Test Harness on a Preconditioned Conjugate Gradient Solver on GPUs: An Efficiency Analysis

Antonio Wendell De Oliveira Rodrigues, Loïc Chevallier, Yvonnick Le Menach, Frédéric Guyomarch

Test Harness on a Preconditioned Conjugate Gradient Solver on GPUs: An Efficiency Analysis

A. Wendell de O. Rodrigues[1], Loïc Chevallier[2], Yvonnick Le Menach[2], and Frédéric Guyomarch[1]

[1]LIFL, Université de Lille1, 59655 Villeneuve d'Ascq, France
[2]L2EP, Université de Lille1, 59655 Villeneuve d'Ascq, France

The parallelization of numerical simulation algorithms, i.e., their adaptation to parallel processing architectures, is an aim to reach in order to hinder exorbitant execution times. The parallelism has been imposed at the level of processor architectures and graphics cards are now used for general-purpose calculation, also known as "General-Purpose computation on Graphics Processing Unit (GPGPU)". The clear benefit is the excellent performance over price ratio. Besides hiding the low level programming, software engineering leads to a faster and more secure application development. This paper presents the real interest of using GPU processors to increase performance of larger problems which concern electrical machines simulation. Indeed, we show that our auto-generated code applied to several models allows achieving speedups of the order of 10 ×.

***Index Terms*—Gradient methods, numerical simulation, parallel architectures, software engineering.**

I. Introduction

THE large computational power often required by solvers has been a limiting factor that justifies the use of parallel architectures such as the graphics-processing Unit (GPU) [1]–[3]. GPUs are coprocessors for the CPU. Indeed, they do not have autonomy and need a host that is usually CPUs. As a graphics processor, the GPU worked already as coprocessor relieving the processor from graphics tasks. Currently, as GPGPU, instead of CPUs doing the heavy parallel job, CPUs dispatch those tasks to GPUs. However, programming GPU is still complex. Thus, we have decided to assist specialists in algorithms of numerical simulations to create a code that runs efficiently on GPU architectures [4]. During our code generation phase, we present our methodology defining a GPU as a Hardware Processor with its own memory. We propose to distinctly separate CPU and GPU by defining their roles on higher level description. This paper shows, by multiple tests on large simulation models, the efficiency of an automatic generated code for the preconditioned conjugate gradient (PCG) within the Code_Carmel3D tool context.

II. Formulations

In our case, the classical dual formulations in terms of potentials are used to solve static and quasi static fields problems [5]. All these formulations are presented in Table I, where μ and σ are respectively the magnetic permeability and electrical conductivity, $\mathbf{A}$ and Ω are respectively the magnetic vector and scalar potentials, $\mathbf{T}$ and φ are respectively the electrical vector and scalar potentials. Finally, $\mathbf{J}_s$ and $\mathbf{H}_s$ are the source terms [6].

Moreover to simulate lot of applications it is necessary to introduce global quantities, such that the magnetic flux or electrical voltage, in the formulations [7], [8]. Formulations are then coupled with an equation which takes into account the global quantities to impose. In the case of an imposed magnetic flux between two surfaces of the studied $\mathcal{D}$-domain using the A-formulation, the system to solve becomes :

$$\begin{cases} \mathbf{rot}\frac{1}{\mu}\mathbf{rot}\mathbf{A} + \mathbf{rot}\frac{1}{\mu}\mathbf{rot}\mathbf{K}\phi = 0 \\ \int_{\mathcal{D}} \frac{1}{\mu}\mathbf{rot}(\mathbf{A} + \mathbf{K}\phi).\mathbf{N}d\tau = \varepsilon \end{cases} \quad (1)$$

where ε is the imposed magnetomotive force, ϕ is the magnetic flux to be calculated, $\mathbf{K}$ and $\mathbf{N}$ are defined in a similar way that in [6]. All the formulations can be coupled with a global quantity but the addition of the extra unknown can lead to filling out the last row of the matrix.

TABLE I
Formulations to Solve Static and Quasi Static Fields Problems

Formulation	Expression
$\mathbf{A}$	$\mathbf{rot}\frac{1}{\mu}\mathbf{rot}\mathbf{A} = \mathbf{J_s}$
Ω	$div(\mu\mathbf{grad}\Omega) = div(\mu\mathbf{H}_s)$
$\mathbf{A}$-φ	$\mathbf{rot}\mu^{-1}\mathbf{rot}\mathbf{A} + \sigma(\partial_t\mathbf{A} - \mathbf{grad}\varphi) = 0$
$\mathbf{T}$-Ω	$\mathbf{rot}\sigma^{-1}\mathbf{rot}\mathbf{T} + \mu\partial_t(\mathbf{T} - \mathbf{grad}\Omega) = 0$

III. GPU and OpenCL

A GPU is the many-core co-processor that comes in a graphics card. The original idea behind GPUs is to process and calculate which pixel is to be lit up at what instant of time. Since these calculations have to be very fast, the user does not feel any time delay in getting the required graphical output on the screen. This happens due to the high processing capacity achieved by these processors. Usually, it's possible to obtain between 10 and 100 times more processing speed than that of the CPU. The reason why the GPU has higher processing speed than the CPU is that the CPU is a generic processor intended for all kind of applications. It has other functions other than processing like cache control, storing of data, managing the other parts of the system along with many other functions (see Fig. 1). In the other hand, the GPU has basically one task that is to perform calculations. Hence, due to its singularity function, the processing speed of the GPU is very high. Currently, it is usual the use of a GPU together with a CPU to accelerate

Manuscript received November 08, 2012; accepted January 18, 2013. Date of current version May 07, 2013. Corresponding author: Y. Le Menach (e-mail: yvonnick.le-menach@univ-lille1.fr).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TMAG.2013.2243830

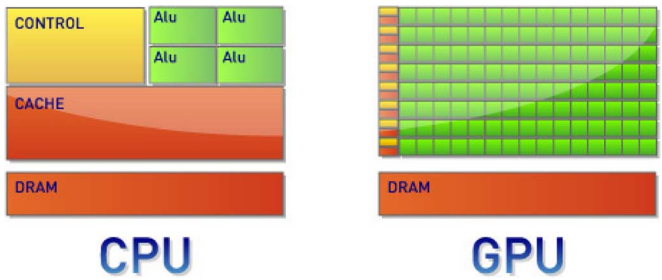

Fig. 1. CPU × GPU Architectural Differences.

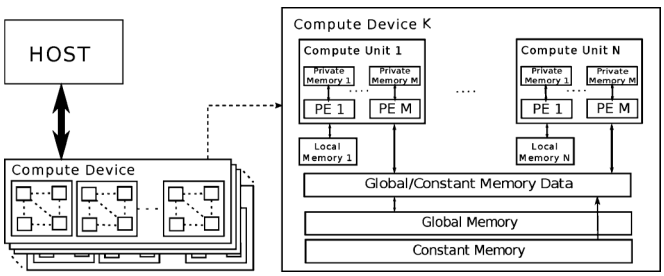

Fig. 2. OpenCL platform and memory models.

general-purpose scientific and engineering applications (a.k.a. GPGPU). Again, as seen in Fig. 1, 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. Serial portions of the code run on the CPU while parallel portions run on the GPU. Hence, this combination allows us to develop parallel applications with common and less expensive available hardware.

Proposals, such as OpenCL[1], have been designed to exploit the parallel programming on GPUs. OpenCL is a standard for parallel computing consisting of a language (an extension of C), API, libraries and a runtime system. OpenCL is based on a platform model that divides a system into one host and one or several compute devices. Compute devices act as co-processors (e.g. GPUs) to the host (e.g. CPU). An OpenCL application is executed in the host, which sends instructions, defined in special functions called kernels, to the device. A single host can manage multiple devices, even heterogeneous devices. OpenCL allows for creating contexts and queues in order to manage tasks being launched by the host in all attached devices. Fig. 2 shows the main elements of the platform and memory models of OpenCL. The high parallelism usually achieved is mainly function of the high number of *processor elements* (PE) and the memory hierarchy which allows for faster data access.

IV. MDE and Parallel Solvers

In [4], we aimed to generate an effective code for GPU from a new branch of a development environment based on model driven engineering (MDE). MDE allows us to develop software from high-level specification models. The core of our code generation approach lies mainly in the model transformations. We have defined several model transformations modules that now, along with other ones, are part of the Gaspard2 Model Transformation Library [9]. Choosing the suitable transformations modules is part of the compiling engineering process. As an MDE approach, the new branch proposed for Gaspard2 comprehends all models, metamodels, transformation modules, and, foremost, how to determine the compiling process layers in order to achieve all necessary model element analysis. During the application design, developers specify the main concerns of the application using unified modeling language (UML). Then, using transformation chains, a source code is generated for the chosen target platform. The main advantages of this approach are that they clearly distinguish the hardware components from the software components, and describe the potential parallelism of applications. This methodology can be applied to potentially parallel algorithms such as PCG and insert them into a general context of simulation tools. Thus, physicists can develop parallel applications without having in-depth knowledge about software and hardware issues.

Globally, the model designers (e.g., the physicists), in order to implement an application, follow the steps as described below. This phase is based on directives defined by MARTE [10] and Gaspard2.

1) They define application and architecture models. At first, there is no link between both models. Thus, it is possible to divide this step into two parts executed by different teams or experts.
2) They place every task and data onto hardware architecture elements. Moreover, at this moment the designer associates Intellectual Property (IP)[2] to each elementary task in the application model.

We have worked with high-level abstraction models of numerical methods of simple problems such as an electric field induced by a changing magnetic field or more complex problems such as the simulation of electrical machines (e.g., automotive alternators). After designing the model of such methods, we generate OpenCL, compile it, then we have a ready-to-use parallel specialized function.

V. Application

From the generated code, we have the PCG as a solver module for Code_Carmel3D. The idea behind the model that we will provide is that it will replace the original solver written in Fortran90 by a GPU solver in OpenCL. This process is a simple step in the compilation process of Code_Carmel3D.

Once having a simulation tool that includes a parallel solver running on GPUs, we are able to start our testbed. The GPU speedup, i.e., ratio of CPU over GPU computational time, is computed on 141 models spanning a wide range of problem sizes, i.e., size of the linear system to solve as number of non-null elements in the system sparse matrix, in both static (A and Ω) and dynamic ($A-\varphi$ and $T-\Omega$) formulations for harmonic (complex values) and time-domain (real values) problems. We retained 134 models from our unit test catalog; one academic cube model for six various mesh sizes and one realistic alternator machine (see Fig. 3).

Cube and alternator models are static (A-formulation) harmonic problems. The hardware used in this testbed is composed

[1]www.khronos.org/opencl

[2]Piece of code that implements atomic or elementary functions.

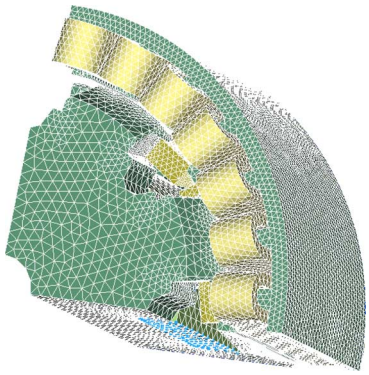

Fig. 3. Mesh of the alternator machine model, made from 682,358 tetrahedra.

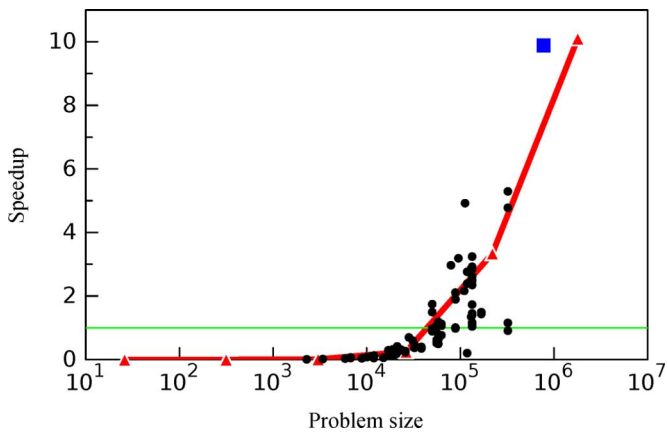

Fig. 4. GPU speedup against CPU as a function of the problem size for unit-test (small diamonds), cube (solid line with triangles) and alternator machine (square) models. The speedup equal to 1 line appears as a reference.

by four 2.4 GHz 8-cores AMD Opteron 6136 processors driving a S1070 unit (4 NVIDIA Tesla T10 GPUs).

VI. Results

Computation is made on one GPU core only. In every test, GPU results presented good agreement with CPU results, i.e., relative differences between solutions or physical quantities (total magnetic energy) are less than 0.01%. GPU speedup tends to increase with the problem size. Due to GPU setup and communication times, the speedup is greater than one typically for large problems, i.e., with at least 60,000 degrees of freedom, and reaches the value 10.1 for the alternator machine model (see Fig. 4). Using GPU computations is not useful when GPU speedup equals 1, as CPU computation time is weak, i.e., one second typically.

From Fig. 4, it clearly appears that unit-test models do not follow a simple relationship between speedup and the problem size, i.e., several values of speedup are found for a given value of the problem size. We will focus hereafter on the four unit-test models whose problem size, approximately equal to 500,000, is the highest (see Fig. 4). These four models describe the same magneto-static physical apparatus, i.e., a rectangular rod in which magnetic permeability varies with space, which is solved with the A-formulation. The source is a magnetic field which is imposed in the rod through two of its opposite faces, either imposing the magnetic flux (named hereafter FLUX) or

TABLE II
Results for the Rectangular Rod Model (Unit Test Catalog)

		CPU		GPU	
Model	Speedup	Nb. iter.	Time/s	Nb. iter.	Time/s
FLUX, UNV	5.29	659	68.03	683	12.85
FLUX, MED	4.78	586	55.26	604	11.57
DDPM, MED	1.15	699	81.38	722	70.53
DDPM, UNV	0.91	758	68.51	774	75.39

Each model is described by its source (FLUX or DDPM) type and mesh storing (MED or UNV) format. Results from the solver, both using CPU and GPU, are given, *i.e.*, the number of iterations to reach the 10^{-12} accuracy on the residual of the linear system and the computational time (in seconds). Speedup is also provided.

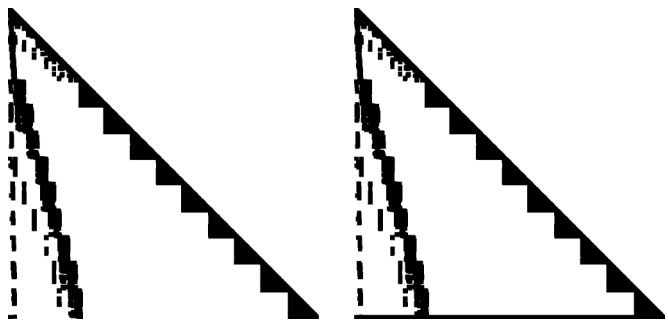

Fig. 5. Graphical view of the non-null (black points) elements in the linear system sparse matrix for the FLUX (left) and DDPM (right) models.

imposing the magnetomotive force (named hereafter DDPM). Mesh is made from 288,092 tetrahedra and is stored both in MED[3] and UNV[4] equivalent formats, which are supported by Code_Carmel3D. Table II details the results for the four possible combinations of source and format.

Fig. 4 shows that models are separated in two, FLUX and DDPM, groups and characterized by a significant difference in speedup. Comparison between (FLUX, UNV) and (DDPM, UNV) models shows a huge difference, by a factor nearly equal to 6, on GPU times, due to the linear system matrix (see Fig. 5). The DDPM matrix has an extra full row as compared to the FLUX matrix, as stated (see Section II). This full raw is not suitable for parallelization in this context of matrix-vector products as, when made on a single GPU core, it is much slower than other block matrix-vector products made on separate GPU cores. Parallelizing this full row matrix-vector product is possible but not efficient, due to the time spent in order to gather all results. As a consequence GPU computational time stretches out much more than CPU time, for which the DDPM model is also more difficult to solve.

For a given, FLUX or DDPM, source, MED and UNV models should give the same results as meshes are identical. CPU computation time for FLUX UNV model is 23.1% higher than the FLUX MED corresponding result. However the number of iterations is not the same. This 12.5% difference may be due to the fact that numbering of elements is different between MED and UNV formats. CPU computation time is proportional to the

[3]MED is a binary-coded format for storing finite element meshes and results (http://www.code-aster.org/outils/med/).

[4]UNV is a text-based format used by the formerly I-Deas, now Siemens NX, software (http://www.plm.automation.siemens.com/en_us/products/nx/ideas/).

number of iterations with a very good accuracy. Using this linear relation, the interpolated CPU computation time for the FLUX UNV model at the same, i.e., 586, number of iterations as the FLUX MED model is 60.49s, i.e., 9.5% higher than the CPU computation time for the FLUX MED model. This is a significant difference still. This difference is even higher when comparing DDPM models using CPU, as computation time for the MED model is higher than the one for the UNV model when the number of iterations is in the reverse order. Applying a proportional relation to these models, the interpolated CPU computation time for the DDPM UNV model at the same, i.e., 699, number of iterations as the DDPM MED model is 63.16s, i.e., 28.8% lower than the CPU computation time for the DDPM MED model. On the other side GPU computation time scales well with the number of iterations. We checked that a linear, but not proportional due to the GPU setup time approximately equal to 5s, relation is still valid between the computation time and the number of iterations. Applying this relation, we note a weak, i.e., 3.2%, difference in computation time for FLUX MED and UNV models. This difference is even weaker, i.e., 0.2%, when comparing GPU-computed DDPM MED and UNV models. We also note that, for a given model, we hardly reproduce the CPU computation time with various tries, i.e., a 33% variation around the mean value for the FLUX MED model and a 37% variation for the DDPM MED model, over a few dozens of tries. This is surprising as the number of iterations is not changing with tries. Looking at CPU load during these tries shows that the used CPU changes during the iterative resolution task. On the other side, the GPU computation time was reproduced with a very good accuracy, i.e., a 1.6% variation for the FLUX MED model. Looking at CPU load during these tries shows that the used CPU does not change, as only one CPU is used in order to pilot the GPU core. After computing the FLUX MED model on a *fixed* CPU[5] again, the CPU computation time was reproduced with a good accuracy, i.e., a 2% variation.

VII. Discussion

The authors think that the CPU load-balancing effect detailed, in Section VI, explains the high variation met both in CPU computation time for the same model but the mesh storage format. This is a well-known effect. After dozens of tries, CPU computation time was seen to vary between 51 and 71s, and between 62 and 90s, for the FLUX and DDPM MED models, respectively. Results from Table II are all in agreement with the ranges stated above. From these ranges and using the GPU computation times from Table II, we infer that the speedup value could vary from 4.41 to 6.21, i.e., a 34% variation around the mean value, for every FLUX model; and from 0.88 to 1.29, i.e., a 38% variation, for every DDPM model.

VIII. Conclusion

This work states the robustness of our implementation in the Code_Carmel3D tool tested on several models. Moreover our automatic generated code from high-level specification of algorithms applied to very large models (exceeding 60,000 degrees of freedom) achieves good performances provided that the pattern of the linear system to be solved is sparse and does not contain full rows. Also the ordering of the elements could have a significant impact on the speedup. A preliminary analysis of the pattern of the matrix is required to guarantee that these requirements are satisfied. A deep insight into the numerical behaviour of the noted speedup significant variation at constant problem size was conducted. The authors think that this behaviour is fully explained by the three effects which are detailed, as the CPU automatic load-balancing. In summary, the results presented here confirm a twofold purpose: the automatic generated code has high efficiency; those larger problems that usually demand more time to complete offer better speedups with fully sparse linear systems.

[5]CPU is fixed by the *taskset* Linux command.

Acknowledgment

This work was supported by the MEDEE project with financial Assistance of European Regional Development Fund and the region Nord-Pas-de-Calais. The authors would like to thank T. Henneron and J. Korecki for a fruitful discussion on results and on the way to display sparse matrices.

References

[1] N. Gödel, N. Nunn, T. Warburton, and M. Clemens, "Scalability of higher-order discontinuous galerkin FEM computations for solving electromagnetic wave propagation problems on GPU clusters," *IEEE Trans. Magn.*, vol. 46, no. 8, pp. 3469–3472, Aug. 2010.

[2] D. M. Fernandez, M. M. Dehnavi, W. J. Gross, and D. Giannacopoulos, "Alternate parallel processing approach for FEM," *IEEE Trans. Magn.*, vol. 48, no. 2, pp. 399–402, Feb. 2012.

[3] I. Kiss, S. Gyimothy, Z. Badics, and J. Pavo, "Parallel realization of the element-by-element FEM technique by CUDA," *IEEE Trans. Magn.*, vol. 48, no. 2, pp. 507–510, Feb. 2012.

[4] A. W. O. Rodrigues, F. Guyomarc'h, J. Dekeyser, and Y. Le Menach, "Automatic multi-GPU code generation applied to simulation of electrical machines," *IEEE Trans. Magn.*, vol. 48, no. 2, pp. 831–834, Feb. 2012.

[5] Z. Ren and A. Razek, "Comparison of some 3d eddy current formulations in dual systems," *IEEE Trans. Magn.*, vol. 36, no. 4, pp. 751–755, Jul. 2000.

[6] Y. Le Menach, S. Clenet, and F. Piriou, "Numerical model to discretize source fields in the 3d finite element method," *IEEE Trans. Magn.*, vol. 36, no. 4, pp. 676–679, Jul. 2000.

[7] T. Henneron, S. Clenet, and F. Piriou, "Calculation of global quantities using incidence matrixes in the a-phi formulation," in *Proc. 6th Int. Conf. CEM*, Apr. 2006, pp. 1–2.

[8] P. Dular, W. Legros, and A. Nicolet, "Coupling of local and global quantities in various finite element formulations and its application to electrostatics, magnetostatics and magnetodynamics," *IEEE Trans. Magn.*, vol. 34, no. 5, pp. 3078–3081, Sep. 1998.

[9] A. Gamatié, S. Le Beux, E. Piel, R. Ben Atitallah, A. Etien, P. Marquet, and J. Dekeyser, "A model driven design framework for massively parallel embedded systems," *ACM Trans. Embedded Comput. Syst.*, vol. 10, no. 4, pp. 39:1–39:36, Nov. 2011.

[10] Object Management Group, UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems [Online]. Available: http://www.omg.org/spec/MARTE/1.1 2011