.

Emphasis has been devoted to GPU parallel computing with CUDA language. This combination of architecture and language is outstanding. It permits a huge amount of execution threads, compared to conventional parallel computing, while not many lines of code are required to achieve a great control of the hardware due to CUDA language.

The analysis of the results shows the importance of parallel computing when dealing with large problems that are impractical or impossible to solve them with a serial computing procedure as the execution times grow so fast.

Additionally we can see the advantages the new architectures of GPUs. On one side inclusion of caches has improved the performance of the simpler implementations. On the other side the increase of performance from one architecture to the next is remarkable, but the performance adjusted for energy consumption is even more impressive.

# 6. References

[1] LOGAN JD, *Applied Partial Differential Equations*, Springer, New York, 2004.

[2] RAPPAZ M, BELLET M, DEVILLE M, *Numerical Modeling in Materials Science and Engineering*. Springer, New York, 2003.

[3] BURDEN RL, FAIRES JD, *Numerical Analysis*, Brooks/Cole-Thomson Learning, 2001.

[4] JOHN F, *Partial Differential Equations*, Springer, New York, 1982.

[5] I. BUCK, T. FOLEY, D. HORN, J. SUGERMAN, K. FATAHALIAN, M. HOUSTON, AND P. HANRAHAN. *Brook for GPUs: Stream Computing on Graphics Hardware*, páginas específicas consultadas, Stanford University, ACM Trans. Graph. (SIGGRAPH), 2004.

[6] BRODTKORB A, DYKEN C, HAGEN T, HJELMERVIK J, STORAASLI O. *State-of-the-art in heterogeneous computing. Scientific Programming Volume 18 Issue 1, Amsterdam 2010.*

[7] HARRIS MARK. *Finite difference Methods in CUDA C/C++.* https://developer.nvidia.com/content/finite-difference-methods-cuda-cc-part-1

**Sobre los autores:**

*Nombre:* Vicente Cuellar Moro
*Correo Electrónico:* vicente.cuellar.moro@gmail.com
*Institución:* M2i Group, ETS Ingenieros de Caminos, Universidad Politécnica de Madrid, España.

*Nombre:* Miguel Martín Stickle
*Correo Electrónico:* miguel.martins@upm.es
*Institución:* M2i Group, ETS Ingenieros de Caminos, Universidad Politécnica de Madrid, España.

*Nombre:* Manuel Pastor Pérez
*Correo Electrónico:* manuel.pastor@upm.es
*Institución:* M2i Group, ETS Ingenieros de Caminos, Universidad Politécnica de Madrid, España.