Performance Evaluation and Analysis for Conjugate Gradient Solver on Heterogeneous (Multi-GPUs/Multi-CPUs) platforms

Najlae Kasmi[†*] , Mostapha Zbakh[†], Sidi Ahmed Mahmoudi[††] *and* Pierre Manneback[††]

[†]ENSIAS, Mohammed V University of Rabat, Morocco, [††]University of MONS, Faculty of engineering
20, Place du Parc, Mons, Belgium

Summary
High performance computing (HPC) presents a technology that allows solving high intensive problems in a reasonable period of time, and can offer many advantages for large applications in various fields of science and industry. Current multi-core processors, especially graphic processing units (GPUs), have quickly evolved to become efficient accelerators for data parallel computing. They can maintain parallel programmability and provide high computing throughput. In this paper, the authors present an implementation and performance analysis of sparse iterative linear solver on heterogeneous multi-CPUs/multi-GPUs architectures using PARALUTION and StarPU libraries. More particularly, the authors compare the performance of parallel preconditioned conjugate gradient (PCG) solver on different platforms. Experimental results have been conducted using GPU platforms and show a significant speed up compared to central processing units CPUs implementations. In order to provide the highest performance, the system supports Multi-CPU/Multi-GPU architectures, where it scales up very high.

Key words:
HPC, Multi-GPUs/Multi-CPUs architectures, Sparse linear systems, PARALUTION, StarPU

1. Introduction

Accelerating HPC applications is currently under extensive research within new hardware technologies such as the recent CPUs that dispose of multiple processor cores for parallel computing [**1**], [**2**] or, GPUs that can process huge data blocks in parallel [**3**]. Hybrid computation (using CPUs and GPUs) is a common solution for supercomputers, desktop computers [**4**] and field-programmable gate arrays (FPGAs). This heterogeneous computation allows exploiting the full power of new hardware, required form several applications [**5**], [**6**].

GPUs are getting more attention than other HPC accelerators [**7**] due to their high computational power, strong performance, functionality and low price. The modern GPU is a highly parallel programmable processor featuring peak arithmetic and memory bandwidth [**8**]. GPUs are used to accelerate 2D/3D graphic rendering and general applications with high data parallelism known as general purpose graphic processing unit (GPGPU). GPU programming has been easier within the application programming interfaces (APIs) such as compute unified device architecture (CUDA) and open computing language (OpenCL) [**9**].

In this context, the authors evaluate the performance of an iterative algorithm for solving sparse symmetric positive definite linear systems, the conjugate gradient (CG) method, on heterogeneous platforms (multi-CPUs/multi-GPUs). In conjunction with an appropriate preconditioner (PCG), the method has proven its efficiency in a wide spectrum of applications [**10**]. The goal of CG algorithm [**11**] is to solve large sparse linear systems of equations that have the form of

$$Ax = b$$

where A is a sparse matrix and b is the unit vector.

The authors investigate the influence of employing GPUs to accelerate PCG using PARALUTION library [**12**] with CUDA and OpenCL APIs [**9**]. This allows us to confirm that the GPU implementation of PCG is more efficient and the CUDA platform is more suitable. Moreover, the StarPU runtime [**13**] is used for exploiting heterogeneous architectures (multi-GPUs/multi-CPUs) within different schedulers [**13**].

To sum up, the objective of this study is to exploit effectively hybrid platforms in order to improve the performance of PCG solver.

The rest of the paper is organized as follows. Section II, reviews some related literature works. Section III gives a brief introduction of GPU architectures and CUDA programming model. Section IV describes the conjugate gradient method with its most costly operation, the sparse matrix vector product (SpMV). In Section V, the authors present the evaluation of SpMV and PCG computing performance obtained with CPU, Multi-CPU and GPU platforms using PARALUTION and StarPU. Then, the overall results and performances are discussed within the analysis of influence of the applied optimizations. Finally, authors draw conclusion in section VI.

Manuscript received August 5, 2017
Manuscript revised August 20, 2017

2. Related Works

The Conjugate Gradient method (CG) [**14**] is an iterative algorithm for solving a system as shown in equation (1)

$$Ax = b \quad (1)$$

Where $x, b \in R^n$ and A is symmetric, positive define matrix.

CG solver involves recurrence relationships, so it is convenient to use $x_1, x_2, \ldots,$ to denote successive iterates. The authors denote the unique solution of this system by x^*, Starting with x_0 they search for the solution and in each iteration authors need a metric to tell whether they are closer to the solution x^*. This metric comes from the fact that the solution x^* is also the unique minimizer of the quadratic function (2):

$$f(x) = \frac{1}{2} x^T A x - x^T b \qquad x \in R^n \qquad (2)$$

It is also convenient to define a residual $r(x) = b - Ax$ to calculate the unknown $x_1, x_2, \ldots.$. The algorithm is described in detail by Hestenes and Stiefel [**15**] and its principal steps are drawn in Algorithm 1.

Algorithm 1 *Conjugate Gradient*

```
$r_0 = b - Ax_0$;
$p_0 = r_0$;
$k = 0$;
Repeat
    $\alpha_k = r_k^T r_k / p_k^T A p_k$;
    $x_{k+1} = x_k + \alpha_k p_k$;
    $r_{k+1} = r_k - \alpha_k A p_k$;
    if $r_{k+1}$ is small then exit loop
    $\beta_k = r_{k+1}^T r_k / r_k^T r_k$
    $p_{k+1} = r_{k+1} + \beta_k p_k$;
end repeat
The result is $x_{k+1}$
```

Where, the input vector x_0 can be an approximate initial solution or 0.

A variant of the PCG algorithm on CPU platform is presented by Demmel and al. in [**16**] in order to facilitate data transfers overlapping with computation

Another variant is proposed by Chronopoulos and Gear in [**17**], that improves data locality and parallelism in PCG solver.

Different implementations of the CG method for GPUs have been published in the last years. With the advent of CUDA studies on iterative solvers, new implementations of PCG on GPU architecture have been presented in [**18**] using different preconditioners, such as squared polynomials, incomplete Cholesky factorizations or symmetric successive over relaxation smothers which allows to decrease the number of iterations and improve the resolution of PCG solver.

Authors in [**19**] improved the performance of the SpMV kernel on an Nvidia Tesla GPU (C1060) by a factor of up to 25% on average by using a methodology to choose the right data structures to represent sparse matrices then they exploited their SpMV in the conjugate gradient method and showed an average of 20% improvement of CG performance compared to using a standard SpMV.

GPU implementation of the CG sparse solver is presented also by Bolz and al [**20**]. This method relies on the programmable graphics pipeline of modern GPUs and was implemented using fragment shaders. Authors show that reduction operators would benefit greatly from a few globally writable registers. Limiting such registers to commutative operators would avoid troublesome order dependencies. They noticed that the authors could simplify reductions by allowing borders on floating point textures. Their results show that the performance of PCG solver could be boosted with use of texture memory by 20%.

A CG implementation comparison between CPU and GPU platforms is made in [**21**], where the CG method is implemented on Woodcrest CPUs architecture, on the one hand, and NVidia 8800GTX GPUs architecture, on the other hand. In this work authors propose a stream model for arithmetic operations on vectors and matrices that exploits the intrinsic parallelism and efficient communication on modern GPUs. The authors report a speedup of about 3 times when using their model on the 8800GTX GPU.

The use of multi-GPU platforms on the CG can be found in [**22**], where the main work is focused on minimizing the communication overhead through overlapping techniques. The authors propose an online auto tuning approach which does not only take into account characteristics of the machine and the input data, but also considers influences that are changing during the simulation which allows minimizing total execution time of the CG solver.

In order to well distribute the workload between CPU and GPU nodes in an optimal way, an "execution time and energy model" is developed in [**23**]. This model allows improving the performance of PCG solver on the heterogeneous CPU/GPU platform with respect to the execution time or the energy consumption.

Another work, which is similar to this one, is presented in [**24**] where the authors use StarPU runtime system to perform the CG solver in heterogeneous CPU-GPU architecture. The difference is that the authors of this work have integrated an ILU preconditioner and they take advantages of texture and shared memories of GPU in order to have better performance. Authors compare with PARALUTION [**25**], library that is a recent one for solving sparse systems of linear equations.

According to the works mentioned above the contributions are offering another perspective of PCG implementation on CPU, GPU and CPU/GPU architecture. The authors are proceeding as follows:

- ✓ Evaluation of PCG performance on GPU using PARALUTION library within CUDA and OpenCL. Indeed, the authors analyzed in detail the execution time of conjugate gradient algorithm for three storages formats CSR, ELL and HYB. This benchmarking allowed us to note that CUDA platform is more efficient, and that the CSR storage format is the most adapted for the testbed.
- ✓ Heterogeneous implementation of PCG solver using StarPU runtime system. Within this implementation, the authors take advantage of the hybrid system (Multi-CPUs/Multi-GPUs) in order to boost the performance of PCG algorithm. The best combination (that offers better performance) of GPUs and CPUs cores is chosen automatically.

3. GPU Architecture and CUDA Programming

This section describes GPU architecture and CUDA programming model. Authors explain the management of threads and memories on GPUs and present examples of CUDA functions.

3.1 GPU Architecture

GPU is considered as an extensible array, which consists of multi-threaded Streaming Multiprocessors (SMs) [**26**], composed of multiple Scalar Processor (SP) cores. To manage the distribution of threads, the multiprocessors use a Single Instruction Multiple Threads (SIMT) model [**27**], [**28**]. Each thread is mapped into one SP core and executes independently with its own instruction address and register state [**26**].

The NVIDIA GPU memory can be classified as follows: Global memory, Texture memory, Constant memory, Shared memory, Local memory and Registers (Figure 1). Global memory corresponds to the mass memory of the graphic processor accessible by the CPU. The capacity of the memory varies from a few hundred MB to a few GB (depending on the graphic card) and has a much higher latency (100 to 300 clock cycles for the NVIDIA GPUs) compared to the shared memory (4 clock cycles). Global memory is not cached, so it is important to follow the right access pattern to achieve good memory bandwidth [**29**].

All modern CUDA cards (Fermi architecture and later) have a coherent L2 Cache. The GPU's L2 cache memory is smaller than L2 and L3 cache memory of CPU, but has higher bandwidth available [**26**]. The L1 cache onboard a GPU is smaller than L1 cache in a CPU, but also has much higher bandwidth. High-end NVIDIA graphic cards have several streaming multiprocessors, or SMs, each is equipped with its own L1 cache [**29**].

The threads are organized in warps. A warp is defined as a group of 32 threads of consecutive thread IDs. A half-warp is either the first or second half of a warp [**30**]. The most efficient way to use the global memory bandwidth is to coalesce the simultaneous memory accesses by threads in a half-warp into a single memory transaction.

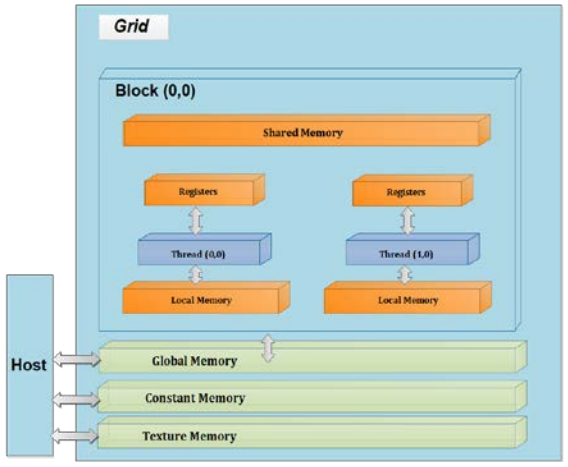

Fig. 1: GPU memory architecture

3.2 CUDA Programming

Compute unified device architecture (CUDA) is a parallel programming model and computing platform invented by NVIDIA [**26**]. A program on the host (CPU) can call a GPU to execute CUDA functions called kernels. See example in Table 1.

The authors define a kernel using the __global__ declaration specifier with the number of CUDA threads that execute the kernel. This kernel is specified by using a new <<<numBlock, threadsPerBlock>>> execution configuration syntax [**31**].

Table 1: CUDA Function Declarations

Function	***Executed on Only***	***callable from***

__device__ float DeviceFunc ()	Device	Host
__global__ void KernelFunc ()	Device	Host
__host__float HostFunc ()	Host	Host

The same parallel kernel is executed by many threads, which are organized into thread blocks, and distributed to SMs and split into warps scheduled by SIMT units. All threads in the same thread block share the same shared memory of size 32 KB (or 96 KB) [**32**] and can synchronize themselves with a barrier. Threads in a warp execute one common instruction at a time. This is specified as warp-level synchronization [**26**]. Full efficiency is achieved when all 32 threads of a warp follow the same execution path. Branch divergence causes serial execution.

4. Conjugate Gradient Method And Assiciated Libraries

In this section, the authors first present the Preconditioned Conjugate Gradient (PCG) algorithm then they introduce PARALUTION library and StarPU runtime system with different scheduling techniques.

4.1 Conjugate gradient method

The conjugate gradient method was initially developed by Hestenes and Stiefel [**14**]. It is one of the well-known iterative methods to solve large sparse linear systems. In addition, it can be adapted to solve nonlinear equations and optimization problems. However, it can only be applied to problems dealing with positive definite symmetric matrices.

The main idea of the CG method is the computation of a sequence of approximate solutions {xk}k≥0 in a Krylov subspace of order k shown in equation (3):

$$x_k \in x_0 + K_k(A, r_0) \qquad (3)$$

In such a way that the Galerkin condition (4) must be satisfied:

$$r_k \perp K_k(A, r_0) \qquad (4)$$

Where x_0 is the initial guess, $r_k = b - Ax_k$ the residual of the computed solution x_k , and K_k the Krylov subspace of order *k* which is defined in (5):

$$K_k(A, r_0) \equiv span\{r_0, Ar_0, A^2r_0, \ldots, A^{k-1}r_0\} \qquad (5)$$

In fact, CG is based on the construction of a sequence $\{p_k\}_{k\in N}$ of direction vectors in K_k wich are pairwise A-conjugate (A-orthogonal) as noted in (6):

$$p_i^T A p_j = 0, \quad i \neq j \qquad (6)$$

At each iteration k, an approximate solution x_k is computed by recurrence as shown in equation (7):

$$x_k = x_{k-1} + \alpha_k p_k, \quad \alpha_k \in \mathbb{R} \qquad (7)$$

Consequently, the residuals r_k are computed in the same way as noted in (8):

$$r_k = r_{k-1} - \alpha_k A p_k \qquad (8)$$

In the case where all residuals are nonzero, the direction vectors p_k can be determined so that the following recurrence holds as shown in equation (9):

$$p_0 = r_0, \quad p_k = r_k + \beta_k p_{k-1}, \quad \beta_k \in \mathbb{R} \qquad (9)$$

Moreover, the scalars $\{\alpha_{k>0}\}$ are chosen to minimize the A-norm error $\| x^* - x_k \|_A$ over the Krylov subspace K_k, and the scalars $\{\beta_{k>0}\}$ are chosen to ensure that the direction vectors are pairwise A-conjugate. The assumption that matrix A is symmetric and the recurrences allow the deduction of (10):

$$\alpha_k = \frac{r_{k-1}^T r_{k-1}}{p_k^T A p_k}, \qquad \beta_k = \frac{r_k^T r_k}{r_{k-1}^T r_{k-1}} \qquad (10)$$

The CG method’s convergence rate relies on the distribution of eigenvalues of a coefficient matrix **A** or a preconditioned matrix $A^{\sim}$. The condition number of a matrix is a simpler indicator that allows having the complete information of eigenvalues that enables prediction of the exact convergence behaviour. This number is defined by equation (11):

$$cond(\mathbf{A}) = \|\mathbf{A}\|\|\mathbf{A}^{-1}\| \qquad (11)$$

There are various definitions of the norm $\|.\|$, which allows defining various condition numbers. These two formulas (12) and (13) are commonly used.

$$\|A\| = max_j \sum_{i=1}^{n} |a_{i,j}| \qquad (12)$$

Or

$$\|A\| = \left(\sqrt{maximumeignvalue\ of\ A^T A}\right) \qquad (13)$$

Moreover, if the matrix **A** is symmetric positive define (SPD), then the condition number given by the second norm can be giving by equation (14):

$$\text{cond(A)}=\frac{maximum\ eigenvalue\ of\ A}{minimum\ eigenvalue\ of\ A} \qquad (14)$$

Preconditioning is an important technique used to develop an efficient conjugate gradient method solver for challenging problems in scientific computing [**33**]. The technique comes in the picture when the authors try to solve a linear system with a very large condition number [**34**]. The idea behind preconditioning is using the CG on an equivalent system. Thus, instead of solving Ax=b the authors solve a related problem for which A is chosen such that its condition number is closer to one. Note that "very large condition number (C)" means roughly log(c) $\geq$ the precision of matrix entries

Algorithm 2 shows the key points of the preconditioned CG method [**35**]. ε is the convergence tolerance threshold, maxiter is the maximum number of iterations, and an (_;_) defines the dot product between two vectors in $\mathbb{R}^n$. At each iteration, a direction vector p_k is determined, so that it is orthogonal to the preconditioned residual z_k and to the direction vectors $\{p_i\}_{i<k}$ previously determined (from line 8 to line 13). Then, at lines 16 and 17, the iterate x_k and the residual r_k are computed using formulas (7) and (8), respectively. The CG method converges after, at most, n iterations. In practice, the CG algorithm stops when the tolerance threshold ε and/or the maximum number of iterations maxiter is reached.

Algorithm 2 Parallel preconditioned CG method [**35**]

```
1. Choose an initial guess $x_0$;
2. $r_0 = b - Ax_0$;
3. convergence = false;
4. $k = 1$;
5.  repeat
6.  $z_k = M^{-1} r_{k-1}$;
7.  $\rho_k = (r_{k-1}, z_k)$;
8.      if $k = 1$ then
9.          $p_k = z_k$;
10.     else
11.         $\beta_k = \rho_k / \rho_{k-1}$;
12.         $p_k = z_k + \beta_k p_{k-1}$;
13.     end if
14. $q_k = Ap_k$;
15. $\alpha_k = \rho_k / (p_k, q_k)$;
16. $x_k = x_{k-1} + \alpha_k p_k$;
17. $r_k = r_{k-1} - \alpha_k q_k$;
18.     if $(\rho_k < \varepsilon)$ or $(k \geq maxiter)$ then
19.        convergence = true;
20.     else
21.        $k = k + 1$;
22.     end if
23.   until convergence
```

4.2 Associated Libraries and schedulers

4.2.1 StarPU Runtime System

StarPU [**13**] proposes a task-based programming paradigm where the algorithm is expressed as a Directed Acyclic Graph (DAG), vertices representing tasks and edges representing dependencies between them. The DAG does not need to be entirely provided, but the sequence of tasks to be executed is provided dynamically as well as the set of data and the access mode (read, write, read-write modes) onto which those tasks operate. Based on this information, the dependencies are implicitly computed by the runtime system. In this study, tasks are performed on a different processing unit (GPUs and CPUs). The submission of a task is a non-blocking operation so that multiple tasks may be processed concurrently. The runtime system then performs the actual execution of that task only once the related dependencies are satisfied and that the appropriate data has been transferred on the required processing unit [**24**]. StarPU thus ensures both data consistency and transfers between processing units.

4.2.2 PARALUTION Library

PARALUTION [**25**] is an open source C++ library used to solve sparse systems of linear equations, developed at Uppsala University in Sweden. It offers a variety of iterative solvers such as the CG, BiCGStab and GMRES Krylov method [**36**], [**14**] and preconditioners based on additive (Jacobi, Gauss - Seidel) and multiplicative (ILUp, ILUT, enhanced multi-colored ILUmethod) [**21**] splitting schemes as well as approximate inverse preconditioning approaches (Figure 2). It also features different matrix storage formats, which are crucial for GPU internal bandwidth exploitation. Furthermore, PARALUTION offers several hardware backends for execution on multi/manycore CPU and GPU devices [**25**]. Currently, multi-core CPUs and GPUs are supported by PARALUTION, this offers the possibility to switch between different architectures without modifying any line of existing code and thus to exploit the available computational power of many computer systems [**37**]. This is achieved within PARALUTION by hiding all hardware relevant details from the user while maintaining optimal usage of the resources.

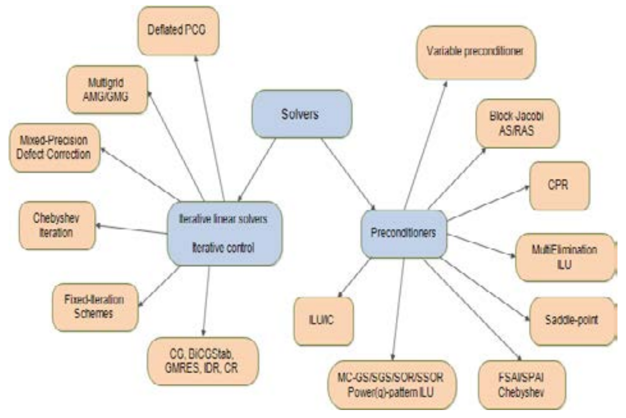

Fig. 2: Detail of the PARALUTION library architecture [**25**]

During the past years, a variety of different libraries for solving large sparse linear systems have been developed [**25**]. While most of them support highly parallel architectures multi- GPUs or multi-CPUs, they are restrictive in terms of portability [**25**], [**38**]. However, past evolution showed that hardware architectures have a very limited life cycle. Therefore, it is very important to maintain good flexibility of solution techniques and software in the current computing landscape [**39**]. PARALUTION, as one of those frameworks [**25**], offers the addressed flexibility while fully exploiting all available resources. It is under constant development and features most of the latest hardware backends. Therefore the authors choose to work with this library in order to implement and evaluate the conjugate gradient solver.

5. Experimental Results and Discussion

In this section, the authors describe briefly the hardware setup used for the experiments, and then they evaluate the performance of Preconditioned Conjugate Gradient algorithm with MultiColoredILU preconditioner on GPU using PARALLUTION library and on heterogeneous architecture (Multi GPUs/Multi CPUs), using StarPU runtime system. Finally, the authors present the result of their comparisons with discussion.

5.1 Hardware and software setup

The implementation environment is based on three NVIDIA GTX580 GPUs with 3GB memory and 512 CUDA Fermi cores 780 MHz. This platform is equipped with Intel(R) Core i7 CPU (12-core, 3,33 GHz) and 24 Go RAM, running on Ubuntu Linux. PARALUTION library and StarPU runtime system has been compiled using CUDA 7.5 version.

5.2 Experiment results

The authors use a set of matrices from MATRIX MARKET[1] cited in Table 2 where Row is the number of row elements and entries are the number of non zero elements. They selected sparse matrices with variable dimensions, to perform a fair and comprehensive evaluation of the results.

The authors implement the preconditioned conjugate gradient algorithm with MultiColoredILU preconditioner on GPU using CUDA and OpenCL platforms and on CPU using OpenMP, then they evaluate the related performance using PARALLUTION library. The authors use StarPU runtime system for the same implementation with the same preconditioner to compare the execution time between PCG on hybrid architecture (Multi GPUs/Multi CPUs) on the one hand, and over a multiple cores or a GPU on the other hand.

[1] http://math.nist.gov/ MatrixMarket /

Table 2: Storage matrix format from the Matrix Market repository

Matrix	Entries	Row (n)
G2_circuit	726 674	150 102
apache2	4 817 870	715 176
tmt_sym	5 080 961	726 713
ecology2	4 995 991	999 999
thermal2	8 580 313	1 228 045
G3_circuit	7 660 826	1 585 478

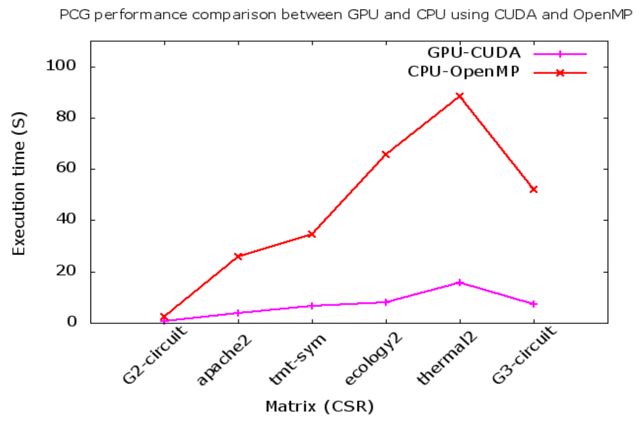

Fig. 3 GPU and CPU performance of CG using CUDA and OpenMP platforms

The authors can notice that the efficiency of GPU is related to the storage format. HYB format, a combination of ELL and COO sparse matrix data structures, outperforms in a majority of cases.
In figure 3, they compare the performance of PCG solver in term of execution time between CPU and GPU versions using OpenMP and CUDA platforms. Figure 4 shows the performance evaluation of CG algorithm on GPU using two different platforms CUDA and OpenCL.

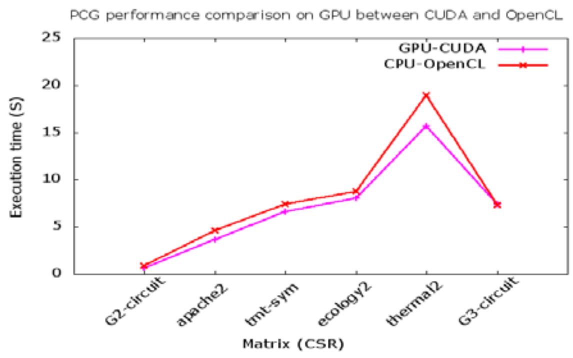

Fig. 4 GPU and CPU performance of PCG using CUDA and OpenCL platforms

The result shows that the execution time of PCG solver on GPU using CUDA platform is 5x faster than the execution time on CPU using OpenMP especially for thermal2 matrix due to the high level of density. This result confirms what Cevahir and Nukada found in [**40**]; the Matrix structure greatly affects performance. For denser matrices the algorithm is slower than the average. While for the tests on GPU using CUDA and OpenCL platforms, CUDA performed 1.2 x better, so it seems to be a better choice for applications where achieving as high a performance as possible is important. Otherwise the choice between CUDA and OpenCL can be made by considering factors such as prior familiarity with either system, or available development tools for the target GPU hardware.

Figure 5 represents a performance comparison in term of execution time of PARALUTION and StarPU libraries on a platform containing one GPU using PCG solver. The result shows that a better performance obtained with StarPU. For the reason that PARALUTION implementation is based on a static distribution of tasks and data, whereas StarPU use scheduling policies that allows a dynamic one.

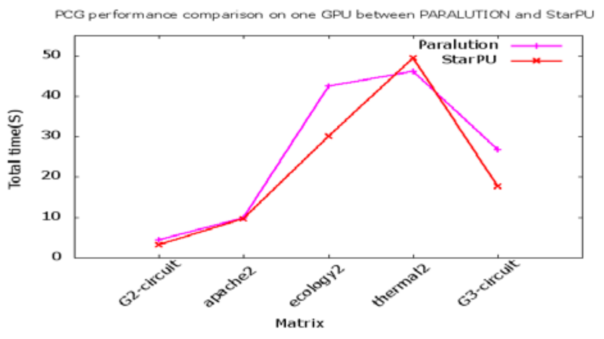

Fig. 5 PCG performance on one GPU platform using StarPU and PARALUTION libraries

In the heterogeneous case, Figure 6 and 7 show the PCG performance with different schedulers of StarPU. In figure 6, the authors fix the number of GPUs on 3 and we make CPUs vary from 0 to 12. In figure 7, they fix the number of CPUs on 4 and vary the number of GPUs from 1 to 3. The results demonstrate that the authors don't have to increase the number of processors to have better performance. In fact they have to choose the best combination of CPUs and GPUs, which is combination of 3 GPUs and 4 CPUs in this case.
This choice is based on three parameters; the size of matrix, scalability and schedulers.

To benefit from multi-GPU acceleration, large-size matrices are required because with smaller matrices the use of multiple GPUs does not accelerate the processing.

From 3 CPU the gain becomes negligible (in this case) this is due to the distribution of tasks according to the scheduling strategy for example dmda scheduler that takes into account the data transfer assigns the task to the worker who will do the processing as soon as possible respecting the minimization of the transfer time. If the number of workers is increased, the transfer time increases and thus the total execution time increases.

Comparing with state-of-the-art method, the parallel Preconditioned Conjugate Gradient (PCG) solver computation in [**41**] the authors confirm that PCG algorithm is more efficient on Multi-GPU, beside the Multi-CPU implementation and that the performance cannot scale arbitrarily [**42**].
Furthermore, they can see that the random scheduler performs poorly because it assigns tasks randomly to the worker without knowing the already assigned workload of workers, which limits the number of ready tasks in the system, and introduces significant idle time on the critical resource (GPUs). The other schedulers which are based on data aware and early finish time strategies like WS, dmda and dmdar perform much better than the random scheduler.

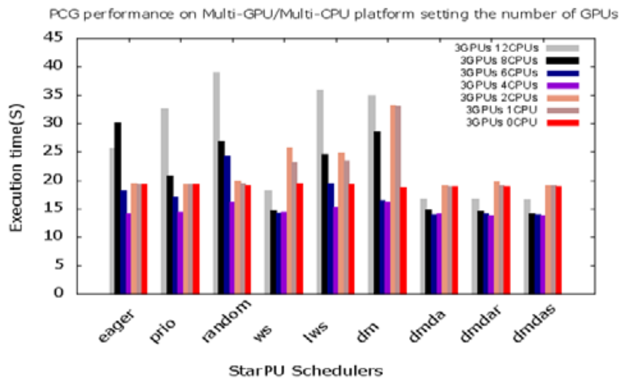

Fig. 6 PCG performance on Multi-GPUs/Multi-CPUs architecture using G3-circuit matrix with StarPU schedulers and by setting the number of GPUs

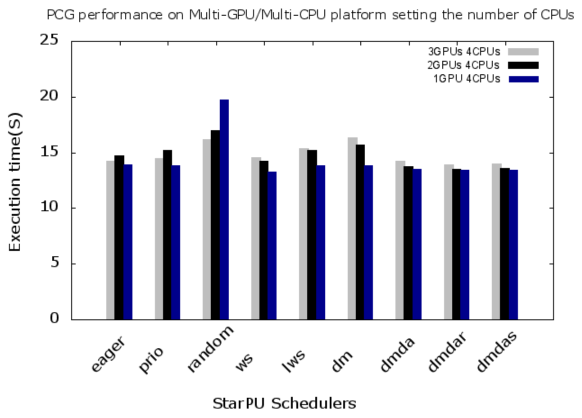

Fig. 7 PCG performance on Multi-GPUs/Multi-CPUs architecture using G3-circuit matrix with StarPU schedulers and by setting the number of CPU

The authors investigated the generated trace files with different schedulers using ViTE profiler[1] in order to determine the reasons of this behaviour. Figure 8 and 9 show traces with WS and dmda schedulers using a combination of 3 GPUs and 4 CPUs.

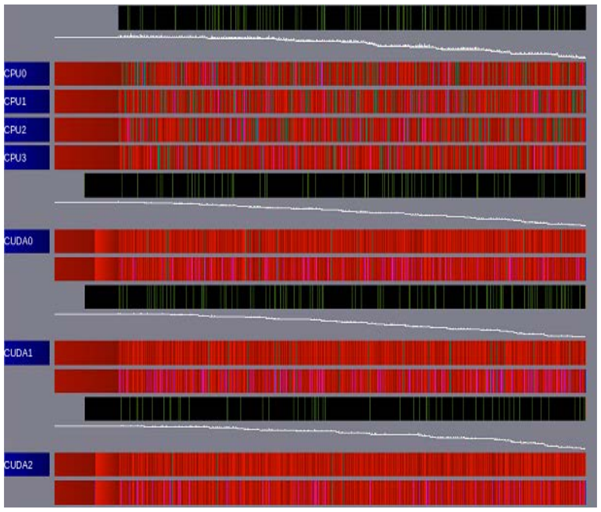

Fig. 8 CPU/GPU trace with WS scheduler

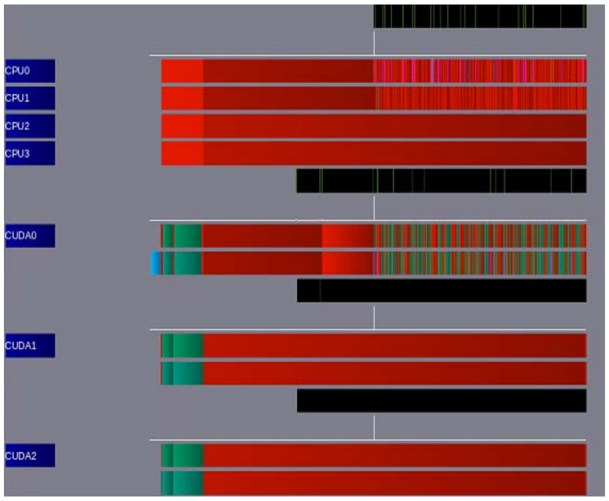

Fig. 9 CPU/GPU trace with dmda scheduler

⬛ Memory 🟥 Idle 🟥 Prep 🟩 Active

Figure 8 shows traces with WS scheduler. The authors can see better load balancing than dmda that is illustrated in figure 9. This is due to strategy of each scheduler. WS schedules by default a task on the processor. When a worker becomes idle, it steals a task from the most loaded worker so it allows better distribution of tasks comparing with dmda scheduler.

In figure 9 authors see traces with dmda scheduler, they note that GPU2 and GPU3 are idle, and GPU1 is the one who process the majority of tasks with the other 2 CPUs. The scheduling strategy of dmda takes into account the data transfer time. It assigns the task to the worker who will do the processing respecting the minimization of the transfer time. If the number of processors is increased, the transfer time increases and thus the total execution time increases. Therefore dmda scheduler does not have the best load balancing but it is the most efficient because it allows minimizing the transfer time and thus the total execution time.

However, the authors found that dmda puts emphasis on critical path rather than parallelism, since it selects some tasks in the beginning that are critical but are not generating enough level of parallelism. This introduces some idle time on the critical resource (GPUs) and degrades the overall performance of the system, which is a known defect of the HEFT scheduler in general. The heterogeneity of performance between different processing units leads to a load imbalance that have been avoided with The WS scheduler as the authors knew that task completion takes so much time on CPUs.

6. Conclusion

In this paper, we presented a performance evaluation of the preconditioned conjugate gradient solver on heterogeneous platform, using StarPU runtime system. The authors also evaluated the performance of PCG on GPU platform with a recent library for solving sparse linear systems PARALUTION.

For the PCG kernel, the authors employed PARALUTION library to compare between two different platforms, CUDA and OpenCL. Furthermore, within the scope of this work, it has been proven that CUDA presents the best choice for applications willing to achieve high and important performances. Moreover, the authors considered the multiple advantages of heterogeneous architecture (Multi-CPUs/Multi-GPUs) to boost the performance of PCG solver using StarPU runtime system. First, they compared the execution time of PCG with PARALUTION and StarPU on one GPU platform then they used different schedulers to achieve a better performance with Multi-CPU/Multi-GPU platforms. The authors concluded that we could run the algorithm on a heterogeneous node composed of all available computational resources but that each GPU has a CPU core dedicated to handle it. As a result, the authors need to choose the best combination of GPUs and CPUs cores.

For future work, authors plan to improve these results by a balanced and synchronized distribution of tasks, efficient

[1] http://vite.gforge.inria.fr/documentation.php

management of CPU and GPU memories and an overlapping of data transfer and computation in order to hide the CPU-GPU communications.

References

[1] D. Zydek and H. Selvaraj, "Hardware implementation of processor allocation schemes for mesh-based chip multiprocessors," Microprocessors and Microsystem, pp. 39-48, 2010.

[2] R. E. Duran, D. Chen, R. Saraswat, and A. Hallmark, "A framework for comparing high performance computing technologies," International Journal of Computational Science and Engineering, pp. 119-129, 2014.

[3] D. D'Ambrosio, G. Filippone, R. Rongo, W. Spataro, and G. A. Trunfio, "Cellular automata and GPGPU: an application to lava flow modeling," International Journal of Grid and High Performance Computing (IJGHPC), pp. 30-47, 2012.

[4] B. Liu, D. Zydek, H. Selvaraj, and L. Gewali, "Accelerating high performance computing applications: Using cpus, gpus, hybrid cpu/gpu, and fpgas," Parallel and Distributed Computing, Applications and Technologies (PDCAT), pp. 337-342, 2012.

[5] R. Dimond, S. Racaniere, and O. Pell, "Accelerating large-scale HPC applications using FPGAs," Computer Arithmetic (ARITH), pp. 191-192, 2011.

[6] P. da Cunha Possa, S. A. Mahmoudi, N. Harb, and C. Valderrama, "A new self-adapting architecture for feature detection," Field Programmable Logic and Applications (FPL), IEEE, pp. 643-646, 2012.

[7] Y. Jararweh and S. Hariri, "Power and performance management of gpus based cluster," International Journal of Cloud Applications and Computing (IJCAC), pp. 16-31, 2012.

[8] John D. Owens and al., "GPU computing," Proceedings of the IEEE 96.5, pp. 879-899, 2008.

[9] F. Vazquez, E. M. Garzon, and J. J. Fernandez, "A matrix approach to tomographic reconstruction and its implementation on GPUs," Journal of Structural Biology, pp. 146-151, 2010.

[10] E. Müller, X. Guo, R. Scheichl, and S. Shi, "Matrix-free GPU implementation of a preconditioned conjugate gradient solver for anisotropic elliptic PDEs," Computing and Visualization in Science, pp. 41-58, 2013.

[11] N. Bell and M. Garland, CUSP library v0. 2: Generic parallel algorithms for sparse matrix and graph computations. Version 0.3. 0, 35., 2012.

[12] N. Trost, J. Jiménez, D. Lukarski, and V. Sanchez, "Accelerating COBAYA3 on multi-core CPU and GPU systems using PARALUTION," Annals of Nuclear Energy, pp. 252-259, 2015.

[13] C. Augonnet, S. Thibault, R. Namyst, and P. A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," Concurrency and Computation: Practice and Experience, pp. 187-198, 2011.

[14] M. R. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems," NBS, p. 1, 1952.

[15] M. R. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems," NBS, p. 1, 1952.

[16] J. W. Demmel, M. T. Heath, and H. A. Van Der Vorst, "Parallel numerical linear algebra," Acta numerica, pp. 111-197, 1993.

[17] A. T. Chronopoulos and C. W. Gear, "s-Step iterative methods for symmetric linear systems," Journal of Computational and Applied Mathematics, pp. 153-168, 1989.

[18] Y. Gui and G Zhang, "An Improved Implementation of Preconditioned Conjugate Gradient Method on GPU," JSW, pp. 2695-2702, 2012.

[19] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on GPUs," ACM sigplan notices, pp. 115-126, 2010.

[20] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder, "Sparse matrix solvers on the GPU: conjugate gradients and multigrid.," ACM Transactions on Graphics, pp. 917-924, 2003.

[21] J. Krüger and R. Westermann, "Linear algebra operators for GPU implementation of numerical algorithms," ACM Transactions on Graphics (TOG), pp. 908-916, 2003.

[22] A. W. O. Rodrigues, F. Guyomarc'h, J. L. Dekeyser, and Y. Le Menach, "Automatic multi-GPU code generation applied to simulation of electrical machines," IEEE Transactions on Magnetics, pp. 831-834, 2012.

[23] A. Corrigan, F. F. Camelli, R. Löhner, and J. Wallin, "Running unstructured grid-based CFD solvers on modern graphics hardware," International Journal for Numerical Methods in Fluids, pp. 221-229, 2011.

[24] E. Agullo, L. Giraud, A. Guermouche, S. Nakov, and J. Roman, "Task-based Conjugate-Gradient for multi-GPUs platforms," INRIA, 2012.

[25] N. Trost, J. Jiménez, D. Lukarski, and V. Sanchez, "Accelerating COBAYA3 on multi-core CPU and GPU systems using PARALUTION," Annals of Nuclear Energy, pp. 252-259, 2015.

[26] Documentation NVIDIA, Cuda C programming guide. Version 5.5: NVIDIA, 2014.

[27] R. Li and Y. Saad, "GPU-accelerated preconditioned iterative linear solvers," The Journal of Supercomputing, pp. 443-466, 2013.

[28] N. Kasmi, S. A. Mahmoudi, M. Zbakh, and P. Manneback, "Performance evaluation of sparse matrix-vector product (SpMV) computation on GPU architecture," Complex Systems (WCCS), IEEE, pp. 23-27, 2014.

[29] M. D. Marino and K. C. Li, "Insights on memory controller scaling in multi-core embedded systems," International Journal of Embedded Systems, pp. 351-361, 2014.

[30] M. Bauer, H. Cook, and B. Khailany, "CudaDMA: optimizing GPU memory bandwidth via warp specialization," Proceedings of 2011 international conference for high performance computing, networking, storage and analysis, p. 12, 2011.

[31] S. Cook, "CUDA programming: a developer's guide to parallel computing with GPUs," Newnes, 2012.

[32] Armstrong and Derek Elswick, "CUDA GPU Programming Applied to HSI Exploitation," Los Alamos National Laboratory (LANL), 2017.

[33] S. F. Ashby and R. D. Falgout, "A parallel multigrid preconditioned conjugate gradient algorithm for groundwater flow simulations," Nuclear Science and Engineering, pp. 145-159, 1996.

[34] S. Zhong and S. Chen, "An improved unsharp masking model for intensive care unit chest radiograph enhancement," Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), pp. 655-661, 2016.

[35] C. Guyeux, R. Couturier, P. C. Héam, and J. M. Bahi, "Efficient and cryptographically secure generation of chaotic pseudorandom numbers on GPU," The journal of Supercomputing, pp. 3877-3903, 2015.

[36] Jesse D. Hall, Nathan A. Carr, and John C. Hart, "Cache and bandwidth aware matrix multiplication on the GPU," 2003.

[37] S. Mittal, "A study of successive over-relaxation method parallelisation over modern HPC languages," International journal of high performance computing and networking, pp. 292-298, 2014.

[38] J. C. Ikuno, M. Wrulich, and M. Rupp, "System level simulation of LTE networks," Vehicular Technology Conference (VTC 2010-Spring), IEEE, pp. 1-5, 2010.

[39] A. Van der Sluis and H. A. van der Vorst, "The rate of convergence of conjugate gradients," Numerische Mathematik, pp. 543-560, 1986.

[40] A. Cevahir, A. Nukada, and S. Matsuoka, "Fast conjugate gradients with multiple GPUs," , Berlin Heidelberg, 2009, pp. 893-903.

[41] R. Helfenstein and J. Koko, "Parallel preconditioned conjugate gradient algorithm on GPU," Journal of Computational and Applied Mathematics, pp. 3584-3590, 2012.

[42] M. Ament, G. Knittel, D. Weiskopf, and W. Strasser, "A parallel preconditioned conjugate gradient solver for the poisson problem on a multi-gpu platform," Parallel, Distributed and Network-Based Processing (PDP), IEEE, pp. 583-592, 2010.

[43] C. Augonnet, S. Thibault, R. Namyst, and P. A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," Concurrency and Computation: Practice and Experience, pp. 187-198, 2011.

[44] StarPU handbook, version 1.02 Orc5., 2016.

[45] H. Topcuoglu, S. Hariri, and M. Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," IEEE transactions on parallel and distributed systems, pp. 260-274, 2002.

[46] Hestenes, Magnus Rudolph, and Eduard Stiefel, "Methods of conjugate gradients for solving linear systems," NBS, p. 49, 1952.

[47] A. Munshi, "The opencl specification," Hot Chips 21 Symposium (HCS), IEEE, pp. 1-314, 2009.

[48] K. K. Matam and K. Kothapalli, "Accelerating sparse matrix vector multiplication in iterative methods using GPU," Parallel Processing (ICPP), IEEE, pp. 612-621, 2011.

[49] G. Chen, G. Li, S. Pei, and B. Wu, "High performance computing via a GPU," Information Science and Engineering (ICISE), pp. 238-241, 2009.