NOVEL APPROACHES IN IMPLEMENTING THE LEGENDRE SPECTRAL-COLLOCATION METHOD USING THE COMPUTE UNIFIED DEVICE ARCHITECTURE

Dana-Mihaela PETROŞANU[1], Alexandru PÎRJAN[2]

In this paper, we develop and propose a powerful, efficient solution for numerically solving second order partial differential equations with general boundary conditions using the spectral discretization of the equations and the Legendre spectral-collocation method. The developed solution has a multitude of applications in solving general second order partial differential equations that allow the separation of variables, with general boundary conditions. The novel aspect of our research consists in developing and implementing a solving algorithm on the latest parallel processing architecture, Maxwell, using the latest features of the Compute Unified Device Architecture Toolkit 6.5, thus obtaining an efficient high performance solution.

Keywords: partial differential equations, Legendre-Gauss-Lobatto grids, parallel processing architecture, Compute Unified Device Architecture

1. Introduction

In the last years, the researchers expressed a great and increasing interest for using parallel computing in computer science and in developing solutions for solving problems from various fields: economy, biology, chemistry, physics, mechanics, geospatial information systems, mathematics, medicine, architecture and many others. The explosive evolution of the hardware and software architectures potentiated the researches in the above-mentioned fields, facilitating the development of efficient parallel computing algorithms, implemented on graphics processing units (GPUs). The new generations of these processors, based on the Compute Unified Device Architecture (CUDA), offer a huge parallel processing power at affordable costs, leading to reduced execution times and tremendous efficiency. Thus, new opportunities and challenges have emerged for the researches from different fields. Among these, an area that proves to be of particular interest consists in the study of numerical methods for partial differential equations, in the development and implementation of algorithms on CUDA-enabled GPUs [1], [2]. In some of these studies, the numerical Legendre

[1] Lecturer, Department of Mathematics-Informatics, University POLITEHNICA of Bucharest, Romania, e-mail: danap@mathem.pub.ro
[2] Lecturer, Department of Informatics, Statistics and Mathematics, The Romanian-American University, Bucharest, Romania, e-mail: alex@pirjan.com

spectral-collocation method poses a great interest, but none of the works so far (to our best knowledge) has researched and implemented optimization solutions for solving equations using the Legendre spectral-collocation method implemented on the Maxwell architecture.

In this paper, we have first briefly recalled issues regarding the Legendre-Gauss-Lobatto grids and afterwards, we have analyzed the matrix diagonalization method, useful for the spectral discretization of the elliptic and parabolic equations that allow the separation of variables. We have depicted the method and its applications in solving elliptic bidimensional and tridimensional equations, with homogeneous Dirichlet boundary conditions. The method can be extended to more general second order partial differential equations that allow the separation of variables, with general boundary conditions. As the matrix diagonalization stage is a high resource consuming process, we have considered of paramount importance to optimize this step of our computing algorithm by employing the huge parallel processing power of a CUDA enabled Graphic Processing Unit (GPU) from the Maxwell architecture and the new features offered by the CUDA Toolkit 6.5 development environment. Afterwards, we have analyzed the latest enhancements offered by the Compute Unified Device Architecture (CUDA) Toolkit 6.5 and finally, using these features, in order to apply the above-mentioned theory, we have analyzed a particular case of equation.

We have developed two versions of an application that implements the algorithm for computing the numerical solutions of the considered equation, using two different approaches: a "classical" approach, based on processing the data solely on a Central Processing Unit (CPU) and a "novel" hybrid approach, based on processing the data both on a CPU and on a CUDA enabled GPU from the latest Maxwell architecture, Nvidia GTX 980, harnessing its huge parallel processing power. Analyzing the obtained experimental results, we have concluded that our novel approach for computing the numerical solution of second order partial differential equations with general boundary conditions is a high-performance and useful tool.

2. The Legendre-Gauss-Lobatto grids

The Legendre-Gauss-Lobatto grids are of paramount importance in obtaining numerical solutions of partial differential equations using the nodal spectral methods approach. The study of the zeros for the orthogonal polynomials and their derivatives pose a lot of interest in the literature [3], [4]. We address this subject as it represents the starting point for studying the Legendre-Gauss-Lobatto grids, generated by the zeros of the Legendre orthogonal polynomials' derivatives. The Legendre polynomials are particular cases of the ultraspherical (or

Gegenbauer) polynomials that are briefly presented below, in order to recall their definition and main properties.

Considering $\lambda > -\frac{1}{2}$ and $n \in \mathbb{N}$ fixed parameters, the ultraspherical (or Gegenbauer) polynomials $C_n^{(\lambda)}$ of degree n represent the orthogonal polynomials on the interval $(-1,1)$, having the weighting function:

$$\rho^{(\lambda)}(x) = (1-x^2)^{\lambda-\frac{1}{2}}. \quad (1)$$

For any degree $n \in \mathbb{N}$, these polynomials satisfy the symmetry relation:

$$C_n^{(\lambda)}(-x) = (-1)^n C_n^{(\lambda)}(x) \text{ for all } x \in \mathbb{R} \quad (2)$$

the differentiation rule:

$$\frac{d}{dx} C_n^{(\lambda)}(x) = 2\lambda C_{n-1}^{(\lambda+1)}(x) \text{ for all } x \in \mathbb{R} \quad (3)$$

and are solutions of the linear ordinary differential homogeneous equation of the second order:

$$(1-x^2)y'' - (2\lambda+1)xy'(x) + n(n+2\lambda)y(x) = 0. \quad (4)$$

Using the Rodrigues formula for the orthogonal polynomials [5], the ultraspherical polynomials can be written as:

$$C_n^{(\lambda)}(x) = a_n(1-x^2)^{\frac{1}{2}-\lambda}\frac{d^n}{dx^n}\left[(1-x^2)^{n+\lambda-\frac{1}{2}}\right], \text{ where } a_n \in \mathbb{R} \quad (5)$$

A particular case of the Gegenbauer polynomials is represented by the Legendre polynomials P_n, obtained when $= \frac{1}{2}$, therefore $P_n = C_n^{(1/2)}$. In this case, the weighting function is:

$$\rho(x) = 1 \quad (6)$$

and the Rodrigues formula applied for the Legendre polynomials gives the following form of the Legendre polynomials:

$$P_n(x) = a_n\frac{d^n}{dx^n}[(1-x^2)^n], \text{ where } a_n \in \mathbb{R}. \quad (7)$$

Using the Legendre polynomials, one can define now the Legendre-Gauss-Lobatto (LGL) nodes and grids. For each $0 \le k \le n, k \in \mathbb{N}$, the LGL nodes ξ_k^n of order n are the $(n+1)$ zeros of the polynomial

$$(1-x^2)P'_n(x) \quad (8)$$

where P'_n is the first derivative of the n-th degree Legendre polynomial. We consider the LGL nodes sorted in an increasing order, $\xi_k^n < \xi_{k+1}^n$, for each $0 \le k \le n-1$. The set of all these nodes,

$$\Gamma_n^{LGL} = \{\xi_k^n | 0 \le k \le n\} \quad (9)$$

represents the LGL grid of order n.

For a Γ_n^{LGL} grid, one can compute for every $0 \le k \le n-1$, the corresponding Legendre-Gauss-Lobatto intervals, having in the boundaries two adjacent LGL nodes, ie:

$$\Delta_k^n = [\xi_k^n, \xi_{k+1}^n] \subset [-1,1]. \quad (10)$$

of length $l_k^n = \xi_{k+1}^n - \xi_k^n$.

One must remark that the polynomial in the relation (8) has the zeros

$$\xi_0^n = -1, \xi_n^n = 1 \text{ and } \xi_k^n \in (-1,1) \text{ for } 1 \leq k \leq n-1. \quad (11)$$

Using the symmetry relation (2), one can easy conclude that the Legendre-Gauss-Lobatto nodes are symmetric with respect to the origin,

$$\xi_k^n = -\xi_{n-k}^n \text{ for } 0 \leq k \leq n.$$

Therefore, the LGL grids on the $[-1,1]$ interval are symmetric around zero. If the degree $n \in \mathbb{N}$ of the Legendre polynomial is an even number, then $\xi_{n/2}^n = 0$ and it represents the middle of the $[-1,1]$ interval. An interesting and useful result regarding the Δ_k^n Legendre-Gauss-Lobatto intervals refers to their lengths monotonicity. Thus, one can easily prove [6] that the LGL intervals' lengths l_k^n are strictly increasing from the boundaries to the center of the $[-1,1]$ interval.

3. The matrix diagonalization method

In the following, we analyze the matrix diagonalization method, useful for the spectral discretization of the equations that allow the separation of variables. We consider an elliptic bidimensional equation, with homogeneous Dirichlet boundary condition:

$$\begin{cases} au - \Delta u = f \text{ in } D_2 = (-1,1) \times (-1,1) \\ u|_{\partial D_2} = 0 \end{cases}. \quad (12)$$

We denote by Π_n the space of the polynomials having the degree $k, 0 \leq k \leq n$, by $A_n = \{P \in \Pi_n | P(\pm 1) = 0\}$ and by $\{x_i = y_i, i = 0, n\}$ the set of Legendre-Gauss-Lobatto nodes on the Ox and Oy axis. According to the formula (11) from the previous section, we have $x_0 = y_0 = -1$ and $x_n = y_n = 1$.

We now apply the Legendre spectral-collocation method, useful for obtaining the numerical solution of ordinary differential equations, partial differential equations and integral equations. We intend to find $u_n \in A_n \times A_n$ that satisfies the equation:

$$au_n(x_i, y_j) - \Delta u_n(x_i, y_j) = f(x_i, y_j), 1 \leq i, j \leq n-1. \quad (13)$$

We consider $L_j(x) \in \Pi_n$, the Lagrange polynomials corresponding to $\{x_k, k = 1, n\}$, therefore $L_j(x_i) = \delta_{ij}$, where δ_{ij} is the Kronecker's delta. We consider $u_n(x, y)$ having the form:

$$u_n(x, y) = \sum_{i,j=1}^{n-1} u_n(x_i, y_j)\, L_i(x) L_j(y) \quad (14)$$

By introducing (14) in (13) and denoting:

$$m_{ij} = -L_j''(x_i), M = (m_{ij})_{1 \leq i,j \leq n-1}$$
$$U = (u_n(x_i, y_j) = u_{ij})_{1 \leq i,j \leq n-1}, \quad (15)$$
$$F = (f(x_i, y_j) = f_{ij})_{1 \leq i,j \leq n-1}$$

the equation (12) could be written:

$$aU + MU + UM^T = F. \quad (16)$$

In order to highlight the eigenvalues of the matrix M and their corresponding eigenvectors, we consider the equation:

$$ME = E\Lambda, \tag{17}$$

where Λ is the matrix that contains on the principal diagonal the eigenvalues $\lambda_1, \lambda_2, \dots, \lambda_{n-1}$ of the matrix M, while E is the matrix whose columns are the corresponding eigenvectors. The values $\lambda_1, \lambda_2, \dots, \lambda_{n-1}$ are all real and positive numbers [7].

Considering U in the form:

$$U = ETE^T \tag{18}$$

and multiplying then in the equation (16) by E^{-1}at the left and by E^{-T} at the right, we obtain:

$$aT + \Lambda T + T\Lambda = H, \text{where } H = E^{-1}FE^{-T} \tag{19}$$

If we write the equation (19) on components, we obtain the equations:

$$(a + \lambda_i + \lambda_j)T_{ij} = h_{ij}. \tag{20}$$

Each of these equations contains only the (i, j) component of the unknown matrix T, thus there are decoupled equations, since the variables do not interact with each other; each variable can be solved independently, without knowing anything about the others.

In conclusion, in order to compute the components of the matrix U, one must first compute the matrices M, E, E^{-1}, Λ, then the matrix $H = E^{-1}FE^{-T}$ from equation (19), then solve the decoupled linear system (20) for obtaining the matrix T and finally, compute U through the equation (18). This ratiocination represents the basis for the future development of our numerical computation algorithm for obtaining the solutions of the previously studied type of equations (but also of other types) and we will present it in detail later, in the Section 5.

A similar ratiocination could be used in order to solve an elliptic tridimensional equation on the domain $D_3 = (-1,1) \times (-1,1) \times (-1,1)$, with homogeneous Dirichlet boundary condition:

$$\begin{cases} au - \Delta u = f \text{ in } D_3 \ , \\ u|_{\partial D_3} = 0 \end{cases} \tag{21}$$

We obtain the same steps as in the two-dimensional case, the difference being that all the calculations take place in $\mathbb{R}^3$.

Moreover, the above-depicted method could be extended to more general second order partial differential equations that allow the separation of variables, with general boundary conditions [8]:

$$\begin{cases} (a(x) + b(y))u - \frac{\partial}{\partial x}(\alpha(x)\frac{\partial u}{\partial x}) - \frac{\partial}{\partial y}(\beta(y)\frac{\partial u}{\partial y}) = f(x,y) \text{ in } D_2 = (-1,1) \times (-1,1) \\ \alpha_1 u(x,1) + \beta_1 u(x,1) = g_1(x), \forall x \in (-1,1) \\ \alpha_2 u(x,-1) + \beta_2 u(x,-1) = g_2(x), \forall x \in (-1,1) \\ \gamma_1 u(1,y) + \delta_1 u(1,y) = h_1(y), \forall y \in (-1,1) \\ \gamma_2 u(-1,y) + \delta_2 u(-1,y) = h_2(y), \forall y \in (-1,1) \end{cases} \tag{22}$$

From the computational point of view, the most resource consuming steps of this algorithm are the ones in which the matrix multiplications are computed, ie when the matrix H from the equation (19) is computed and later, when computing U through the equation (18). Therefore, when processing these steps, it is of paramount importance to employ the huge parallel processing power of the CUDA enabled GPU and the new features offered by the development environment CUDA Toolkit 6.5.

4. The latest enhancements offered by the Compute Unified Device Architecture (CUDA) Toolkit 6.5

The Compute Unified Device Architecture (CUDA) has been developed by the NVIDIA Company with the intent to create a perfect interaction between a parallel computing platform and a programming model so that the overall computing performance is significantly improved by harnessing the computational power of the Graphic Processing Units (GPUs). The GPUs are no longer restricted to graphics processing. Along with the introduction of this technology, CUDA enabled GPUs can be used to perform general-purpose computations. In contrast with the traditional central processing units (CPUs), the GPU expose a parallel architecture in which data is processed using thousands of simultaneously execution threads. In today's information age many scientists, researchers and developers use CUDA enabled GPUs in a broad range of applications and researches from different fields: mathematics, informatics, physics, finance, medicine, aeronautics etc. The CUDA development environment has been evolving significantly ever since its first version was launched in 2006 up to the latest CUDA 6.5 Toolkit from August 2014. This version brings a series of significant improvements regarding the parallel programming features. In the following, we briefly recall the main features offered by the CUDA Toolkit 6.5 development environment.

The Unified Memory is a significant feature offered by the CUDA Toolkit 6.5 that allows a software application to access both the CPU's memory and the GPU's memory without having to copy explicitly the data between the host and the device equipment. Thus, the programming effort of the developer is simplified because certain steps are automatically processed without the need of explicit supplementary programming instructions. Before CUDA 6, the CPU's memory was distinct from the GPU's memory, the two memories being separated by the PCI-Express bus. For sharing data between the two memories, the developers had to use explicit memory allocation instructions on both the host and the device along with explicit copy operations. This was a time consuming process for the programmer that had to program all these steps in order to have access to the data.

The above-mentioned limitations can be overcome using the Unified Memory feature leading to a simplified memory management process when developing software applications that harness the parallel processing power of the CUDA enabled GPUs. The Unified Memory offers a manageable memory zone that is shared between the GPU and the CPU, eliminating the need to use explicit allocation and copy instructions. The Unified Memory can be accessed by the CPU and by the GPU using a single pointer.

The whole process can be summarized as follows: the system automatically transfers the data from the Unified Memory between the CPU (the host) and the GPU (the device). The Unified Memory acts as if it were CPU memory when it receives data requests from the CPU and acts as GPU memory when the data requests are made from the GPU. From the programmer's point of view, the main advantages that are obtained by using the Unified Memory are:

- The Unified Memory simplifies the process of parallel programming when developing CUDA applications. Due to the improvements of the Unified Memory, the programmers are now able to focus directly on developing parallel CUDA kernels, without wasting time in programming the details concerning the allocation and copying of memory within the device equipment.
- By using the Unified Memory, the programming process within the CUDA Toolkit 6.5 is more flexible and enhances the source code porting on GPUs when necessary. The Unified Memory makes it easier to access and manage the memory, to manage complex data structures within the device.
- The Unified Memory automatically transfers the data between the CPU and the GPU and offers to the GPU the same level of performance as if the data had been stored in local memory while providing access to the globally shared data. All of these new features are possible through optimizations of the driver and of the CUDA runtime, offering the possibility of developing the CUDA kernels at a much faster rate than it had been possible before.
- One of the main objectives that must be targeted when employing the Unified Memory concept is to harness the whole available bandwidth of the CUDA streaming multiprocessors.

Another important aspect that must be taken into account for is that are certain technical scenarios when a CUDA software application that uses streams and asynchronous memory copies offers a higher degree of performance than a program that uses solely the Unified Memory. The main cause for this situation consists in the fact that the CUDA runtime does not have the same amount of information that the human factor has regarding the data that has to be processed. In order to successfully tune a CUDA application and obtain an optimum

performance, CUDA developers must use a complex set of tools for improving the software performance: shared memory management, asynchronous memory copies, CPU-GPU concurrency etc. A CUDA programmer must look upon the Unified Memory Concept as an added tool available to improve his parallel programming activity without neglecting the complex performance enhancing solutions that the CUDA runtime has to offer.

The cudaMemcpy() instruction is no longer mandatory but it is still available to the developer as a powerful optimization tool. Using the new cudaMallocManaged() instruction the Unified Memory is allocated and one can share complex data structures between the CPU and the GPU. Therefore, the CUDA programming can be performed with less effort because the kernel functions can be coded directly instead of consuming time and resources for managing the data, for maintaining data duplicates in both the memories of the host and device equipment. The cudaMemcpy() and cudaMemcypAsync() instructions are still available to the developer. These functions are extremely useful for particular problems and can lead to a significant increase in performance due to the source code's optimization, but their use is no longer mandatory. The NVIDIA Company intends to bring subsequent hardware and software optimizations regarding the flexibility and performance of software applications that have been developed using the Unified Memory feature. The company intends to add features regarding data prefetching that stores in advance the data in the cache memory before the processing operations are needed. The support for operating systems is also going to be extended offering more opportunities for broadening the software applications[3].

5. Examples and numerical results

In the following, we discuss a particular case of the general equation and boundary conditions (22), mentioned in the 3rd section,

$$\begin{cases} (x+y)u - \frac{\partial}{\partial x}\left(sin\pi x \frac{\partial u}{\partial x}\right) - \frac{\partial}{\partial y}\left(cos\pi y \frac{\partial u}{\partial y}\right) = \\ = e^{x+y}\left[(x+y) - \pi\cos\pi x + \pi\sin\pi y - \sin\pi x - \cos\pi y\right] in\ D_2 = (-1,1)\times(-1,1) \\ u(x,1) = e^{x+1},\ u(x,-1) = e^{x-1}, \forall x \in (-1,1) \\ u(1,y) = e^{y+1}, u(-1,y) = e^{y-1}, \forall y \in (-1,1) \end{cases} \quad (23)$$

We focus now our interest on analyzing the efficiency of the above-depicted numerical method and therefore, we have chosen the equation (23) as we already know its analytical solution,

$$u(x,y) = e^{x+y}\ in\ D_2 = (-1,1)\times(-1,1) \quad (24)$$

Our main purpose is to develop a numerical method, an algorithm for computing the numerical solution that approximates the analytic one and

[3] http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/, accessed on 04.06.2014

especially to develop implementations of the algorithm that provide maximum software performance.

In this purpose, we develop two versions of an application and we implement our algorithm for computing the numerical solutions of the equation (23), using two different approaches:

- a "classical" approach, based on processing the data solely on a CPU
- a "novel", hybrid approach, based on processing the data both on a CPU and on a CUDA enabled GPU, from the latest Maxwell architecture, Nvidia GTX 980, harnessing its huge parallel processing power.

In order to compute the components of the matrix U from the equation (18), we have designed the algorithm as to process the steps described in *Table 1*.

Table 1

The algorithm's steps and the processing units on which they are implemented

The step	The executed tasks	The processing unit	
		The "classical" implementation	The "novel", hybrid implementation
1	Computes the elements of the matrix M.	CPU	CPU
2	Computes the elements of the matrices E, E^{-1}, Λ.	CPU	GPU
3	Computes the matrix $H = E^{-1}FE^{-T}$ from equation (19), using data from Step 2.	CPU	GPU
4	Solves the decoupled linear system (20) for obtaining the matrix T.	CPU	GPU
5	Computes U through the equation (18).	CPU	GPU

Our implementation on the CPU made use of the BLAS (Basic Linear Algebra Subprograms) library and on the GPU we have used the corresponding optimized CUDA Basic Linear Algebra Subroutines (cuBLAS) library. We have optimized the memory management by using the Unified Memory feature available in the CUDA Toolkit 6.5 for devices with the Compute Capability of SM 3.0 and above in 64-bit environments. We have developed and implemented the CUDA kernel so that it scales according to the input data's size, offering a high level of performance on a broad scale, from small sizes up to huge sizes of the input data, the number of blocks and threads being automatically adjusted and scaled accordingly. We have employed a load balancing strategy and decomposition technique in order to increase the parallelism and achieve a higher throughput.

In the benchmarking, we have used the following hardware and software configuration: the Intel i7-4770K processor, operating at 3.5 GHz, with 16 GB (2x8GB) of 1333 MHz, DDR3 dual channel; the Nvidia GeForce GTX 980 graphic card from the latest Maxwell architecture; programming and access to the GPU used the CUDA toolkit 6.5.19, with the NVIDIA driver version 343.98. In

addition, in order to reduce the external traffic to the GPU, all the processes related to graphical user interface have been disabled. We have used the Windows 8.1 64-bit operating system.

In both implementations, in order to compute the average execution time that the processing units spend for computing the execution steps, we have used the "StopWatchInterface" available in the CUDA Toolkit 6.5. In order to compute correctly the average execution time, we have implemented separate timers for each of the tasks. The GPU begins to execute the code while the CPU continues the execution of the next line code before the GPU has finished, thus the execution is asynchronous. We have taken into account this fact and so we had to use the "cudaDeviceSynchronize()" instruction in order to be sure that all the execution threads have processed their tasks before timing the event. In this way, we get a reliable measurement of the execution time.

In *Table 2*, we present the experimental results obtained by taking different values for n, the maximum degree of the polynomials from the space Π_n introduced in the Section 3. The value n helps obtaining the cardinal $(n+1)$ of the set of Legendre-Gauss-Lobatto nodes $\{x_i = y_i, i = 0, n\}$ considered on the Ox and Oy axis. Each of the 1-10 lines of the table represents an average of 10,000 iterations, computed after having removed the first five results, in order to be sure that the GPU reaches its maximum clock frequency. The unit of measure is milliseconds (ms).

Table 2

The average execution time for different values of n

Test number	n	The analyzed case	
		The execution time in the "classical" implementation T1	The execution time in the "novel" hybrid implementation T2
1	16	0.043	0.039
2	32	0.061	0.044
3	64	0.095	0.053
4	128	0.113	0.045
5	256	0.275	0.076
6	512	0.659	0.116
7	1024	1.542	0.214
8	2048	2.989	0.315
9	4096	5.33	0.517
10	8192	11.937	1.012

The impact that the values of n has on the average execution time (based on the results listed in the *Table 2*) is represented in **Fig. 1**.

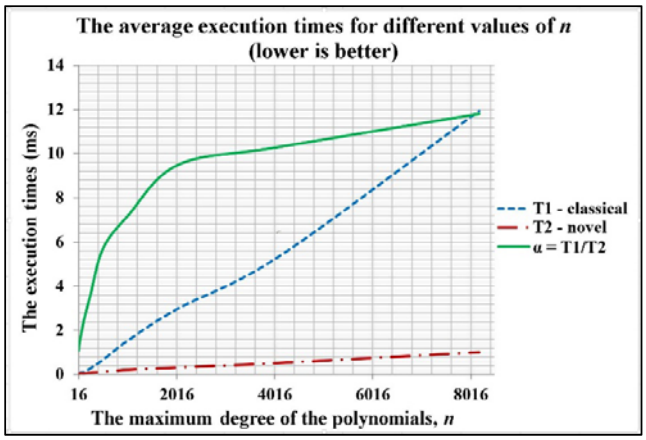

Fig. 1. The average execution times for different values of *n*

As we had expected, the average execution time increases with the values of n, but the performance penalty is more drastically felt in the case of the "classical" implementation. As the values of n increase, both the values of T1 and T2 increase, but the value of T2 increases at a much slower pace than T1. Thus, for $n = 8192$, $T1 = 11.937\ ms$ while $T2 = 1.012\ ms$. In order to highlight the improvements in software performance brought by our second solution, the "novel" hybrid implementation, we have also computed and represented the software improvement ratio, $\alpha = T1/T2$. The value of this ratio has been varying in our experimental tests between 1.1 (corresponding to $n = 16$) and 11.8 (corresponding to $n = 8192$).

We have prototyped the CUDA kernel functions more efficiently with less programming effort, by taking advantage of the new Unified Memory feature available in the CUDA Toolkit 6.5.

6. Conclusions

The matrix diagonalization method and the Legendre spectral-collocation method are useful for obtaining the numerical solutions of a wide range of equations. In our research, we have used these methods for solving equations that allow the separation of variables, with general boundary conditions. The numerical method and the algorithm based on it have been successfully implemented using two different approaches.

The novel aspect of our research consists in developing and implementing the algorithm on the latest Maxwell architecture using the novel features offered by the CUDA Toolkit 6.5, thus obtaining an efficient high performance solution. We have optimized the memory management by using the Unified Memory concept. We have prototyped the CUDA kernels so that they offer a high level of

performance on a broad scale by automatically adjusting the number of blocks and threads according to the input data's size. In order to increase the parallelism and achieve a higher throughput, we have applied a load balancing strategy and decomposition technique.

Even if in the literature the Legendre spectral-collocation method poses a great interest, none of the works so far (to our best knowledge) has researched and implemented optimization solutions for solving equations using the Legendre spectral-collocation method on the Maxwell architecture. Analyzing the obtained experimental results, we can conclude that the Maxwell architecture proves to be a novel approach for computing the numerical solution of second order partial differential equations with general boundary conditions.

R E F E R E N C E S

[1]. *F. Chen, J. Shen,* A GPU Parallelized Spectral Method for Elliptic Equations in Rectangular Domains, Journal of Computational Physics, Vol. 250, 555-564, 2013.

[2]. *T. Takahashi, T. Hamada*, GPU-accelerated boundary element method for Helmholtz' equation in three dimensions, International Journal for Numerical Methods in Engineering, 80(10):1295–1321, July 2009.

[3]. *C. Canuto,* Stabilization of spectral methods by finite element bubble functions. Comput. Methods Appl. Mech. Eng. 116, 13–26, 1994.

[4]. *C. Canuto, M. A. Hussaini, Quarteroni,, T. Zang,* Spectral Methods. Fundamentals in Single Domains, Springer Verlag, Heidelberg, 2006.

[5]. *R. Askey,* The 1839 paper on permutations: its relation to the Rodrigues formula and further developments, in *S. Altmann, E. Ortiz*, Mathematics and social utopias in France: Olinde Rodrigues and his times, History of mathematics 28, Providence, R.I.: American Mathematical Society, 2005.

[6]. *K. Jordaan, F.Tookos*, Convexity of the zeros of some orthogonal polynomials and related functions. J. Comput. Appl. Math. 233(3), 2009.

[7]. *G. H. Golub, C. F. van Loan,* Matrix Computations, 2nd Edition, The John Hopkins University Press, Baltimore, 1989.

[8]. *J. Shen, T. Tang, L. L. Wang*. Spectral Methods: Algorithms, Analysis and Applications, volume 41 of Springer Series in Computational Mathematics, 2011.