Copyright © 2009 Tech Science Press CMES, vol.51, no.2, pp.143-167, 2009

An Implementation of the Longman's Integration Method on Graphics Hardware

E. Mesquita[1], **J. Labaki**[1] **and L. O. S. Ferreira**[1]

Abstract: There is a growing trend towards solving problems of computational mechanics by parallelization strategies. The traditional approach is to implement the parallelization procedures on CPUs based on the MPI or OpenMP paradigms. Recent efforts have been made to implement computational tasks on general-purpose programmable graphics hardware (GPGPU). The GPU is specially well-suited to address problems that can be formulated in form of data-parallel computations with high arithmetic intensity. This work addresses the implementation of the Longman's integration method on graphics hardware. A serial implementation of Longman's method was rewritten under the SIMD (Single Input Multiple Data) parallel programming paradigm. The code was developed on an NVidia™ CUDA programming environment and executed on a graphics card hosted by a regular dual-cored CPU. The structure of a GPU as visible from the CUDA programming language is briefly described in order to assess the possible strategies for parallel implementation on the graphics card. The accuracy and efficiency of the implemented strategies are investigated by solving the improper integral of a simple, but representative, oscillatory and decaying function possessing closed-form solution. The paper reports the performances of the GPU and the CPU on solving different numbers of integrals for distinct parameters of the integrand and required degrees of accuracy. For a large number of integrals the GPU has shown a speedup capacity ranging from one to two order of magnitudes compared to the CPU.

Keywords: High Performance Computing, Graphics Hardware, Improper Numerical Integration, Oscillatory-Decaying Functions, Numerical Inversion of Integral Transforms.

1 Introduction

In the last years, the edges of computing capability have been pushed by the emergence of General Purpose Graphics Processing Units (GPGPU). These graphic

[1] University of Campinas, SP, Brazil

cards have been applied to Finite Difference Methods, Particle Based Methods, Lattice Boltzmann Method and also to the Finite Element Method [Oishi and Yoshimura (2008)]. The articles of Göddeke and his collaborators explored GPUs as scientific co-processors within the context of mixed precision iterative refinement techniques applied to the solution of PEDs [Göddeke et ali. (2007a)], analyzing performance, cost and power demands on heterogeneous clusters [Göddeke et ali. (2008)] as well as weak scalability for FEM calculations [Göddeke et. ali. (2007b)]. In the implementations reported in the articles cited above, the GPU was controlled by the so-called graphics APIs (Application Programming Interfaces), like OpenGL [Shreiner, Woo, Neider and Davies (2005)] or DirectX [Jones (2004)]. Around the end of 2006, a new technology of graphic devices was launched. This new generation of devices is not only dedicated to graphics computation, but it is also capable of performing general-purpose data processing. Along with this technology a new API called CUDA (Compute Unified Device Architecture) has been launched by the NVidia™ Corporation for this new generation of GPUs [CUDA (2008)]. CUDA is a programming language that allows the programmer to code the GPGPU in a higher level paradigm, compared to the former graphics-dedicated APIs such as OpenGL and DirectX. [Owens, Luebke, Gobindaraju, Harris, Krüger, Lefohn and Purcel (2007); CUDA (2008)]. Recently the GPUs under the CUDA programming environment has been applied to solve the Boundary Element Method (BEM) [Takahashi and Hamada (2009)].

Graphics hardware was born as a parallel computation device. Its high-bandwidth memories and its floating-point operations are significantly faster than ordinary CPUs and have called attention of the scientific community. Large scale computational tasks, whose parallel formulations have been explored for CPU clusters [Araujo and Gray (2008)], now find in general-purpose GPU a new and promising alternative of implementation.

The need for calculation of numerical integrations over unbounded limits is a frequent task in the solution of physical problems. One such integral is given in Eq. 1, where J_0(x) is the Bessel function of first kind and order zero [Korenev (2002), Longman (1956)],

$$I_{Bessel} = \int_0^\infty J_0(x)\,dx = 1 \tag{1}$$

Likewise a large class of problems of mathematical physics may be solved with the use of integral transforms [Selvadurai (2000)]. Usually, the first integral transformation from the original physical to the transformed domain is performed analytically. The inverse transformation, back to the physical domain, however, is generally accomplished numerically. An example of a Hankel integral transform is given in Eq. 2. In this equation J_ν is the Bessel function of the first kind and order

v. The variables in the original and transformed domains are, respectively, x and ξ. The function to be transformed to the original domain is f(x). This improper integration must be evaluated repeatedly for all desired values of the variable ξ.

$$F_{Hankel}(\xi) = \int_0^\infty x f(x) J_v(\xi x)\, dx \tag{2}$$

Double Fourier integral transforms also represent examples of improper integrations that must be determined a large number of times. Equation 3 shows a double Fourier inverse transformation that must be numerically determined over the function f(x, y)

$$F_{Fourier^2}(\xi,\eta) = \frac{1}{2\pi}\int_{-\infty}^{\infty}\left\{\int_{-\infty}^{\infty} f(x,y)\, e^{i\xi x}\, dx\right\} e^{i\eta y}\, dy = \\ = \frac{1}{2\pi}\int_{-\infty}^{\infty}\{I(\xi,y)\}\, e^{i\eta y}\, dy \tag{3}$$

In Eq. 3 the inner integral over the variable x in must be evaluated for every value of the variable y at the outer integral. The operations described so far are numerically very intensive and tend to impose a limitation in the application of integrals transforms that must be evaluated numerically.

The integrand of these numerical inversions may present an oscillating and decaying behavior. This is the case of function f(x) shown in Eq. 1. For the improper integration of decaying and oscillating functions, Longman (1956) proposed a very efficient method. This work addresses the implementation of the Longman's integration method on graphics hardware within the CUDA programming environment. A relative simple, but representative, function with known closed form solution is used to exemplify the potentiality of the proposed implementation.

The paper begins describing the method of integration proposed by Longman. The classical serial implementation is overviewed. Next, the new technology of GPGPU is described in some detail. The structure of a GPU, as visible from the CUDA programming language, is briefly described in order to formulate the possible strategies for parallel implementation on the graphics card. It is shown why the GPU implementation is more efficient than its CPU counterpart for the present application and how the coding of non-graphical algorithms is treated. The fourth section shows how the method of integration was approached in order to comply with the GPGPU stream computing philosophy. Finally, the presented implementation is applied to solve a large number of integrations. Its performance is compared with an ordinary CPU serial code.

2 Longman's Integration Method

Longman (1956) proposed a method of numerical integration to treat improper integrals of the kind:

$$I = \int_a^{\infty} f(x)\,dx \tag{4}$$

in which a is a constant and f(x) oscillates around zero in such a manner that the absolute magnitude of the integral over each half-cycle is smaller than (and opposite in sign to) that over the preceding half-cycle. An example of such a function is shown in Eq. 5, the behavior of which is depicted by Fig. 1 for the parameters λ = 0.5 and ω = 10.

$$f(x) = e^{-\lambda x}\cos(\omega x) \tag{5}$$

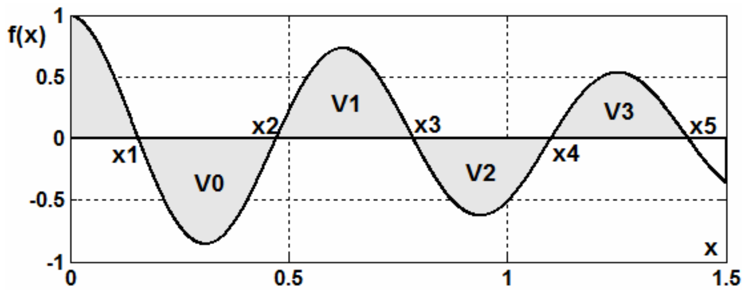

Figure 1: Curve of $f(x) = \exp(-1/2x)\cos(10x)$ and illustration of the slowly decreasing areas.

Longman's method is based on Euler's transformation of slowly convergent alternating series [Bromwich (1942)]. Consider the following series,

$$V_0 - V_1 + V_2 - V_3 + V_4 - \cdots \tag{6}$$

where

$$V_n > 0, \quad V_{n+1} < V_n, \quad \forall n \in \mathrm{N} \tag{7}$$

The differences Δ^{r+1} of order r+1 of area n are defined as follows:

$$\Delta^{r+1} V_n = \Delta^r V_{n+1} - \Delta^r V_n, \tag{8}$$

with

$$\Delta^{r=0}V_n = \Delta^1 V_n = V_{n+1} - V_n, \quad \Delta^{r=-1}V_n = V_n$$

Using Eq. 8, the sum of the infinite series can be expressed as [Longman (1956)]:

$$\sum_0^\infty (-1)^n V_n = \tfrac{1}{2}V_0 - \tfrac{1}{4}\Delta V_0 + \tfrac{1}{8}\Delta^2 V_0 - \cdots \tag{9}$$

The series on the right-hand side of Eq. 9 can be shown to be convergent whatever the original series is [Kaplan (2002); Longman (1956)]. A remainder R_N of this sum after N terms can be estimated [Bromwich (1942)]:

$$|R_N| \leq 2^{-N} \left|\Delta^N V_0\right| \tag{10}$$

The terms V_n shown in the preceding equations can be related to the areas of the curve of $f(x)$, as it was depicted in Fig. 1. In addition, the requirements stated about the behavior of the function $f(x)$ causes it to have infinite roots bigger than a at the abscises $x = x_i$, (i = 1, 2, 3, ...), with $a \leq x_1 < x_2 < x_3$, etc. (Fig. 1). The integration of Eq. 4 may be expressed in terms of these areas:

$$\int_a^\infty f(x)\,dx = \int_a^{x_1} f(x)\,dx + \int_{x_1}^{x_2} f(x)\,dx + \int_{x_2}^{x_3} f(x)\,dx + \cdots \tag{11}$$

or further,

$$\begin{aligned} \int_a^\infty f(x)\,dx &= \int_a^{x_1} f(x)\,dx - \int_{x_1}^{x_2} \{-f(x)\}\,dx + \int_{x_2}^{x_3} f(x)\,dx + \cdots \\ &= \int_a^{x_1} f(x)\,dx - (V_0 - V_1 + V_2 - V_3 + V_4 - \cdots) \end{aligned} \tag{12}$$

Finally, according to the transformation expressed by Eq. 9,

$$\int_a^\infty f(x)\,dx = \int_a^{x_1} f(x)\,dx + \left(\tfrac{1}{2}V_0 - \tfrac{1}{4}\Delta V_0 + \tfrac{1}{8}\Delta^2 V_0 - \cdots\right) \tag{13}$$

Equation 13 summarizes Longman's integration method, in which the improper integration is rewritten in terms of areas to be integrated individually. In the following subsection, the method is applied to solve a simple case of improper integration.

2.1 Illustration of the method

Consider the function $f(\mathrm{x})$ given in Eq. 5, with l=0.5 and ω=10. Its improper integral has a closed-form solution,

$$\int_0^\infty f(x)dx = \int_0^\infty e^{-\lambda x}\cos(\omega x)dx = \frac{\lambda}{\omega^2+\lambda^2}, \qquad (14)$$
$$\int_0^\infty e^{-\frac{1}{2}x}\cos(10x)dx = 0,004987531172$$

The roots of $f(\mathrm{x})$ in the integration domain are such that:

$$f(x_k) = 0 \Leftrightarrow \cos(10x_k) = 0 \Rightarrow x_k = \frac{1}{10}\left[\frac{\pi}{2}+(k-1)\pi\right],\ k\in \mathrm{Z}_+^* \qquad (15)$$

To perform the numerical calculations, an example with N=7 areas is considered. Eight roots x_k ($1 \le \mathrm{k} \le 8$) had to be determined. Table 1 shows the elements required to apply Longman's integration method. The second row shows the roots of the oscillating function. The third row presents the value of the integrated areas V_n. The following rows depict a sequence of the differences $\Delta^{r+1}\mathrm{V}_n$ up to order $\Delta^{r=5}\mathrm{V}_n = \Delta^6\mathrm{V}_n$ of the V_n areas ($1 \le \mathrm{n} \le \mathrm{N}$).

According to Eq. 13, the integral of f is then given by:

Table 1: Calculation of the areas of integration.

k	1	2	3	4	5	6	7	8
$\mathrm{x_k}$	0.157079	0.471238	0.785398	1.099557	1.413717	1.727876	2.042035	2.356194
$\mathrm{V_n} = \int_{x_n}^{x_n+1} \mathrm{f\,dx}$	0,171027	0,146166	0,124919	0,106760	0,091241	0,077978	0,066643	
$\Delta\mathrm{V_n}$	-0.024861	-0.021247	-0.018159	-0.015519	-0.013263	-0.011335		
$\Delta^2\mathrm{V_n}$	0.003614	0.003089	0.002640	0.002256	0.001928			
$\Delta^3\mathrm{V_n}$	-5.25e-4	-4.49e-4	-3.84e-4	-3.28e-4				
$\Delta^4\mathrm{V_n}$	7.636e-5	6.526e-5	5.578e-5					
$\Delta^5\mathrm{V_n}$	-1.110e-5	-9.487e-6						
$\Delta^6\mathrm{V_n}$	1.614e-6							

$$\int_0^\infty f(x)dx \approx \int_0^{x_1} f(x)dx - \left[\tfrac{1}{2}V_0 - \tfrac{1}{4}\Delta V_0 + \tfrac{1}{8}\Delta^2 V_0 - \tfrac{1}{16}\Delta^3 V_0 + \tfrac{1}{32}\Delta^4 V_0 - \tfrac{1}{64}\Delta^5 V_0 + \tfrac{1}{128}\Delta^6 V_0\right]$$

$$\int_0^\infty f(x)dx \approx 0{,}097204 - \left[\tfrac{1}{2}0{,}171027 - \tfrac{1}{4}(-0.024861) + \tfrac{1}{8}0.003614 - \tfrac{1}{16}\left(-5.25\cdot 10^{-4}\right) + \tfrac{1}{32}7.636\cdot 10^{-5} - \tfrac{1}{64}\left(-1.110\cdot 10^{-5}\right) + \tfrac{1}{128}1.614\cdot 10^{-6}\right]$$

$$\int_0^\infty f(x)dx \approx \underbrace{0{,}00498753}_{\text{accuracy}}2160155 \tag{16}$$

2.2 Serial implementation

An intuitive algorithm for Longman's integration method involves treating the part of Table 1 enclosed by a thicker dark line as an N×N square matrix, called **G** in this paper. The first line of **G** contains the integral of the areas V_n. The (r+2)-th line contains the differences $\Delta^{r+1}V_n$.

Each area V_n of the first line is calculated by classical methods of integration, such as the Gaussian Quadrature [Davis and Rabinowitz (2007)]. Hence, the term located at the k-th column of the first line is given by:

$$G(1,\, k) = \left| \int_{x_k}^{x_{k+1}} f(x)\, dx \right| = \left| \int_{-1}^{1} f(\xi)\, J(\xi) d\xi \right| \approx J(\xi) \sum_{g=1}^{Ng} f(\xi(x_g)) w_g \tag{17}$$

In Eq. 17, x_g and w_g are, respectively, the nodes and weights for Gaussian integration of order N_g [Davis and Rabinowitz (2007)]. $J(\xi)$ is the Jacobian responsible for transforming the range of the integration domain from $[x_k,\, x_{k+1}]$ to $[-1,\, 1]$.

The next $N-1$ lines in matrix **G** are determined by a data reduction scheme starting from the first one. Equation 8 is applied so that, at the end of the reduction, the first column will contain the terms $\Delta^{r+1}V_0$, $(-1 \le r \le N-2)$, over which Longman's formula is applied (Eq. 13). Notice that the hatched cells inside the thick dark line of Table 1 are not used. Therefore, in the same fashion, the corresponding cells of matrix **G** are left blank.

This approach can easily be parallelized. Considering that the calculation of each area of the first line does not depend on the calculation of the remaining areas, all of them can be integrated simultaneously. Further, in the data reduction, all the terms of a line can also be calculated in parallel. It is only required that the prior line has been completely filled in.

In this section, Longman's method for improper integration of oscillatory-decaying functions was described. The method was illustrated by an example and an intuitive implementation was overviewed. In the article of Espelid and Overholt (1994) a more elaborated integration scheme, including adaptativity, local and global error estimators is presented. The strategies described in the above cited article [Espelid and Overholt (1994)] may also be implemented on graphics hardware. In the next section the technology of computation on graphics hardware will be presented.

3 Parallel Computing on Graphics Hardware

Ordinary Central Processing Units (CPUs) must be capable of dealing with a variety of tasks demanded by a computer. Among them, there are recursive, adaptive, and interdependent problems, which demand a large amount of the computation resources to be dedicated to communication of data and control [Wloka, Zeller, Fernando and Harris (2009)]. On the other hand, graphics calculations such as pixel shading, vertex transformation and rasterization are tasks that require little control and communication, when compared to the volume of calculations [Wloka, Zeller, Fernando and Harris (2009)]. Because of that, graphics hardware has been developed since its beginning as data-parallel computing devices. They are specially well-suited to address problems that can be expressed as data-parallel computations with high arithmetic intensity (the ratio of arithmetic operations to memory operations) [NVidia (2008)].

For example, a typical card launched in the end of 2006 contained 128 calculation units, distributed among 8 vector multiprocessors. This architecture of cooperative multi-cored computing units is similar to the one found in some clusters of CPUs [Pacheco (1997)], but it is confined in a single hardware device. Figure 2 shows the basic architecture of a graphics card. As can be seen, the majority of the chip area is devoted to calculation, and only a small part of is dedicated to control and memory tasks.

Because of its architecture, this family of General-purpose Programmable Graphics Processing Units (GPGPUs) requires a single instruction–multiple data programming paradigm (SIMD). These cards have been used for implementation of solutions for many problems of engineering, physics, medicine, mathematics, etc., and an overall superior performance, compared to ordinary CPUs, has been observed [Ryoo, Rodrigues, Baghsorkhi, Stone, Kirk and Hwu (2008); Rasmusson, Mosegaard and Sørensen (2008); Stantchev, Juba, Dorland and Varshney apud Stantchev, Dorland and Gumerov (2008)]. A few APIs (Application Programming Interfaces) were also launched along with this new graphics cards [Peercy, Segal and Gerstmann (2006); CUDA (2009)].

CUDA (Computer Unified Device Architecture) is an API with which NVidia™ graphics cards can be programmed to perform non-graphics tasks. It is a low level language, because it requires the programmer to explicitly allocate and free memory, to declare data copies, to chose parameters of parallelism, and so forth. It is essentially an extension of the C programming language, with the addition of function type qualifiers, variable type qualifiers, kernel execution directives and some additional built-in variables. CUDA is multiplatform, as it can be compiled for any of the new NVidia's GPGPUs architectures [NVidia (2008)].

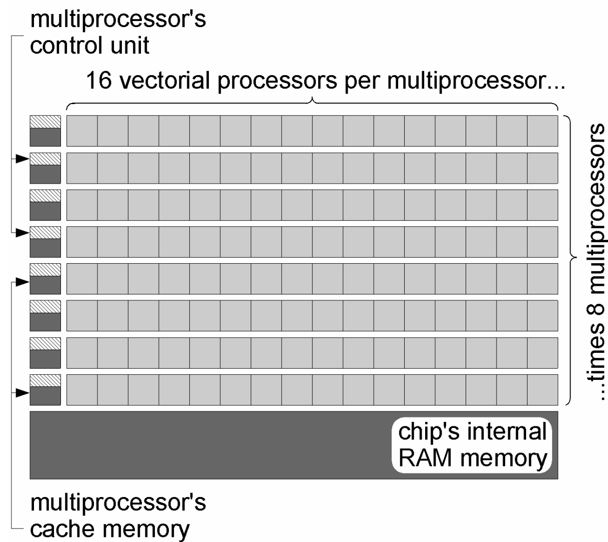

Figure 2: Example of architecture of a graphics card.

In CUDA programming, the concepts of thread, thread block and grid are fundamental. Thread is a virtualized CPU, the basic execution unit: it is the component responsible for executing a given instruction (the kernel) over a single data. Multiple threads may work in parallel executing the same kernel over a set of different data (SIMD paradigm). Thread blocks are used to spread the threads among the various processors of the graphics card. The division of the data in terms of blocks must take in consideration that each processor of the card will take care of one block at a time. In the card of Fig. 2, for example, 128 blocks would be executed simultaneously. If the data of the problem is divided in more than 128 blocks, the remaining ones will be automatically queued to be executed as soon as a processor is available.

There is another level of parallelization. Each thread block, in turn, admits the execution of the limited number of 32 threads at a time. This number is called *warp*. If the block has more than 32 threads, the remaining will automatically be queued for the next round of execution.

In Fig. 3, blank cells represent the data of the blocks that were not processed yet. Shaded cells represent the data being processed in the present round of execution, and hatched cells represent the data already processed. In the first round of execution, the first two thread blocks are assigned to the two processors of the card. The third block is queued. Inside each of the blocks (1) and (2), only 32 of their

36 threads are processed simultaneously by each processor. The four remaining threads are queued. In the second round, the processors deal with these final 8 threads. Only then, the next parcel of blocks is processed, which in this case means the third block. Notice that, in this problem, one of the processors is left inactive in the last two rounds of execution, which is an undesired waste of calculation resource.

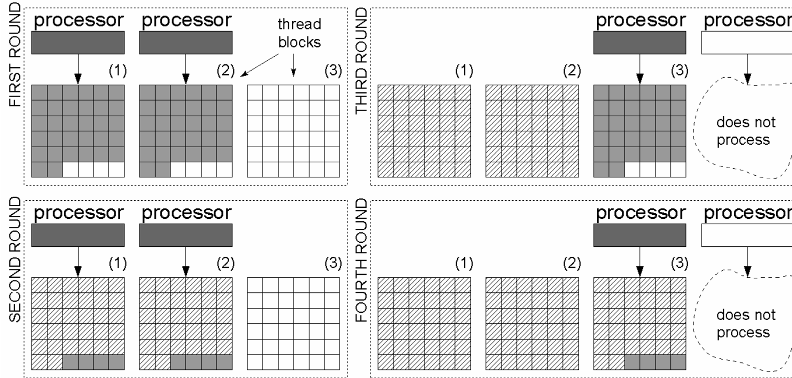

Figure 3: Reduced example of the two levels of parallelization.

Finally, grids are used to spread the data of the problem among thread blocks. For example, a single 16×16 thread block would not have enough room for the million of floating point variables of a 1000×1000 matrix.

This matrix would then be divided into a number of blocks of a given size. A thread block can be organized as a one-, two- or three-dimensional array of threads, and CUDA offers variables with which the index of every thread inside its block can be recovered. Analogously, the grids may be one-, two- or three-dimensional arrays of blocks. The thread blocks within the grids may also be identified by means of indices.

It is up to the programmer to decide in which way the data of the problem will be divided in terms of grids and thread blocks. This is a key decision which impacts directly on the efficiency of the program. A bad decision could cause the computational resources to be misused, as it happened in the example of Fig. 3. Recently, an application has been developed, in which these parameters can be determined automatically by metaprogramming [Klöckner and Hesthaven (2008)].

The number of multiprocessors in a GPU, the number of thread blocks which can be dealt simultaneously by each multiprocessor and the warp size depend on the card's model. For example, the NVidia™ model GTX 280 has 30 multiprocessors, each

one of them capable of dealing with 8 blocks simultaneously, and a warp size of 32 threads. Altogether, this card can execute the same kernel simultaneously over 30 multiprocessors, each containing 8 blocks and each block running 32 threads. This means that the defined kernel can be ran simultaneously on: $30 \times 8 \times 32 = 7680$ distinct data.

Graphics hardware presents a complex memory architecture (see Fig. 4). The most important of them is the *global memory*. The data placed in this memory are available to all the threads of a grid. Each thread block has its own *shared memory*, which presents a very limited, device-dependent size, but possessing an access time faster than global memory's [Manavski and Valle (2008)]. However, only the threads of the given thread block are allowed to access their block's shared memory. Furthermore, each thread has its own registers, accessed only by the thread itself. The graphics card also has the *constant* and *texture read-only* cache memories, devoted to specific purposes in the graphics calculation [NVidia (2008)]. Its specific properties, however, have been also explored for non-graphics purposes [Pharr and Fernando (2005); Nguyen (2007)]. Besides all this graphics hardware memories, a CUDA program also has to deal with the ordinary CPU RAM memory, as every classical low-medium level program does.

The execution of GPU programs requires a sophisticated manipulation of data between all these memories. All the vectors and matrices that might be accessed by the threads have to be allocated in the RAM memory of the CPU that hosts the graphics card, and also allocated in the GPU's global memory. Only pointers to these vectors are passed as arguments to the kernels.

At the end of the execution of a kernel, the data calculated by the threads are saved in the space allocated in the global memory of the GPU. It is necessary to copy back this data to the CPU's memory so that they can be printed, read, saved, etc.

All memory manipulation expends some processor clock cycles. A precise and fair benchmark of processing times between CPU and GPU will be achieved only if the times the GPU and the CPU consume to perform these memories operations are taken into account.

The following section will report how the programming concepts of GPGPU were approached in the present implementation of Longman's integration method.

4 Implementation on the GPU

In the present work, a parallel version of the algorithm described in Section 2.2 is implemented.

The calculation of each area V_n shown in Fig. 1 can be performed independently of its neighbor. This allows the first line of the matrix **G** to be calculated in par-

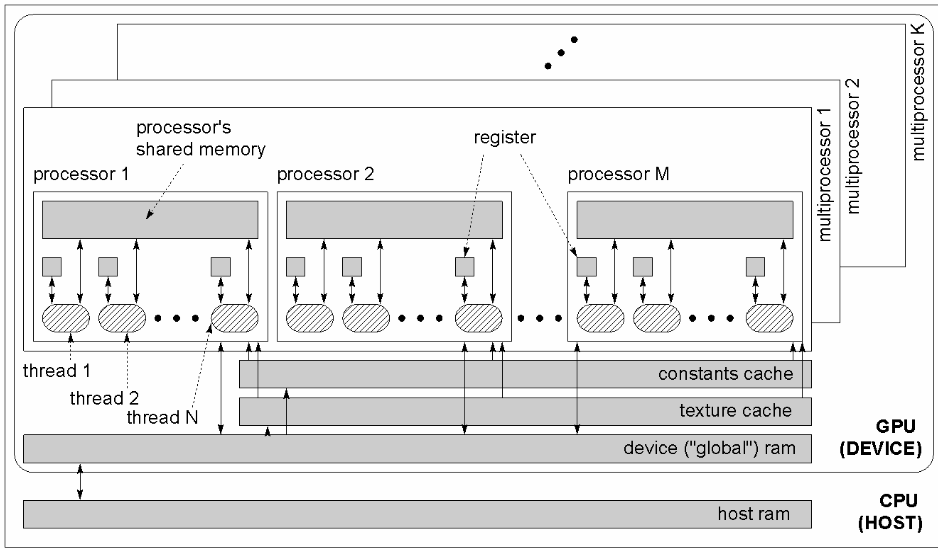

Figure 4: Graphics card memory hierarchy.

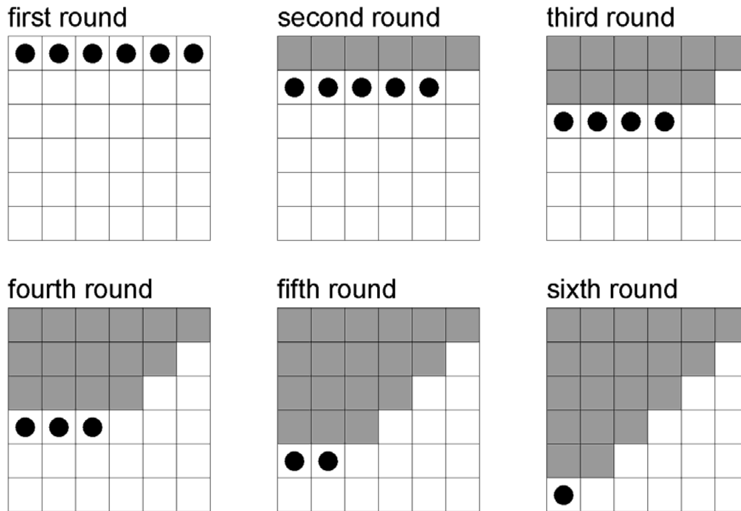

Figure 5: Steps to be executed in series by the GPU.

allel, in a single execution step. This characteristic is exactly what makes Longman's method so interesting to parallel implementation, and it is what motivated this work. For a problem in which N areas are calculated, $N-1$ parallel data reduc-

tion processes are needed to determine the remaining lines of **G**.

The model of parallel programming planed for the GPU in this work is illustrated by Fig. 5 for a reduced example of N = 6 areas. The cells marked with a circle within a line represent the terms that are being calculated simultaneously in the present round. The shaded cells shown in Fig. 5 represent the terms that were already calculated. Every element of one line can be calculated simultaneously. After all the elements of one line are calculated, the next line can be calculated. To perform the calculation of the example, a sequence of N = 6 rounds is needed.

Notice that matrix **G** is filled exactly in the way described in the example of Table 1. The shaded cells of Fig. 5 are analogous to the shaded cells of Table 1. The blank cells of Fig. 5, representing the untouched terms of **G**, are analogous to the hatched cells of Table 1. In the present approach, each individual area V_n is calculated by Gaussian Quadrature. It is also admitted that a vector **nx** containing the roots of f is given.

The matrix **G**, of size N $\times$ N, is fully stored in the thread block's shared memory, because its terms have to be read and written repeatedly along the course of the execution of the kernel. This fact poses a limit to the number of areas N, as the thread block's shared memory must store the N^2 terms of **G**. This limit of N depends on whether the implementation makes use of single or double precision floating point variables.

For the execution of the algorithm, a one-dimensional N $\times$ 1 thread block is created. Initially, the N threads calculate the terms of the first line of **G**, i.e., the areas V_n. Each thread does it by a serial summation of the Gauss points, according to Eq. 17. A synchronization point is inserted in order to guarantee that all the areas are calculated before the program goes on.

Next, the N$-$1 terms of the second line of matrix **G**(2, i) ($1 \leq i \leq N-1$), which represents a first step of a classical data reduction scheme, are calculated simultaneously by a set of threads, leading to the $\Delta^1 V_n$ elements. At this point a synchronization point is inserted to guarantee that all the first-order differences $\Delta^1 V_n$ are determined before the program proceeds. This operation is repeated over every line j ($1 \leq j \leq N-1$) until all the differences $\Delta^{r+1} V_n$ are calculated.

At the end of the data reduction, the first column of **G** contains, in its (r+2)-th line, the terms $\Delta^{r+1} V_0$. Now the N active threads multiply the terms ($\Delta^{r+1} V_0$) in the first column of matrix **G**(i, 1) by the factors $1/2^i$ ($1 \leq i \leq N$) that appear in Eq. 13. After this operation another synchronization point is inserted. Finally, a single thread sums in a serial operation all the weighted differences of Eq. 9 and subtracts

the result from the first part of the RHS of Eq. 13, leading to the desired result:

$$I_{Long} = \int_{a}^{x_1} f(x)dx - \sum_{j=1}^{N} G(j, 1) \tag{18}$$

The first term of Eq. 18 is also determined by Gaussian Quadrature in series by this single thread, in the same fashion that each area V_n was determined.

The procedures above describe the solution of a single integration by Longman's method, which requires the dedication of a single thread block. Examples of these integrals are given in Eqs. 1 to 3. On the other hand, as already discussed in the initial sections of this article, to perform numerical inversion of integral transforms like the ones shown in Eq. 2 and 3, these integration procedures must be repeated for many values of some transform variable. In the present work, multiple thread blocks running in parallel are used to solve multiple integrals.

To accomplish the execution of multiple thread blocks, a one-dimensional grid containing M thread blocks was designed. Each thread block is responsible for solving one integral by Longman's method as described above. It is necessary that the parameters of the function, in the present case the constants λ and ω of Eq. 5, be passed as arguments for the execution of the blocks. The roots of the function, between which the areas are defined, can be passed as arguments or calculated inside the kernels, in some cases. The graphics card will execute concurrently as many blocks (and therefore integrations) as its number of processors. The remaining of the M blocks will be queued for the next round of execution, as it was explained in the prior section.

5 Numerical Results

In the present work a program was written to integrate the function given in Eq. 5. It is assumed that the lower limit of integration in Eq. 13 is zero, a=0. A number M of blocks can be activated, which corresponds to the solution of M different integrals of this function. The constants λ_k and ω_k, k $\in$ [1, M], are passed as arguments in the kernel call. The roots of the function are not passed as argument. Once they are easy to be calculated in this case (see Eq. 15), each thread calculates its respective roots in the moment it calculates the corresponding integral. In other words, a given thread i solves the integral shown in Eq. 19.

$$G_k(1,\ i) = \int_{\left[\frac{\pi}{2}+(i-1)\pi\right]\omega_k^{-1}}^{\left[\frac{\pi}{2}+i\pi\right]\omega_k^{-1}} e^{-\lambda_k x} \cos\left(\omega_k x\right) dx \tag{19}$$

Figure 6 depicts a block diagram of the program. It shows clearly that part of the computations is performed within the CPU and part on the GPU, forming a CPU-GPU complementary system. The operations within the GPU are named *kernel calculations*.

Two distinct measures of execution time are taken. The first is the total time *tt*, measured from the initialization of the CPU up to the printing of the results. The second measure *tc* indicates the time specifically needed for the execution of the kernel (see Fig. 6). Only *tc*, which stands for *calculation time*, is related to the solution of the integral by Longman's method. The difference between *tt* and *tc*, called *ta*, is the time consumed by the code in memory operations. Within the *ta* time, it is included the time to allocate space in the CPU and GPU's RAM for the variables λ_k, ω_k and *integ*, which is the vector that will recover the results of the M integrals from the kernel. The time spent to copy these vectors between these memories is also included in *ta*.

The program was executed in an NVidia™ GTX 280 graphics card, the architecture of which was described in Section 3. The card was hosted by a 64 bits, dual-core AMD Athlon™ X2 CPU (2.6GHz). This model of card is nowadays one of the few GPUs whose hardware is natively capable of dealing with double precision arithmetic.

Whenever it was possible, hardware-implemented transcendental functions were used. These functions are known for improving performance on a penalty of a somewhat lower accuracy [CUDA (2009)].

Figure 7 shows the required execution time to perform M integrations. In this numerical experiment, N=16 areas were used to perform the integration. Single precision has been applied. As mentioned in Section 3, the architecture of the GTX 280 admits the simultaneous execution of up to 240 thread blocks (30 multiprocessors, each one of them capable of dealing with 8 blocks simultaneously). The hardware schedules automatically the remaining to be executed as soon as the present have been executed. Hence, it is observed that the execution time is organized in levels as the values of M go from 1 to 240 blocks, from 241 to 480, from 481 to 720, and so forth. The results show that up to M=240 blocks the processes run very well in parallel and execution time is almost insensitive to the number of blocks within this range.

In the next analysis the normalized total time (*tt*/M) spent by the GPU to solve small numbers of improper integrals ($1 < M < 5$) is compared to the time spent by the CPU.

In Fig. 8, the experiment is repeated for a larger number of integrals being solved and, consequently, a larger number of blocks being executed. The limit of integrals

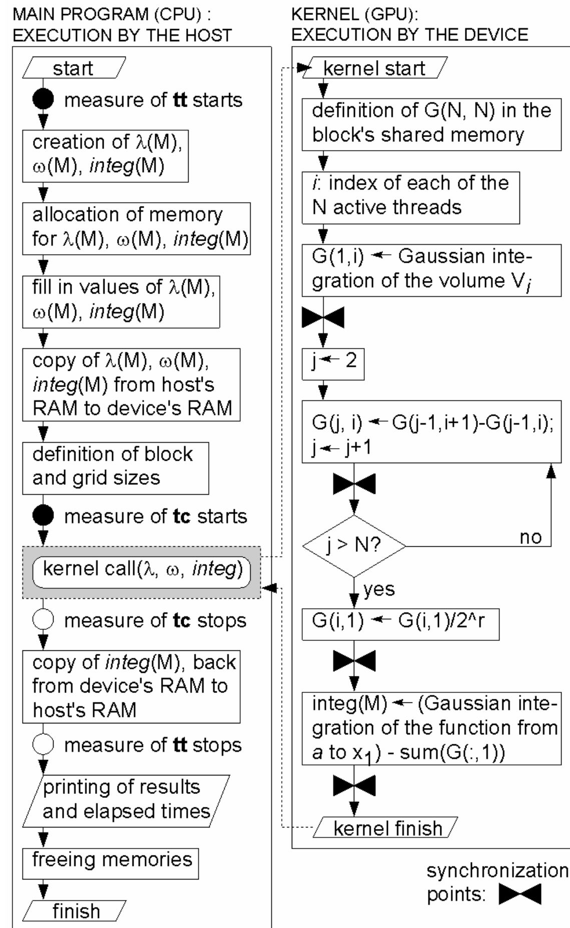

Figure 6: Execution flow of the present program, showing where the execution times are measured

that are possible to be solved corresponds to the maximum number of thread blocks per grid. This number is device-dependent, and in the present case is $M_{max} = 65{,}536$ [CUDA (2008)]. The figure indicates that the integration time varies almost linearly as the number of integrals surpasses by large the number of block threads that can be handled simultaneously by the card.

In this experiment, it can be observed that as number of integrals being solved increases, the parcel of computing capability devoted to memory manipulation (*ta*) decreases relatively to the total time (*tt*). In the next analysis the normalized total

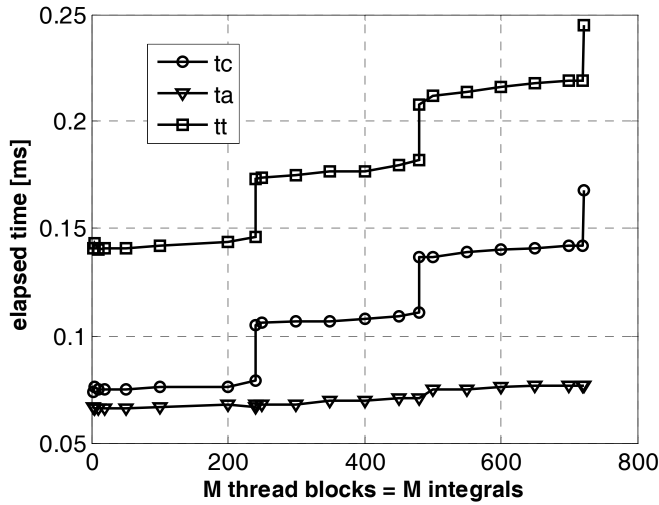

Figure 7: Execution time of the GPU for M thread blocks.

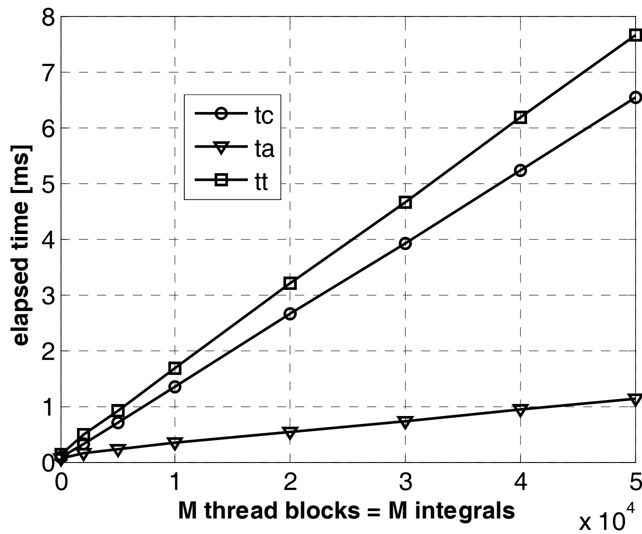

Figure 8: Execution times of the GPU for M up to 50 thousands thread blocks.

time (*tt*/M) spent by the GPU to solve small numbers of improper integrals ($1 < M < 5$) is compared to the time spent by the CPU. Four distinct number of areas per integration is considered (N = 8, 16, 24, 32). As the number of areas to be

integrated is smaller than the warp number of the graphics card (32 threads), all area calculations are performed by simultaneous threads within a thread block of the GPU.

Consider the results shown in Fig. 9. First it can be noticed that the normalized time (*tt*/M) required by the CPU to perform the integrations increases as the number of used areas also increases (N = 8, 16, 24, 32). But as the number of improper integrations to be solved increase (1 < M < 5), the normalized integration time decreases by a small amount and it tends to stabilize at a constant value.

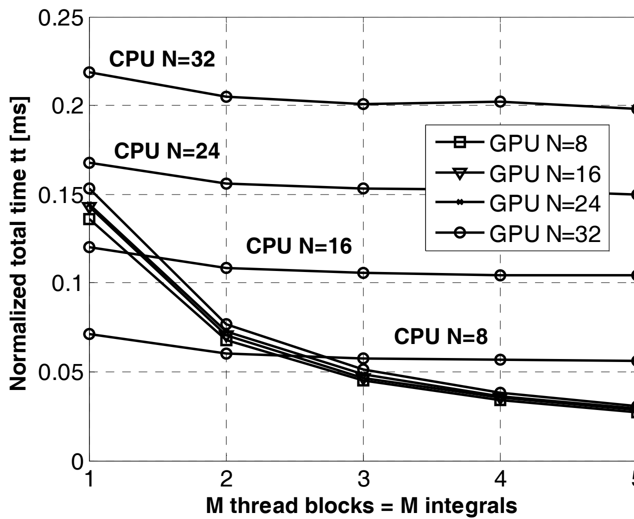

Figure 9: Comparisons of CPU and GPU performances for few integrals (1 < M < 5) but for distinct numbers of integration areas (8 < N < 32).

On the other hand, the time required by the GPU to solve the integrations is almost insensitive with respect to the number of areas being used in the integration (N = 8, 16, 24, 32). The fact that every area is calculated simultaneously by parallel threads within a thread block is the explanation for this behavior. The small difference between the GPU times is due to the allocation of variables of different sizes in each case.

Figure 9 also shows that for the case of one single improper integral (M=1) the CPU performs better than the GPU for low number of areas used in the integration (N = 8, 16). The reason to that is that, in order to execute the kernel that calculates this integral on the GPU, a few allocations and copies of memory between the CPU RAM and the GPU global memory are needed, which are not necessary in the CPU

program. This allocation time is rather short and depends little on the number of integrals M. So the increase of M causes this allocation time to be dissolved in the total execution time of the kernel. For a number of areas greater than 24 the GPU outperforms the CPU even for a single realization of the improper integral (M=1).

The same figure also shows that as the number of improper integrations (M) increase, the normalized time (*tt*/M) decreases for the GPU. The arithmetic intensity of the problem increases with the number of areas, and so does the performance of the GPU compared to the CPU.

For larger numbers of integrals, an overall superior performance of the GPU is observed in Fig. 10. This figure depicts the normalized integration time (*tt*/M) for the CPU and the GPU. This is essentially a continuation of Fig. 9 for larger number of integrals (M). It can be seen that the normalized calculation time for the CPU remains practically insensitive to the number of integrals, which means that every integration consumes the same amount of time. The total time therefore increases linearly with M. In the analysis shown in Fig. 10 the final number of integrals solved was M=50,000, each containing N=16 areas. The calculations were performed in single precision.

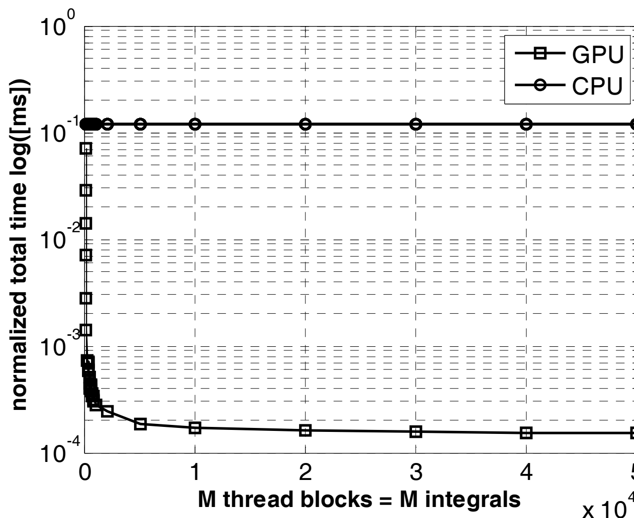

Figure 10: Comparison of normalized GPU and CPU times, with N = 16 and single precision.

The time spent by the GPU to solve all M=50,000 integrals was 6.508 ms. The normalized total time was then 0.1302 μs. The CPU spent 5.780 seconds to solve

the same number of integrals. For this large number of integrals, the performance of the GPU was 888 times better than the CPU.

Next an experiment was made, concerning the precision of the integration. The integral of Eq. 5 was performed with parameters λ=0.5 and ω=10. The lower integration bound was $a = x_1$, so that the formula of estimation of the remainder could be used (Eq. 10). Figure 11 shows the error of the present integration on the GPU for three different implementations: single-precision variables and hardware-implemented transcendental functions (SP+HTF), single-precision variables and ordinary software transcendental functions (SP+STF), and double-precision variables (DP).

As the literature advises [CUDA (2009)], the accuracy is harmed when the hardware functions are used. Nevertheless, the precision of this implementation is still within the bounds of the estimated remainder furnished in Eq. 10. For the other two implementations, Equation 10 has shown to be a good, conservative estimative of the remainder (integration error).

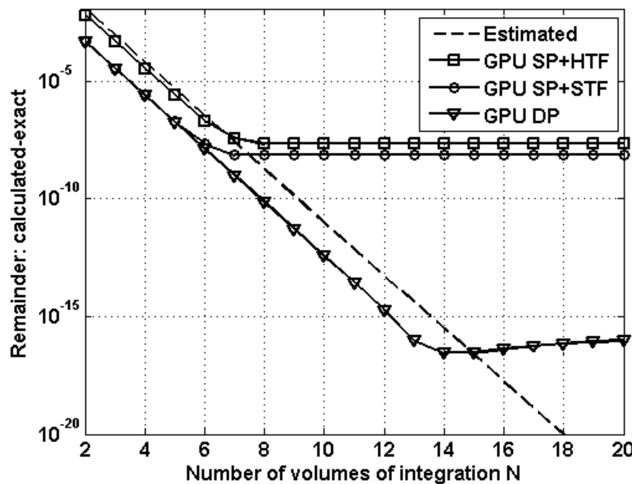

Figure 11: Comparison between the estimated remainder (error) and the real remainders (errors) of single and double precision.

There is, of course, a price for achieving double precision results, which is a performance decrease. Figure 12 compares the normalized total times (*tt*/M) of CPU and GPU for single and double precision implementations. In these experiments, N=16 areas of integration and software implemented transcendental functions were

used.

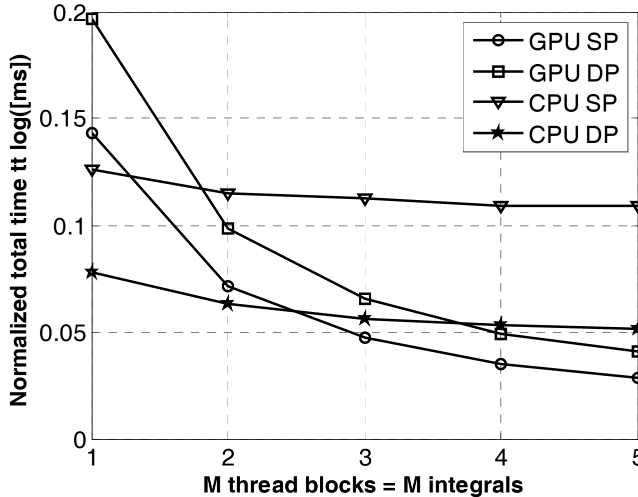

Figure 12: Comparison of performances on single and double precision calculations on the CPU and the GPU.

It can be seen that the time on the GPU increases for the double precision case. On the other hand the CPU results are faster for double precision. This kind of behavior, according to which C language implementations on double precision may outperform single precision implementations has already been observed in the literature [McMamee, (2004)]. This behavior is CPU hardware architecture dependent and is not an issue of the present article.

The results of Fig. 12 show that in single precision the GPU becomes more efficient when more than 2 integrals are performed (M > 2). For the double precision case the GPU is more efficient for M > 4.

Figure 13 shows, analogously to Fig. 10, the normalized total time (*tt*/M) of CPU and GPU for a larger number of integrals (M $\leq$ 50,000). Figure 13 also shows the GPU speed up for a large number of integrals determined with double precision.

In the final experiment, in which M = 50,000 integrals were solved with N = 16, the time spent by the GPU to solve the integrals on double precision was 40.848 ms, 6.3 times higher than in the single precision version. The normalized total time is *tt*/M= 0.827 μs. The CPU spent 2.243 seconds to solve the same 50,000 integrals in double precision. For this case the performance of the GPU was 55 times better than the CPU, as can also be seen inf Fig.13.

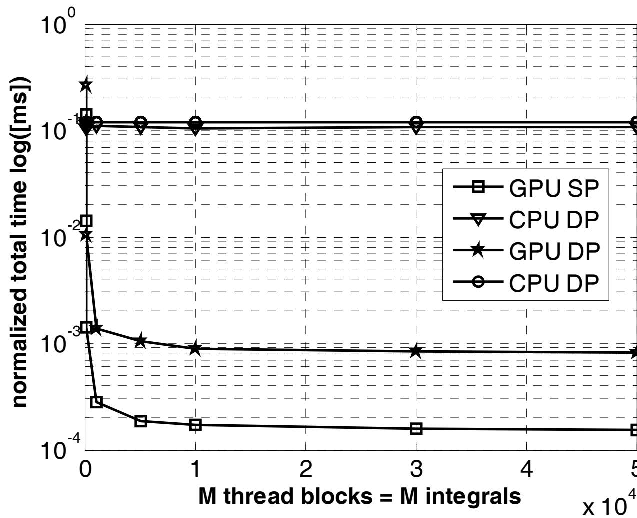

Figure 13: Comparison of normalized GPU and CPU times for single and double precision considering large number of integrals M with N=16.

6 Concluding Remarks

This paper has described the implementation of the Longman's integration method for improper numerical integration of oscillatory-decaying functions on graphics hardware (GPGPU). A classical serial implementation was rewritten under the SIMD parallel programming paradigm.

The basic structure of a GPU as observed from the CUDA programming environment has been described.

The paper has reported the performances of the GPU and the CPU on solving different numbers of integrals with various degrees of numerical accuracy. There is a number of integrals (M), starting from which the GPU is more efficient than its CPU counterpart. This number depends on the required accuracy of the integration (N) and on the kind of variables utilized, single or double precision. For N>16 the GPU was faster than the CPU even for a single integration (M=1). In the worst case, for single precision calculations, the GPU has outperformed the CPU for M>3. For double precision and N=16 the GPU was faster than CPU whenever more than 4 integrals (M>4) were calculated.

The GPU has shown an overall superior performance, being almost 900 times faster than the CPU when the number of integrals to be solved was very large and the single precision case was considered. For the double precision case the GPU was

55 times faster than its CPU counterpart.

The article indicates that numerical inversion of integral transforms, which present oscillating and decaying functions, is an ideally suited task for parallel computations on GPGPUs.

References

Araújo, F. C.; Gray, L. J. (2008): Evaluation of EffectiveMaterial Parameters of CNT-reinforced Composites via 3D BEM. *CMES: Computer Modeling in Engineering & Sciences*, vol.24, no.2, pp.103-121.

Bromwich, T. J. (1942): *An Introduction to the Theory of Infinite Series*. McMillan.

CUDA (2008): *Developer's Zone*. http://www.nvidia.com/ object/cuda home.html.

CUDA (2009): *NVIDIA CUDA C Programming Best Practices Guide*. NVIDIA Corporation, Santa Clara.

Davis, P. J.; Rabinowitz, P. (2007): *Methods of Numerical Integration*. 2nd Ed. Dover Publications, Mineola.

Espelid, T. O. ; Overholt, K. J.(1994): DQAINF: an algorithm for automatic integration of infinite oscillating tails, *Numerical Algorithms*, 8(1):83-101

Göddeke, D.; Strzodka, R.; Turek, S. (2007a): Performance and accuracy of hardware-oriented native-, emulated- and mixed precision solvers in FEM simulations, *The International Journal of Parallel, Emergent and Distributed Systems*, v.22, n.4, p.221-256.

Göddeke, D.; Strzodka, R.; Mohd-Yosof, J.; McCormick, P.; Buijssen, S.H.M.; Grajewski, M.; Turek, S. (2007b): Exploring weak scalability for FEM calculations on a GPU-enhanced cluster, *Parallel Computing*, v.33, p.685-699.

Göddeke, D.; Strzodka, R.; Mohd-Yosof, J.; McCormick, P.; Wobker H.; Becker, C.; Turek, S. (2008): Using GPUs to Improve Multigrid Solver Performance on a Cluster, *International. Journal of Computational Science and Engineering*, v.4, n.1, p.36-55.

Jones, W. (2004): *Beginning Directx9*. Premior Press.

Kaplan, W. (2002): *Advanced Calculus*. 5th Ed. Addison Wesley. 736 p.

Klöckner, A.; Hesthaven, J. S (2008): *Metaprogramming Graphics Processors from High-Level Languages*. http://mathema.tician.de/entry/dam

Korenev, B. G. (2002): *Bessel functions and their applications*. Chapman & Hall/CRC.

Longman, I. M. (1956): Note on a method for computing infinite integrals of

oscillatory functions. *Proc. Cambridge Phil. Soc.*; **52** : 764-768.

Manavski, S. A.; Valle, G. (2008): CUDA Compatible GPU Cards as Efficient Hardware Accelerators for Smith-Waterman Sequence Alignment. In: *BMC Bioinformatics* 2008. **9** : 1–10.

McNamee, J. M. (2004): A comparison of methods for accurate summation, *ACM SIGSAM Bulletin*, v.38 n.1.

Nguyen, H. (2007): *GPU Gems 3*. Addison-Wesley Professional, 2007.

NVidia (2008): *NVIDIA CUDA – Compute Unified Device Architecture – Programming Guide*. NVIDIA Corporation, Santa Clara.

Oishi, A.; Yoshimura, S. (2008): Finite Element Analyses of Dynamic Problems Using Graphics Hardware. *CMES: Computer Modeling in Engineering & Sciences*, vol.25, pp.115-131.

Owens, J. D; Luebke, D.; Govindaraju, N.; Harris, M.; Krüger, J.; Lefohn, A. E.; Purcell, T. J. (2007): A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*; **26** : 80–113.

Pacheco, P. (1997): *Parallel Programming with MPI*. Morgan Kaufmann: San Francisco. 418 p.

Peercy, M.; Segal, M.; Gerstmann, D. (2006): A performance-oriented data parallel virtual machine for GPUs. In: *ACM SIGGRAPH 2006 Conference Abstracts and Applications*.

Pharr, M.; Fernando, R. (2005): *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional.

Rasmusson, A.; Mosegaard, J.; Sørensen, T. S. (2008): Exploring parallel algorithms for volumetric mass-spring-damper models in cuda. In: *International symposium on computational models for biomedical simulation* (pp. 49–58).

Ryoo, S.; Rodrigues, C. I.; Baghsorkhi, S. S; Stone, S. S.; Kirk, D. B.; Hwu, W. M. W. (2008): Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, February 20-23, 2008, Salt Lake City, UT, USA.

Selvadurai, A. P. S. (2000): *Partial Differential Equations in Mechanics 2*, Springer.

Shreiner, D.; Woo, M.; Neider, J.; Davies, T. (eds.) (2005): *OpenGL Programming Guide*, 5th edition Addison-Wesley.

Stantchev, G. Dorland, W.; Gumerov, N. (2008): Fast parallel Particle-To-Grid interpolation for plasma PIC simulations on the GPU, *Journal of Parallel and Dis-*

tributed Computing, v.68 n.10, p.1339-1349.

Stantchev, G.; Juba, D.; Dorland, W.; Varshney, A.: High-performance computation and visualization of plasma turbulence on graphics processors, *Computing in Science and Engineering* (in press).

Takahashi, T.; Hamada, T. (2009): CPU-accelarated boundary element method for Helmholtz equation in three dimensions. *Int. Journal for Numerical Methods in Engineering*; (published online) DOI: 10.1002/nme.2661.

Wloka, M.; Zeller, C.; Fernando, R.; Harris, M. (2004): *Programming Graphics Hardware*. http://http.download.nvidia.com/developer/presentations/2004/Eurographics/EG_04_TutorialNotes.pdf.