Task-based Conjugate-Gradient for multi-GPUs platforms

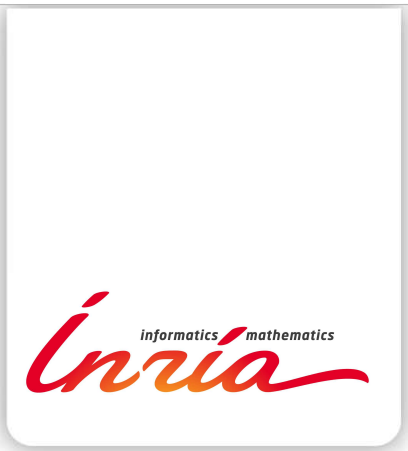
Emmanuel Agullo, Luc Giraud, Abdou Guermouche, Stojce Nakov, Jean Roman

Task-based Conjugate-Gradient for multi-GPUs platforms

Emmanuel Agullo, Luc Giraud, Abdou Guermouche, Stojce Nakov, Jean Roman

RESEARCH REPORT
N° 8192
Décembre 2012
Project-Teams HiePACS

ISSN 0249-6399 ISRN INRIA/RR--8192--FR+ENG



Task-based Conjugate-Gradient for multi-GPUs platforms

Emmanuel Agullo, Luc Giraud, Abdou Guermouche, Stojce Nakov, Jean Roman

Project-Teams HiePACS

Research Report n° 8192 — Décembre 2012 — 25 pages

Abstract: Whereas most today parallel High Performance Computing (HPC) software is written as highly tuned code taking care of low-level details, the advent of the manycore area forces the community to consider modular programming paradigms and delegate part of the work to a third party software. That latter approach has been shown to be very productive and efficient with regular algorithms, such as dense linear algebra solvers. In this paper we show that such a model can be efficiently applied to a much more irregular and less compute intensive algorithm. We illustrate our discussion with the standard unpreconditioned Conjugate Gradient (CG) that we carefully express as a task-based algorithm. We use the StarPU runtime system to assess the efficiency of the approach on a computational platform consisting of three NVIDIA Fermi GPUs. We show that almost optimum speed up (up to 2.89) may be reached (relatively to a mono-GPU execution) when processing large matrices and that the performance is portable when changing the low-level memory transfer mechanism.

Key-words: High Performance Computing (HPC); GPU; Task; Runtime System; Conjugate Gradient.

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

351, Cours de la Libération
Bâtiment A 29
33405 Talence Cedex

L'algorithme du Gradient Conjugué en tâches pour les plate-formes multi-GPUs

Résumé : Tandis que la plupart des logiciels de calcul haute performance (HPC) actuels sont des codes extrêmement optimisés en prenant en compte les détails de bas-niveau, l'avènement de l'ère manycore incite la communauté à considèrer des paradigmes de programmation modulaires et ainsi déléguer une partie du travail à des librairies tierces. Cette dernière approche s'est avérée très productive et efficace dans le cas d'algorithmes réguliers, tels que ceux issus de l'algèbre linéaire dense. Dans ce papier, nous démontrons qu'un tel modèle peut être efficacement appliqué à un problème beaucoup plus irrégulier et moins intensif en calcul. Nous illustrons notre discussion avec l'algorithme standard du Gradient Conjugué (CG) non préconditionné que nous exprimons sous la forme d'un algorithme en graphe de tâches. Nous utilisons le moteur d'exécution StarPU pour évaluer l'efficacité de notre approche sur une plate-forme de calcul composée de trois accélérateurs graphiques (GPU) NVIDIA Fermi. Nous démontrons qu'une accroissement de performance (jusqu'à un facteur $2,89$) quasi optimal (relativement au cas mono-GPU) peut être atteinte lorsque sont traitées des matrices creuses de grande taille. Nous montrons de surcroît que la performance est portable quand les mécanismes de transfert mémoire bas-niveau sont changés.

Mots-clés : Calcul Haute Performance (HPC); GPU; Tâche; Moteur d'exécution; Gradient Conjugué.

1 Introduction

In the last decade, the architectural complexity of High Performance Computing (HPC) platforms has strongly increased. To cope with this complexity, programming paradigms are being revisited. Among other, one major trend consists of writing the algorithms in terms of task graphs and delegating to a runtime system both the management of the data consistency and the orchestration of the actual execution. This paradigm has been intensively studied in the context of dense linear algebra [4, 5, 6, 17, 19, 33, 40, 42] and is now a common utility for related state-of-the-art libraries such as PLASMA [2], MAGMA [1] and FLAME [46]. Dense linear algebra algorithms were indeed excellent candidates for pioneering in this direction. First, their computational pattern allows one to design very wide task graphs so that many computational units can execute tasks concurrently. Second, the building block operations they rely on, essentially level-three Basic Linear Algebra Subroutines (BLAS), are compute intensive, which makes it possible to split the work in relatively fine grain tasks while fully benefiting from GPU acceleration. As a result, these algorithms are particularly easy to schedule in the sense that state-of-the-art greedy scheduling algorithms may lead to a performance close to the optimum, including on platforms accelerated with multiple Graphics Processing Units (GPUs) [5].

In this paper, we tackle another class of algorithms, the Krylov subspace methods, which aim at solving large sparse linear systems of equations of the form $Ax = b$, where A is a sparse matrix. Those methods are based on the calculation of approximated solutions in a sequence of embedded spaces, that is intrinsically a sequential numerical scheme. Second, their unpreconditioned versions are exclusively based on non compute intensive kernels, Sparse Matrix Vector products (SpMV) and level-one BLAS, which need very large grain tasks to benefit from GPU acceleration. For these reasons, designing and scheduling Krylov subspace methods on a multi-GPUs platform is extremely challenging, especially when relying on a task-based abstraction which requires to delegate part of the control to a runtime system. We discuss this methodological approach in the context of the unpreconditioned Conjugate Gradient (CG) algorithm on a shared-memory machine accelerated with three NVIDIA Fermi GPUs, using the StarPU runtime system [9] to process the designed task graph. The CG solver is a widely used Krylov subspace method, which is the numerical algorithm of choice for the solution of large linear systems with symmetric positive definite matrices [44].

The objective of this study is *not* to optimize the performance of CG on an individual GPU, which essentially consists of optimizing the matrix layout in order to speed up SpMV. We do *not* either consider the opportunity of reordering the matrix in order to improve the SpMV. Finally, we do *not* consider numerical variants of CG which may exhibit different parallel patterns. These three techniques are extremely important but complementary and orthogonal to our work, and we discuss them in the related state-of-the-art methods (Section 2). Instead, we rely on routines from vendor libraries (NVIDIA CUSPARSE and CUBLAS) to implement individual GPU tasks, we assume that the ordering is prescribed (we do not apply permutation) and we consider the standard formulation of the CG algorithm [44].

The objective is to study the opportunity to accelerate CG when multiple GPUs are available by designing an appropriate task flow where each individual task is processed on one GPU and all available GPUs are exploited to execute these tasks concurrently. We first propose a natural task-based expression of CG. We show that such an expression fails to efficiently accelerate CG. We then propose successive improvements on the task flow design to alleviate the synchronizations, exhibit more parallelism (wider graph) and reduce the volume of exchanges between GPUs. We show that (for large enough matrices) CG can be carefully scheduled with a nearly optimum speed-up (up to 2.89 on our three GPUs machine with respect to the single GPU case reference). Finally, we illustrate the benefits of relying on a runtime system abstracting the underlying

hardware. The same task flow (i.e., high-level task-based algorithm) is indeed executed on top of two different GPU to GPU memory transfer mechanisms. Depending on the CUDA version, the programmer has to force data to transit through main memory (GPU-CPU-GPU case) or not (GPU-GPU case). In our task-based abstraction model, the runtime system transparently and efficiently handles that for us.

The rest of the paper is organized as follows. In Section 2, state-of-the-art SpMV and Krylov methods on GPUs are presented, as well as modern runtime systems. We propose a task-based formulation of the CG algorithm in Section 4. The experimental environment is introduced in Section 5. Prior to study the behavior of CG itself, we present a performance analysis of its building block computational kernels in Section 6. We refine the initial formulation of the CG task flow in Section 7 in order to alleviate the synchronizations, exhibit more parallelism and reduce data transfers. Section 7.5 illustrates the benefits of using a runtime system to abstract the hardware in terms of productivity and performance portability. The performance of our refined CG algorithms are finally presented in Section 8 before concluding in Section 9.

2 State-of_the_art

SpMV is the core kernel of sparse iterative methods. In [50], (resp. [49]) an overview is given on the performance of the CSR-based SpMV (resp. the CG algorithm) for a number of modern CPU architectures. Then, with the democratization of GPUs, several studies have focused on the improvement of the performance of the SpMV kernel on GPUs. These efforts have mainly targeted the sparse matrix representation format to improve the memory access pattern/footprint of the kernel. From the format point of view of the matrix representation, several layouts were studied such as the compressed sparse row (CSR) format, the coordinate format (COO) format, the diagonal (DIA) format, the ELLPACK (ELL) format and a hybrid (ELL/COO) format. An exhaustive description of different implementations of the SpMV operation for GPU architectures can be found in [15]. More recently, a new row-grouped CSR format was considered [39] as well as a sliced ELLPACK format [37]. In [14], Bell and Garland propose several methods for efficient sparse matrix-vector multiplications that take into account the structure of the input matrices. They implemented efficient multiplication routines for various sparse matrix layouts including a new hybrid layout (this layout stores part of the matrix using ELLPACK and the remaining elements using the COO format in order to reduce the memory footprint). Their hybrid layout is most suitable for unstructured matrices and delivers in general the best performance for such matrices. This work was followed by many efforts targeting these hybrid representations [22, 36]. A model-driven autotuning approach was introduced in [22] in order to find the best parameters for the hybrid storage format.

From the sparse iterative methods point of view, many efforts have been conducted to adapt the existing algorithms and exploit GPU. The block-ILU preconditioned GMRES method is studied for solving unsymmetric sparse linear systems on the NVIDIA Tesla GPUs in [48]. SpMV kernels on GPUs and the block-ILU preconditioned GMRES method are used in [34] for solving large sparse linear systems in black-oil and compositional flow simulations. More recently, several works have addressed the multi-GPU case where the computational node is enhanced with more than one GPU. In [8], a specific CG method with incomplete Poisson preconditioning is proposed for the Poisson problem on a multi-GPU platform. In [25], the authors present an implementation of the CG algorithm for multi-GPU platforms based on a fast distributed SpMV kernel using optimized data formats (combining ELL and padded CSR) to allow for a good overlapping between communication and computation. In [20], the authors present a CG algorithm running on multiple GPUs using a data parallel approach where the number of non-zeros is balanced

among GPUs. The authors have then refined their approach in [21], by using a new partitioning for the matrix based on a hypergraph model to reduce the amount of communication needed by the algorithm. In [47], the authors describe mappings for the SpMV kernels and show how they parallelize the CG algorithm over the GPUs by implementing parallel operations (i.e., kernels running on multiple devices). Moreover they present results illustrating that reordering the input matrix can improve the performance of the method. Recently, GMRES has been studied in the context of multiple GPUs platforms [13, 24] where authors mainly identify the operations that have to be performed by GPUs (e.g. SpMV) and the operation that need to be processed by CPUs (reductions for example).

All the studies mentioned above were implemented on top of CUDA [23] or OpenCL [38]. Alternatively, expressing the algorithms at a higher level which abstracts the underlying platform may be more productive and ensure performance portability, as successfully demonstrated in the context of the dense linear algebra for the MAGMA [3] and FLAME [41] projects. Such an approach then needs an intermediate software layer, a runtime system, that executes the high-level algorithm on the actual platform. Many initiatives have emerged in the past years to develop efficient runtime systems for platforms equipped with multiple GPUs. Different programming paradigms have been proposed. Qilin [35] provides an interface to submit kernels that operate on arrays which are automatically dispatched between the different processing units of a heterogeneous machine. Moreover, Qilin dynamically compiles parallel code for both CPUs (by relying on the Intel TBB [43] technology) and for GPUs, using CUDA. The event-driven Charm++ [31] runtime system is a parallel C++ library that provides sophisticated load balancing and a large number of communication optimization mechanisms. Charm++ has been extended to provide support for accelerators such as the Cell processors and GPUs [32].

Many runtime systems propose a task-based programming paradigm. The StarSs project is an umbrella term that describes both the StarSs language extensions and a collection of runtime systems targeting different types of platforms [11, 12, 16]. StarSs provides an annotation-based language which extends C or Fortran applications to offload pieces of computation on the architecture targeted by the underlying runtime system. The runtime system dynamically schedules tasks within a node using a work-stealing policy. Other task-based runtime systems, not based on directives, like APC+ [29], DAGuE [18], KAAPI [30], and StarPU [10] offer support for hybrid platforms mixing CPUs and GPUs. In particular, the StarPU runtime system handles low-level data transfers between GPUs very efficiently and proposes a very convenient expression of the task graph. StarPU thus represents an excellent candidate for illustrating our study. We provide more details in the next section.

3 Task-based StarPU runtime system

StarPU 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 fully provided. Instead, the sequence of task 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 our study, each task is performed on a single computational unit. 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 computational unit. StarPU thus ensures both data consistency and transfers between processing units (in our case

GPUs). Furthermore, data prefetching is used to cope with memory latencies.

StarPU offers several advanced utilities for data management. In this study, we specifically consider the data partitioning and data unpartitioning operations. Data partitioning divides a piece of data in several parts. Data unpartitioning is the opposite operation; it assembles several parts of a partitioned data in the original form. Those operations are the counterpart of the scatter/gather operations in the SPMD programming model used by MPI and similar instructions existed also on vector computers a few years ago. Both these operations are blocking operations. That is, they are control instructions for the task flow construction, which prevent the user from submitting any new task as long as the corresponding tasks, partitioning or unpartitioning, have been processed.

4 Task-based Conjugate Gradient algorithm

We now propose a task-based expression of the CG algorithm whose pseudo-code is given in Algorithm 1. This algorithm can be divided in two phases, the initialization phase (lines 1-5) and the main loop (lines 6-16). The initialization phase being executed only once, we only focus on an iteration occurring in the main loop in this paper.

Three types of operations are used in an iteration of the algorithm: SpMV (line 7), scalar operations (lines 9, 13, 14) and level-one BLAS operations (lines 8, 10, 11, 12, 15). In particular three different level-one BLAS operations are used: scalar product (*dot*, lines 8 and 12), linear combination of vectors (*axpy*, lines 10, 11 and 15) and scaling of a vector by a scalar (*scal*, line 15). The *scal* kernel at line 15 is used in combination with an *axpy*. Indeed, in terms of BLAS, the operation $p \leftarrow r + \beta p$ consists of two successive operations: $p \leftarrow \beta p$ (*scal*) and then $p \leftarrow r + p$ (*axpy*). In our implementation, the combination of these level-one BLAS operations represents a single task called *scal-axpy*. The most costly operation in an iteration is the *SpMV* (line 7) and its performance is thus critical for the behavior of the whole algorithm.

Algorithm 1 Pseudo-Code of Conjugate Gradient algorithm

1: $r \leftarrow b$
2: $r \leftarrow r - Ax$
3: $p \leftarrow r$
4: $\delta_{new} \leftarrow dot(r, r)$
5: $\delta_{old} \leftarrow \delta_{new}$
6: **for** $j = 0, 1, ...,$ until $\frac{\|b-Ax\|}{\|b\|} \leq eps$ **do**
7: $q \leftarrow Ap$ /* SpMV */
8: $\alpha \leftarrow dot(p, q)$ /* BLAS-1 */ (dot)
9: $\alpha \leftarrow \delta_{new}/\alpha$ /* scalar operation */
10: $x \leftarrow x + \alpha p$ /* BLAS-1 */ (axpy)
11: $r \leftarrow r - \alpha q$ /* BLAS-1 */ (axpy)
12: $\delta_{new} \leftarrow dot(r, r)$ /* BLAS-1 */ (dot)
13: $\beta \leftarrow \delta_{new}/\delta_{old}$ /* scalar operation */
14: $\delta_{old} \leftarrow \delta_{new}$ /* scalar operation */
15: $p \leftarrow r + \beta p$ /* BLAS-1 */ (scal-axpy)
16: **end for**

According to our programming paradigm (Section 3), data need to be decomposed in order to provide opportunities for executing concurrent tasks. We consider a 1D decomposition of the matrix, dividing the matrix in several block-rows. The number of non-zero values per block-rows

is balanced and the rest of the vectors follows the same decomposition as illustrated in Figure 1.

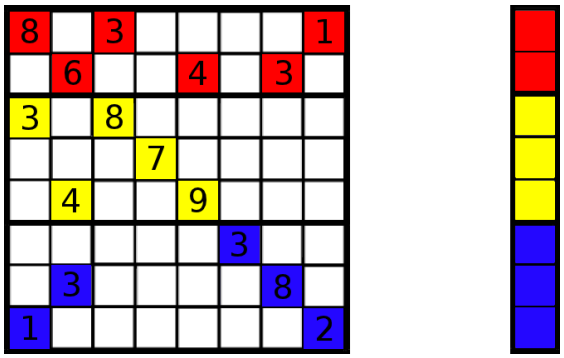


Figure 1: Matrix and vector decomposition. 1D decomposition is applied to the matrix balancing the number of non-zero values per block-rows (left). The rest of the vectors follows the same decomposition (right).

After decomposing the data, tasks that operate on those data can be defined. The tasks derived from the main loop of Algorithm 1 are shown in Figure 2, when the matrix is divided in six block-rows. Each task is represented by a box, named after the operation executed in that task, and edges represent the dependencies between tasks.

Let us examine in more details the task flow in Figure 2. The first instruction executed in the main loop of Algorithm 1 is the *SpMV*. *SpMV* performs the operation $q \leftarrow Ap$ where A is a sparse matrix and q and p are dense vectors. When a 1D decomposition is applied to the matrix, dividing it in six parts implies that six tasks are submitted for this operation (the green tasks in Figure 2): $q_i \leftarrow A_i p, i \in [1,6]$, one for each block-row A_i of the matrix. For these tasks, a copy of the whole vector p is needed (vector p is unpartitioned). But in order to extract parallelism of other level-one BLAS operations where vector p is used (lines 8 and 15 in Algorithm 1), it needs to be partitioned. As discussed in Section 3, the partitioning operation is a blocking call; it thus represents a synchronization point in this task flow (represented with the red vertical bar after *SpMV* tasks in Figure 2). Once vector p is partitioned, the tasks corresponding to the scalar product (line 8 in Algorithm 1) can then be submitted. Both vectors p and q are partitioned in six parts ($p_i, q_i, i \in [1,6]$), so six tasks are again submitted. Each *dot* operation accesses α in read-write mode, which induces a serialization of the operation, as shown in Figure 2 with the dependencies between the successive *dot* tasks. This sequence thus introduces new synchronizations in the task flow. The final value of α for the current iteration is obtained after the scalar operation $\alpha \leftarrow \delta_{new}/\alpha$ (line 9). The twelve *axpy* tasks (six at line 10 and six at line 11) can then all be executed in parallel (see Figure 2 again). Another *dot* operation is then performed (line 12) and induces other serializations (as it was the case for the *dot* at line 8). After the scalar operations at lines 13 and 14 in Algorithm 1, the last operation of the loop can be executed . Similarly to the *axpy* operations, this operation is completely parallel. After this last operation, the new value of vector p is obtained. At this stage, it is partitioned in multiple pieces ($p_i, i \in [1,6]$). In order to perform the *SpMV* tasks (line 7, next iteration) which all need the whole vector p, these pieces are unpartitioned to form the input variable p.

All in all, this task flow contains four synchronization points per iteration and is very thin. Section 7.1 exhibits the induced limitation in terms of pipelining, while sections 7.2, 7.3 and 7.4 propose successive improvements allowing us to alleviate the synchronizations and design a wider task flow, thus increasing concurrency.

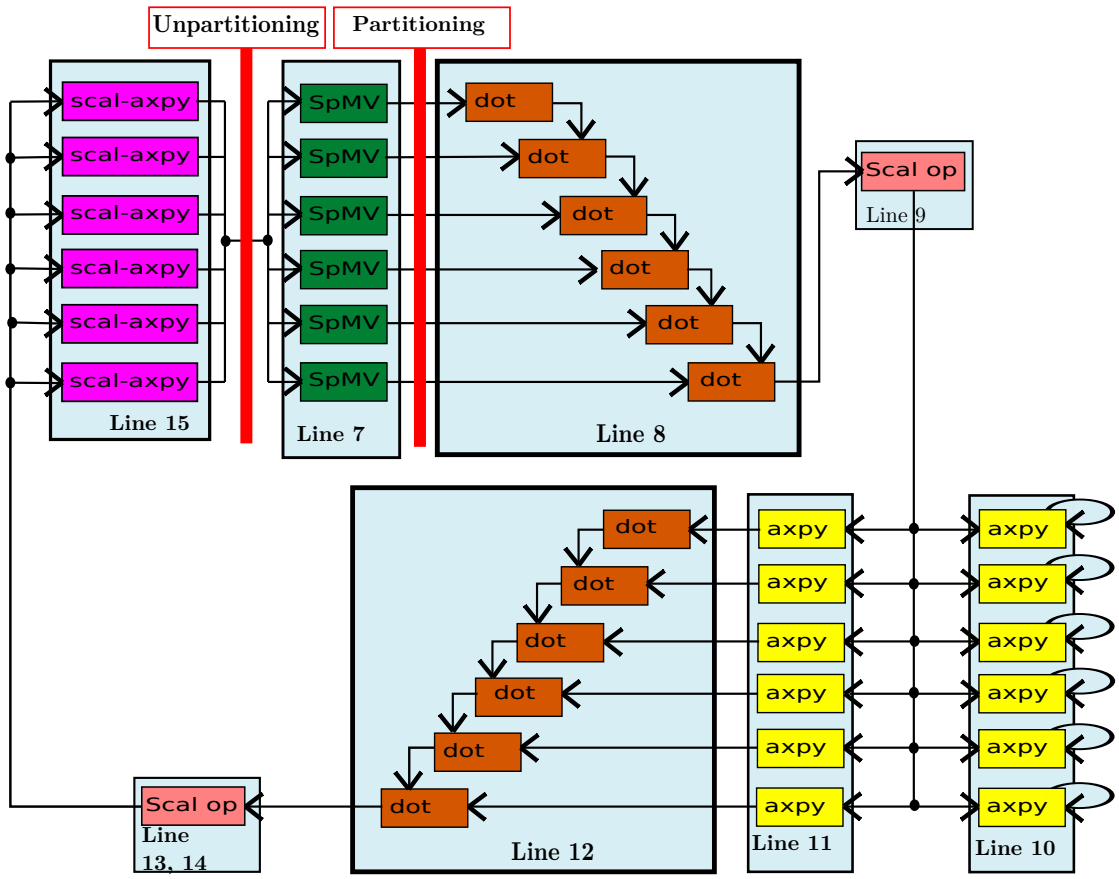


Figure 2: Task flow of the main loop of CG Algorithm (lines 7-15 in Algorithm 1). The matrix is divided in six block-rows. Vertices of the graph represent tasks and edges represent dependencies between them. The red vertical bars between the $scal-axpy$ and $SpMV$ tasks and between the $SpMV$ tasks and the *dot* tasks represent the partitioning and the unpartitioning of the vector p, respectively.

5 Experimental Setup

5.1 Hardware

All the tests presented in this paper were run on a cache coherent Non Uniform Memory Access (ccNUMA) machine with two hexa-core processors Intel Westmere Xeon X5650, each one having 18GB of RAM, for a total of 36GB. It is equipped with three NVIDIA Tesla M2070 GPUs, each one having 6GB of RAM memory. In all the experiments discussed in this paper, CPU are not used to perform computation; instead, we assess the potential benefits to accelerate our algorithms when using multiple GPUs (with respect to a mono-GPU execution).

Since multiple GPUs are used, data have to be moved between GPUs during the execution of the algorithm. Before CUDA v4.0, communication between GPUs required a copy of the data to the RAM before sending it to the receiving GPU (we will call GPU-CPU-GPU this communication mechanism in the rest of the paper). Since CUDA v4.0, data transfers between GPUs may be performed directly from one GPU to another, bypassing the intermediate copy to RAM (GPU-GPU communication mechanism). In Section 7 the GPU-CPU-GPU mechanism is used, except in Section 7.5 where the GPU-GPU mechanism is investigated.

5.2 Software

The task-based CG algorithm proposed in Section 4 is implemented on top of the StarPU runtime system (see Section 3). We use the opportunity offered by StarPU to control each GPU with a dedicated CPU core. We rely on the CUBLAS v2.0 and CUSPARSE v1.0 vendor libraries to implement the GPU tasks that execute the level-one BLAS and SpMV operations.

5.3 Scheduling and mapping strategy

As discussed in Section 4, the task flow derived from Algorithm 1 contains four synchronization points per iteration and is very thin, ensuring only a very limited concurrency. Furthermore, as we will show in Section 6, the tasks are not very compute intensive so that moving data between GPUs is expensive relatively to the actual execution of the task. Pipelining the task flow efficiently is thus very challenging. In particular, dynamic strategies that led to close to optimum schedules in dense linear algebra [5] are not well suited here. We have indeed experimented such a strategy (Minimum Completion Time (MCT) policy [45]) but all studied variants failed to achieve a very high performance. Indeed, with such a thin graph, each inaccurate decision induced a strong imbalance that could not be recovered. We have thus implemented a static scheduling strategy. Each block-row of the matrix is associated with a GPU and the related tasks are performed on that GPU. In order to increase concurrency, we have considered the case where there are more block-rows than GPUs. In that case, we perform a cyclic mapping of the block-rows on the GPUs in order to ensure load balancing.

5.4 Matrices

To illustrate our discussion we consider the matrices presented in Table 1. As discussed above, matrices are divided in block-rows and each block-row is mapped on a GPU. In all the experiments presented in this study, each block-row is thus initially prefetched on the GPU that will perform the corresponding SpMV task, *before* the experiment is timed, in order to represent the behaviour occurring in a CG iteration (except for the first iteration, which is not considered here). Furthermore, when assessing the behaviour of the building block operations (Section 6), the corresponding block-vectors are also prefetched on those GPUs.

Matrix name	nnz	N	nnz / N	flop/iteration
11pts-256	183,6 M	16,7 M	10.9	1,9 G
11pts-128	22,8 M	2,1 M	10.9	224,4 M
audikw_1	154,3 M	943,6 K	163.6	317,2 M

Table 1: Overview of sparse matrices used in this study. The 11pts-256 and 11pts-128 matrices are obtained from a 3D regular grid with 11pt computation stencil. The $audikw\_1$ and af_0_k101 matrices come from irregular finite element mesh for structural mechanics.

6 Building Block Operations

In order to explore the potential parallelism of the CG algorithm, we first study the performance of its building block operations, level-one BLAS and $SpMV$, on our platform.

6.1 Level-one BLAS operations

As mentioned in Section 4, three level-one BLAS operations are used in CG: *dot*, *axpy* and $scal - axpy$. These kernels have a computational cost of $2N$ for the *dot* and *axpy* operations and of $4N$ for the $scal - axpy$ operation, where N is the size of the vectors. This low computational cost relatively to the amount of data involved makes them hard to accelerate on GPUs. Figure 3

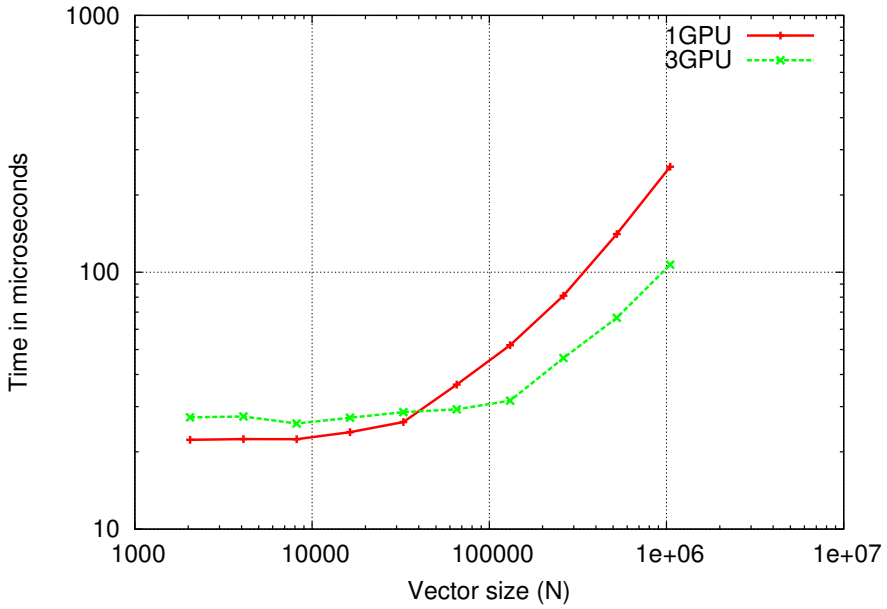


Figure 3: Performance of level-one BLAS operations on multiple GPUs. Both axes are expressed in logarithmic scale. Data is divided in equal pieces and is prefetched on every GPU before execution and performance assessment. Here is shown the performance of the *axpy* kernel, but all kernels follow the same behavior.

shows this effect. We recall that the matrix is split in equal block-rows and that the vectors

are divided accordingly. Following the experimental set up presented in Section 5.4, the input vectors are prefetched on the GPUs before execution. Therefore, in this experiment, only the computational time is measured. Nonetheless, very large vectors need to be considered ($N > 10^6$) to benefit from multi-GPU acceleration (relatively to one GPU). When the input vectors are of intermediate size ($10^5 < N < 10^6$), the use of multiple GPUs does not speed up the overall performance anymore. When the vectors are smaller ($N < 10^5$), a multi-GPU context even slows down the execution. In that latter case, the computation time is negligible and there is still no communication; the only measured time is the start-up spent for launching the kernels. Because of a lock occurring within the NVIDIA driver[1] when launching a CUDA kernel, at such a fine granularity, concurrent tasks are thus actually serialized and the launching times are paradoxically cumulated.

6.2 The SpMV operation

When the matrix is split in multiple block-rows, the SpMV operation may be executed as concurrent SpMV tasks. The performance obtained for the *audikw_1* matrix is shown in Figure 4. The

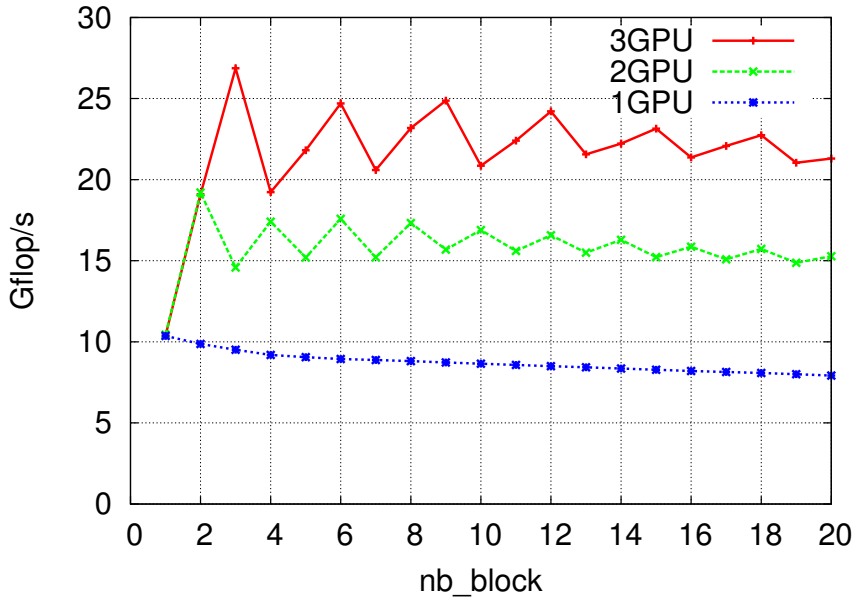


Figure 4: Performance of *SpMV* with *audikw_1* matrix when split in nb_block block-rows (x-axis). These blocks are prefetched on GPUs before the experiment (see Section 5.4).

impact of the task granularity on performance may be assessed with the performance observed on one GPU. When the matrix is split in multiple block-rows, the performance is not strongly penalized. For instance, the mono-GPU execution only decreases of 8% when the matrix is split in three blocks (from 10.37 Gflop/s to 9.50 Gflop/s). When multiple GPUs are used, if the number of blocks is a multiple of the number of GPUs, a correct load balancing may be furthermore achieved. In particular, when the matrix is divided in three block-rows, the penalty on granularity is minimal while achieving a decent load balancing. A performance of 26.87 Gflop/s is indeed achieved which represents a speed-up of 2.83 over the mono-GPU execution with the

[1] we have reported this surprising behavior to NVIDIA

same number of blocks (9.50 Gflop/s) and an overall 2.59 speed-up over the best mono-GPU execution (10.37 Gflop/s).

7 Achieving Highly Efficient Pipelining

In accordance with the example discussed in Section 4, the matrix is split in six block-rows and three GPUs are used. We pursue our illustration with matrix 11pts-128.

7.1 Assessment of the proposed task-based CG algorithm

Figure 5 shows the execution of one iteration of the task flow (Figure 2) derived from Algorithm 1 with respect to the mapping proposed in Section 5.3. Figure 5 can be interpreted as follows. The top black bar represents the state of the CPU Random Access Memory (RAM) during

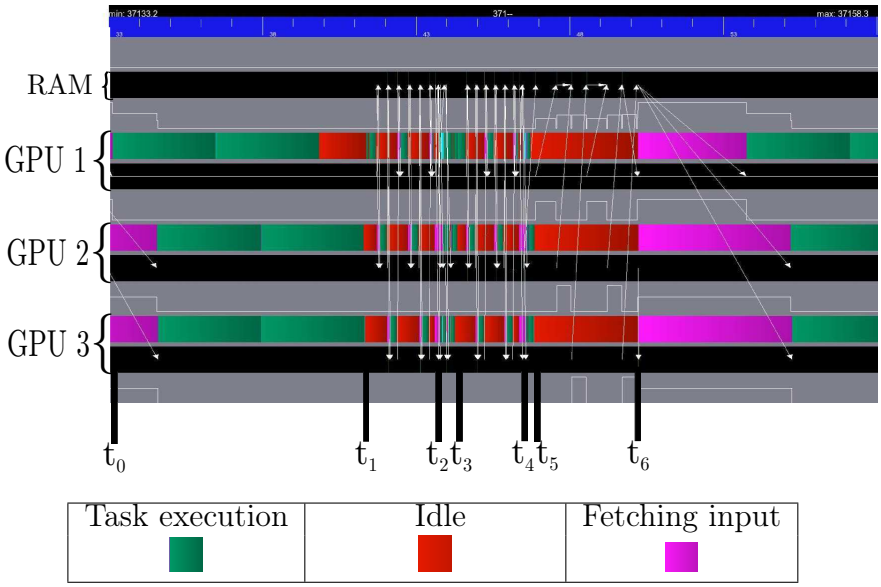


Figure 5: Execution trace of an iteration with the CG task flow of Figure 2 using three GPUs.

the execution. The periods when the RAM is used for a data transfer from (or to) a GPU are highlighted by an arrow from the source to the destination. As explained in Section 5.1, data transfers between GPUs are handled by the runtime system with the GPU-CPU-GPU communication mechanism. Below the bar of the CPU RAM, the three other couples of bars represent the state of the three GPUs. Indeed, for each GPU, the top bar represents the activity of the GPU itself, whereas the bottom bar represents the state of its GPU memory. The activity of a GPU (top bar) may have one of the three following states: active computation (green), idle (red) or active waiting for the completion of a data transfer (purple).

An iteration starts with the execution of a *SpMV* operation (line 7 in Algorithm 1), time interval $[t_0,t_1]$ in Figure 5. As shown in Figure 2, the *SpMV* operation is decomposed in six tasks (green tasks in Figure 2). Following the cyclic mapping strategy presented in Section 5.3, each GPU is thus in charge of two *SpMV* tasks. At time t_1, vector q is available, distributed in six pieces but vector p is unpartitioned. As explained in Section 4, p is partitioned into

six p_i pieces, $i \in [1,6]$, with respect to the block-row decomposition of the matrix. However, this data partitioning operation is a blocking call (see Section 5.2) which means that no other task can be submitted until it is completed at time t_1 (the red vertical bar after the $SpMV$ tasks in Figure 2). Once vector p is partitioned, tasks for all remaining operations (lines 8-15) are submitted. The scalar product tasks are executed sequentially with respect to the cyclic mapping strategy explained in Section 5.3. The reason for this, as explained in Section 4, is that the scalar α is accessed in read-write mode. In addition, α needs to be moved to GPU between each execution of a *dot* task (interval $[t_1, t_2]$ in Figure 5). Once the scalar product at line 8 is computed, the scalar division follows (line 9) executed on GPU 1 (respecting the task flow in Figure 2). The execution of the next two instructions follows (lines 10 and 11). But before the beginning the execution of the *axpy* tasks on GPU 2 and GPU 3, the new value of α is sent according to the GPU-CPU-GPU transfer model (the purple period at t_2 in Figure 5). The *axpy* tasks (yellow tasks in Figure 2) are executed during the period $[t_2, t_3]$ in parallel. The scalar product at line 12 is then executed during the time interval $[t_3, t_4]$, following the same sequence as explained above for line 8. Next, β and δ_{old} are computed on GPU 1 at time t_4 in Figure 5, representing the scalar operations from lines 13 and 14 of Algorithm 1. Tasks related to the last operation of the iteration ($scal - axpy$ tasks in Figure 2) are then processed during the time interval [t_4,t_5]. When all the new vector blocks p_i are calculated, the vector p is unpartitioned (red vertical bar after the $scal - axpy$ tasks in Figure 2). As explained in Section 5.2, this data unpartition is another synchronization point and may only be executed in the RAM. All blocks p_i of vector p are thus moved from the GPUs to the RAM during the time interval [t_5,t_6] for building the unpartitioned vector p. This vector is then used to perform the $q_i \leftarrow A_i \times p$ tasks related to the first instruction of the next iteration ($SpMV$ at line 7). We now understand why the iteration starts with an active waiting of the GPUs (purple parts before time t_0): vector p is only valid on the RAM and thus needs to be copied on the GPUs.

During the execution of the task flow derived from Algorithm 1 (Figure 2), the GPUs are idle during a large portion of the time (red and purple parts in Figure 5). In order to achieve more efficient pipelining of the algorithm, we present successive improvements on the design of the task flow: relieving synchronization points (Section 7.2), reducing volume of communication that is transferred by packing data (Section 7.3) and relying on a 2D decomposition (Section 7.4).

7.2 Relieving synchronization points

Alternatively to the sequential execution of the scalar product, each GPU j can compute locally a partial sum (α^j) and perform a reduction to compute the final value of the scalar ($\alpha = \sum_{j=1,n\_gpus} \alpha^j$). Figure 6 illustrates the benefit of this strategy. The execution of the scalar product, during the time interval $[t_0, t_1]$ is now done in parallel. Every GPU is working on its own local copy of α and once they have finished, the reduction is performed on GPU 1 just after t_1.

The partition (after instruction 7 of Algorithm 1) and unpartition (after instruction 15) of vector p represents two of the four synchronization points of an iteration. They furthermore induce extra management and communication costs. Indeed, after instruction 15, each GPU owns a valid part of vector p. For instance, once GPU 1 has computed p_1, it sends p_1 to the RAM and receives it back, which is useless. Second, vector p has to be fully assembled in the main memory (during the unpartition operation) before prefetching a copy of the fully assembled vector p back to the GPUs (after time t_3 in Figure 6). We have designed another scheme where vector p is kept in a partitioned form all along the execution (it is thus no longer needed to perform partitioning and unpartitioning operations at each iteration). Instead of building and broadcasting the whole unpartitioned vector p, each GPU gathers only the missing pieces. Assuming if p is decomposed

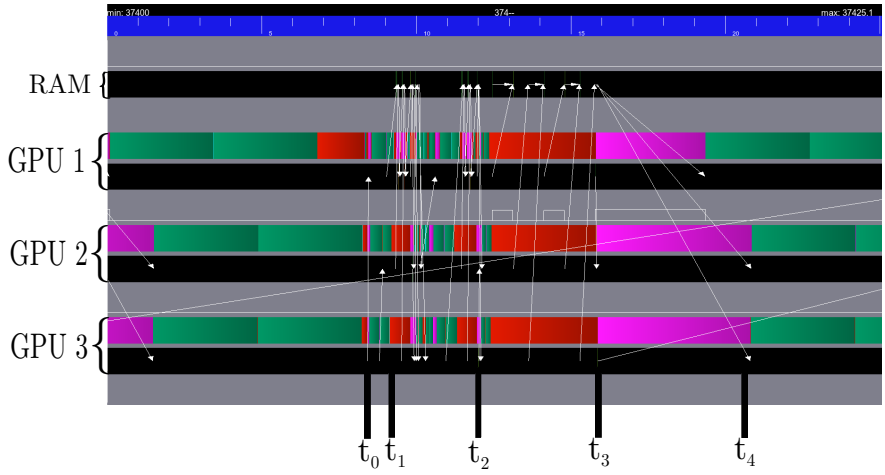


Figure 6: Execution trace of one iteration when the *dot* is performed in reduction mode.

in six pieces, GPU 1 had computed p_1 and p_4 at the previous iteration (following the cyclic mapping strategy presented in Section 5.3). So, by using this vector scheme, the GPU 1 will only receive p_2, p_3, p_5 and p_6 vector blocks. This enables us to decrease the overall traffic. Furthermore, each vector block p_i can be copied from the RAM to GPUs as soon it is fetched in main memory without waiting the other vector blocks to be copied, as it was required in the previous case. Finally, we ensure that all the pieces of p get copied in a contiguous fashion on the device, so that vector p is valid when the task $q_i \leftarrow A_i p$ is executed.

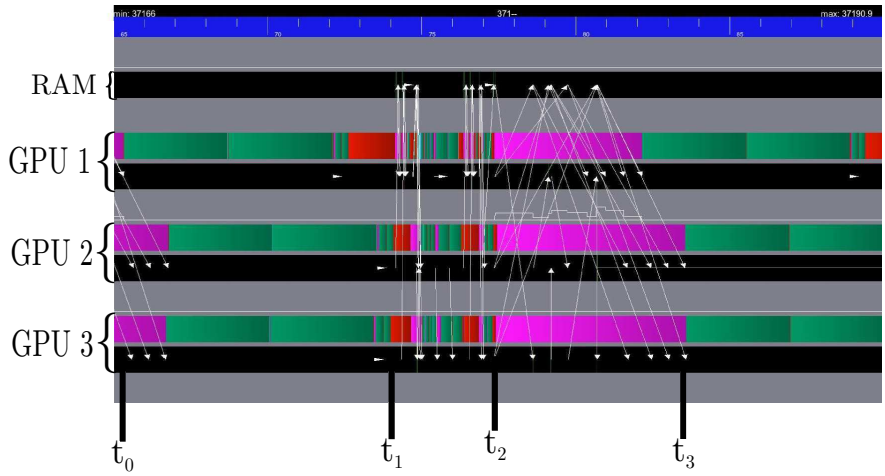


Figure 7: Execution trace of an iteration after furthermore avoiding data partitioning and unpartitioning, and moving instruction 10 after instruction 12.

Figure 7 illustrates the benefits of this policy. Avoiding the unpartitioning operation allows us to decrease the time required between two successive iterations from 8.8 ms to 6.6 ms. Furthermore, since the partitioning operation is no longer needed, the corresponding synchronization in the task flow control is removed. The corresponding idle time (red part at time t_0 in Figure 6) is removed and instructions 7 and 8 are now pipelined (period $[t_0,t_1]$ in Figure 7).

Coming back to Figure 6, one may notice that GPUs are idle for a while just before time t_1 and again just before time t_2. This is due to the reduction that finalizes each *dot* operation ($dot(p, q)$ at instruction 8 and $dot(r, r)$ at instruction 12, respectively). In Algorithm 1, vector x is only used at lines 10 (in read-write mode) and 6 (in read-only mode). The execution of instruction 10 can thus be moved anywhere within the iteration as long as the other input data of instruction 9, i.e., p and α, have been updated to the correct value. In particular, instruction 10 can be moved after instruction 12. This delay enables us to overlap the final reduction of the *dot* occurring at instruction 12 with the computation of vector x.

The red part before t_2 in Figure 6 becomes (partially) green in Figure 7. We do not have a similar opportunity to overlap reduction finalizing the *dot* operation at instruction 8. This is why the red part before t_1 in Figure 6 remains red in Figure 7.

7.3 Reducing communication volume by packing data

By avoiding data partition and data unpartition operations, the broadcast of vector p has been improved (from period $[t_2, t_4]$ in Figure 6 instead of period $[t_3, t_4]$ in Figure 7) and the communication time remains the last main performance bottleneck (time interval $[t_3, t_4]$ in Figure 7). This volume of communication can be decreased. Indeed, if a column within the block-row A_i is zero, then the corresponding element in p does not affect the numerical result of the task $q_i \leftarrow A_i p$. Therefore, p can be pruned. With a SPMD model, using MPI, the natural method would consist in packing the relevant data into buffers before performing the associated communication.

We now explain how we achieve a similar behavior with a task flow model. Instead of broadcasting the whole vector p on every GPU, we only transfer the required subset. Before executing the CG iterations, this subset is identified with a symbolic preprocessing step. Based on the structure of the block $A_{i,j}$, we determine which part of p_j is needed to build q_i. If p_j is not fully required, we do not transfer it directly. Instead, we pack it into an intermediate data, $p_{i,j}$. According to the task-based model, this packing operation is implemented with a task. The receiving GPU then needs to unpack this data before performing the SpMV (since SpMV operates on a full vector). One possibility would be to implement this unpack operation with a new task. In order to limit the overhead due to the task creation, we alternatively merge this unpack operation with the $SpMV$ operation, resulting in a $sparse\_SpMV$ task, where the input vector is now sparse.

Figure 8 shows the resulting pipeline. The time interval $[t_3, t_4]$ in Figure 7 needed for the broadcasting of the vector p has been reduced to the interval $[t_0, t_1]$ in Figure 8. In the rest of the paper we refer to either the $full$ algorithm, where all the block are sent or the *packed* algorithm if this feature is used.

7.4 2D decomposition

The 1D decomposition scheme requires that for each $SpMV$ task, all blocks of vector p (packed or not packed) are in place before starting the execution of the task. In order to be able to overlap the time needed for broadcasting the vector p (time interval $[t_0, t_1]$ in Figure 8) a 2D decomposition must be applied to the matrix. The matrix is first divided in block-rows, and then the same decomposition is applied to the other dimension of the matrix. Similarly as for a 1D decomposition, the entire block-row will be mapped on a single GPU. Contrary to the 1D decomposition, where we had to wait for the transfer of all missing blocks of the vector p, with the 2D decomposition, time needed for the transfer of the vector p can be overlapped with the execution of the tasks for which the blocks of the vector p are already in place.

The result of the impact of a 2D decomposition is shown in Figure 9. During the time interval

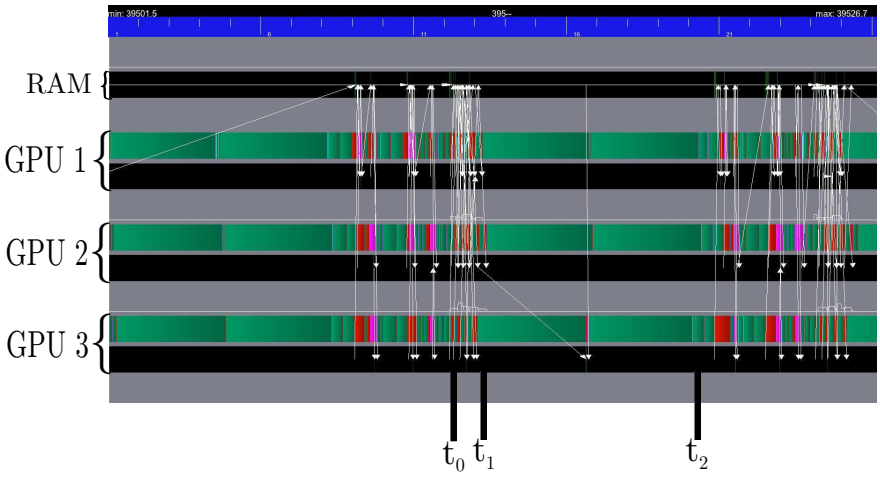


Figure 8: Execution trace when furthermore the vector p is packed.

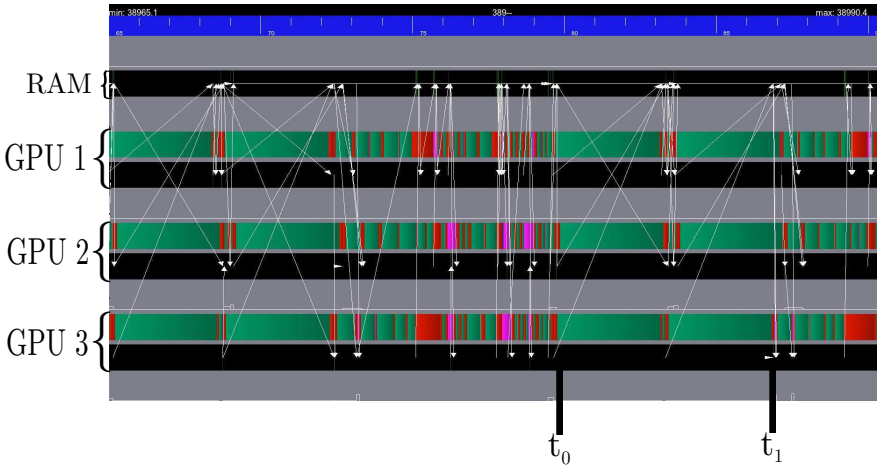


Figure 9: Execution trace when relying of a 2D decomposition of the matrix.

$[t_1, t_2]$ in Figure 8 there is no communication. All of them are done before the beginning of the $SpMV$ task (time interval $[t_0, t_1]$). In Figure 9 during the time interval $[t_0, t_1]$, communications are performed while the $SpMV$ is executed. In the rest of the paper we refer to either 1D or 2D depending on the data decomposition used. The trade-off between large task granularity (1D) and increased pipeline (2D) will be assessed in Section 8.

7.5 Portability

Different data transfers between GPUs are possible with the latest versions of CUDA (see Section 5.1). Depending on the transfer mode, data need to be explicitly copied first to the main RAM (GPU-CPU-GPU mechanism) or may be directly copied from a GPU RAM to another GPU RAM (GPU-GPU mechanism). Relying on a task-based programming model allows us to delegate the management of these complex low-level details to the runtime system. While the GPU-CPU-GPU mechanism has been used so far, we now assess our algorithm with the GPU-GPU mechanism. We use exactly the same high-level algorithm (and code) and only turn on the alternative data transfer mode when compiling the runtime system.

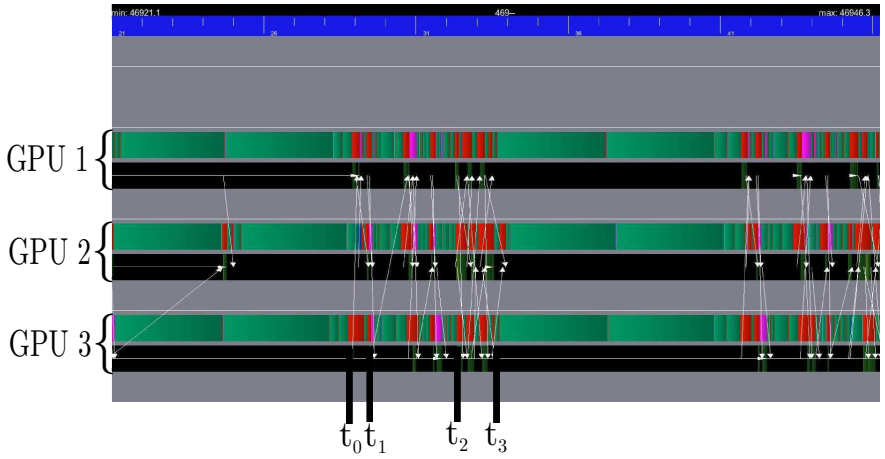


Figure 10: Execution trace when relying on GPU-GPU transfers (1D decomposition case).

Figure 10 shows the resulting behaviour when GPU-GPU transfers are turned on. Now, communications are performed directly from a GPU RAM to another GPU RAM as represented with white arrows in Figure 10. This mechanism, combined with all other presented optimizations, enables us to further limit the idle (red) and active waiting (purple) periods. The only two idle regions left in Figure 10 are the time intervals $[t_0,t_1]$ and $[t_2,t_3]$, which correspond to the reduction finalizing the dot at instruction 8 in Algorithm 1 and the communications for the vector p, respectively.

8 Performance assessment

Tables 2, 3 and 4 present the performance achieved for all three matrices considered in this paper. The best performance is represented for each scheme. We recall that the objective of the paper is to assess the benefits of using multiple GPUs over a mono-GPU execution. We thus discuss the results in terms of metrics which relate these benefits.

		1D		2D	
Communications	# GPUs	full	packed	full	packed
GPU - CPU - GPU	1	**5.851**			
	2	7.863	11.437	7.481	10.801
	3	8.800	**16.821**	8.830	14.770
GPU - GPU	1	5.845			
	2	9.513	11.435	8.861	10.810
	3	8.125	**16.910**	8.660	14.784

Table 2: Performance (in Gflop/s) of CG on the 11pts-256 matrix.

		1D		2D	
Communications	# GPUs	full	packed	full	packed
GPU - CPU - GPU	1	**5.708**			
	2	7.320	10.141	6.874	9.107
	3	7.981	**13.846**	7.250	12.048
GPU - GPU	1	**5.581**			
	2	7.302	10.269	7.676	9.467
	3	6.342	**13.878**	7.484	12.067

Table 3: Performance (in Gflop/s) of CG on the 11pts-128 matrix.

		1D		2D	
Communications	# GPUs	full	packed	full	packed
GPU-CPU-GPU	1	**10.162**			
	2	15.037	15.312	11.372	11.493
	3	17.872	**18.452**	12.539	12.368
GPU-GPU	1	**10.147**			
	2	15.172	15.257	11.971	13.069
	3	15.431	**18.202**	13.056	15.405

Table 4: Performance (in Gflop/s) of CG on the *audikw_1* matrix.

The matrix is decomposed into block-rows in order to ensure concurrency. If we note $T_b(p)$ the elapsed time spent with p GPUs in one execution of the algorithm when the matrix is decomposed in b block-rows, the speed up is thus defined as:

Definition 1. $S = T_1(1)/T_b(p)$

The global efficiency is thus defined as:

Definition 2. $e = T_1(1)/(T_b(p) * p)$.

The efficiency may not be maximum (i.e., $e < 1$) for two reasons. First data need to be decomposed in order to have concurrent tasks. The drawback of this decomposition is that we operate at a smaller granularity and is appreciable in the mono-GPU case: using one GPU, the execution time ($T_b(1)$) when data is decomposed in b blocks is likely to be longer than the execution time ($T_1(1)$) when data is not split and tasks act on the complete matrix. The measure of this penalty due to the fact we operate at a lower granularity on the efficiency may be quantified by the following efficiency:

Definition 3. $e_{granularity} = T_1(1)/T_b(1)$.

Second the tasks may or may not be well pipelined. An execution on p GPUs with b blocks may last longer ($T_b(p)$) than the ideal execution of the same task flow ($T_b(1)/p$) if GPUs are not fully occupied. This effect on efficiency due to a limited pipeline is defined as:

Definition 4. $e_{pipeline} = T_b(1)/(T_b(p) * p)$

The global efficiency e is the result of composing those two effects:

Definition 5. $e = e_{granularity} \times e_{pipeline}$.

We present these values in Table 5 for both GPU-CPU-GPU and GPU-GPU communication mechanisms in the case of a 1D decomposition. Note that the communication mechanism does not impact the effects of the granularity. Dividing the 11pts-256 matrix in several block-rows does not induce a penalty on the task granularity ($e_{granularity} = 0.995 \approx 1$). Furthermore, thanks to all the improvements of the task flow proposed in Sections 7.2 and 7.3 a pipeline efficiency of $e_{pipeline} = 0.969$ is achieved. All in all, a global efficiency of $e = 0.964$ is obtained. For the 11pts-128 matrix, the matrix decomposition induces a higher (but still decent) penalty $e_{granularity} = 0.904$ is a direct consequence of the matrix size. The sum of the sequential time spent in all tasks is increased when the matrix in divided in several blocks as it can be observed in Figure 11. This results in a global efficiency of $e = 0.829$. Finally, the global efficiency for the audikw_1 matrix is lower ($e = 0.598$) than the one for the 11pts-128 matrix despite the fact that they have similar granularity efficiency ($e_{granularity} = 0.897$ for audikw_1). The reason for this is the efficiency of pipeline for the audikw_1 matrix ($e_{pipeline} = 0.667$) that can be explained by a poorer load balancing due to the very irregular pattern of the matrix that is more complex to split evenly. Since the *SpMV* operation is the most costly operation in terms of execution time in the CG algorithm, the global efficiency is influenced by the efficiency of the *SpMV* operation. The efficiency of the *SpMV* operation for this matrix is $e_{SpMV} = 0.862$ (corresponding to the 2.59 speed-up mentioned in Section 6.2). The *SpMV* operation at line 7 in Algorithm 1 is followed by a *dot* operation, which introduces a synchronization point. This synchronization point coupled with the static task mapping (as explained in Section 5.3) leads to an imbalance that cannot be recovered.

In all cases, the 1D algorithms outperformed the 2D algorithms. With the 2D algorithm proposed in Section 7.4, when the matrix is decomposed in b block-rows, the SpMV is decomposed

Matrix	11pts-256	11pts-128	audikw_1
$S_{GPU-CPU-GPU}$	2.875	2.426	1.816
$S_{GPU-GPU}$	2.893	2.487	1.794
$e_{GPU-CPU-GPU}$	0.958	0.809	0.605
$e_{GPU-GPU}$	0.964	0.829	0.598
$e_{granularity}$	0.995	0.904	0.897
$e_{pipeline_{GPU-CPU-GPU}}$	0.963	0.894	0.675
$e_{pipeline_{GPU-GPU}}$	0.969	0.917	0.667

Table 5: Speed-up and efficiency. If $T_b(p)$ represents the execution time with p GPUs when the matrix is decomposed in b block-rows, S the speed-up, e the overall efficiency, the effects of granularity on efficiency $e_{granularity}$ and the effects of pipeline on efficiency $e_{pipeline}$ are defined with Definition 1, 2, 3 and 4 respectively. In our case $p = 3$ and all values in this table are obtained using the algorithm which yields the best overall performance (bold values in tables 2, 3 and 4).

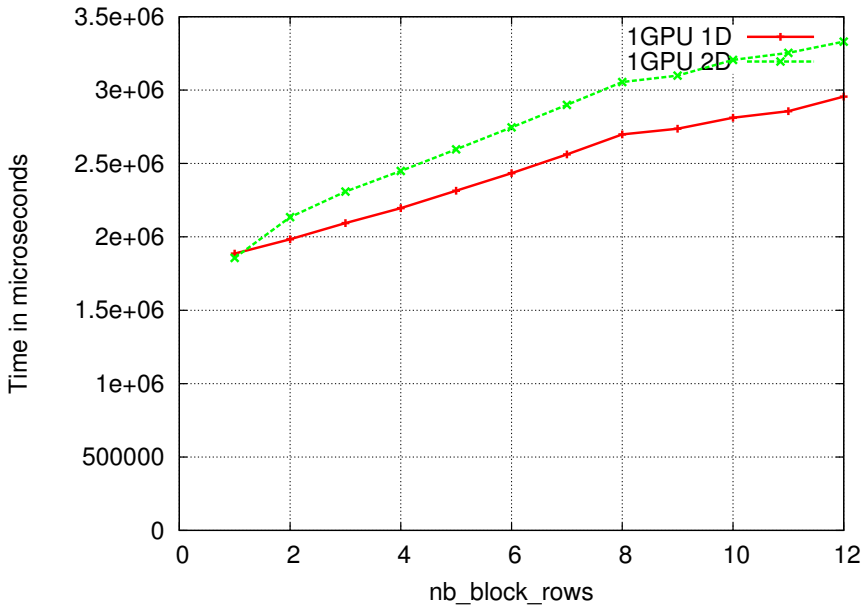


Figure 11: Sum of sequential time spent in kernels for the 11pts-128 matrix.

in b^2 tasks. Figure 11 shows that the subsequent effects on the cumulated time for executing all the tasks is much more tremendous than in 1D. This is mainly due to the fact that the $SpMV$ kernel from CUSPARSE is better suited for matrices with more non-zero per row. In this configuration, the effects of granularity on efficiency are so important in the 2D case that they prevent the 2D algorithm to outperform the 1D algorithm in spite of a potentially better pipeline. For instance, for the 11pts-128 matrix with GPU-GPU transfers, the effects of granularity in the 2D case ($e_{granularity-2D} = 0.804$, not reported in Table 5 since it is outperformed by the 1D case $e_{granularity-1D} = 0.904$) are stronger that the cumulated effects of pipeline and granularity in the 1D case ($e_{1D} = 0.829$).

Table 5 also demonstrates the performance portability of the task-based approach we employed. Indeed, the efficiency is high for both GPU-CPU-GPU and GPU-GPU mechanisms. Let us now compare GPU-CPU-GPU to GPU-GPU transfers. GPU-GPU transfers are about 30% faster than GPU-CPU-GPU transfers. On the other hand, broadcast-like communications (here, when p is broadcasted before SpMV) are faster with the GPU-CPU-GPU mechanism since the runtime system uses the intermediate copy on main RAM memory to perform the actual broadcast from there. If point-to-point communications are dominating, such as with the 11pts-256 and 11pts-128 matrices, the GPU-GPU transfer mode outperforms the GPU-CPU-GPU mode. On the contrary, if the broadcast-like operations are dominating, such as with the audikw_1 matrix, the GPU-CPU-GPU mode outperforms the other scheme. Indeed, for this latter matrix, even with a packed scheme, the volume of data exchange to assemble p is high.

9 Conclusion

We have proposed a task-based formulation of the CG algorithm. We managed to refine the task flow up to removing almost all synchronizations and ensuring an efficient pipelining of the task flow such that GPUs are almost fully exploited. There is a trade-off between task granularity and concurrency, controlled by the number of block-rows. Our performance assessment has shown that the optimum case consistently corresponds to a perfect matching between the number of blocks and the number of GPUs, meaning that benefits in terms of scheduling opportunities by splitting the matrix in a larger number of blocks are by far dominated by the penalty of operating at a smaller granularity. Contrary to dense linear algebra, where state-of-the art dynamic scheduling algorithms are sufficient to achieve nearly optimum performance, scheduling opportunities are thus much more constrained for CG. As a result, each individual scheduling decision may be fatal to the overall performance and we have proposed a carefully defined static scheduling algorithm to prevent such effects. This statement also tells us what is (and what is not) a runtime system. A runtime system is a software layer that allows the user to express *what* to do (which task flow, which scheduling algorithm, ...) and delegate the question of *how* to do it (and how to do it efficiently) to a third party. This approach also ensures performance portability of the code, as illustrated with the benefits of being immediately able to run the code with another low-level communication mechanism and still fully exploit the potential of the platform.

We managed to design a task flow such that all tasks are almost perfectly pipelined. However, we have exhibited that a synchronization point remains (the task graph has a width equal to one after the first *dot*). This synchronization is due to the numerical dependencies in CG. We have shown that an efficient scheduling may almost fully hide the penalty due to this synchronization point. However, if the execution of the SpMV tasks is not well balanced, the synchronization may then prevent us to perform an efficient pipeline, which significantly impacts the performance, as we have shown with the audikw_1 matrix. To overcome this limit, one possibility would consist

of splitting the matrix with finer criterion (for instance, balancing the actual execution time of the SpMV occurring on each block-row) than the basic one we have used, based on balancing the number of non zero elements in each block-row. Another possibility to go further would be to consider another mathematical formulation of CG where the pipelining is expressed not within each iteration as proposed in this work, but between the iterations. Such a mathematical reformulation was first proposed for GMRES [26] and recently extended to CG in [27].

This work will be extended to other standard Krylov methods in order to design a task-based library. Ultimately, we aim at designing task-based algebraic domain decomposition methods similar to [28]. Task-based Krylov methods will thus enable us to pipeline the computation of the preconditioner with the Krylov iterations. We also plan to assess our solvers on multicore platforms enhanced with GPUs, using both CPU cores and GPUs to fully benefit from heterogeneous machines.

References

[1] MAGMA Users' Guide, version 0.2. http://icl.cs.utk.edu/magma, November 2009.

[2] PLASMA Users' Guide, Parallel Linear Algebra Software for Multicore Architectures, Version 2.0. http://icl.cs.utk.edu/plasma, November 2009.

[3] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. volume Vol. 180.

[4] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Julien Langou, Hatem Ltaief, and Stanimire Tomov. LU factorization for accelerator-based systems. In Howard Jay Siegel and Amr El-Kadi, editors, *The 9th IEEE/ACS International Conference on Computer Systems and Applications, AICCSA 2011, Sharm El-Sheikh, Egypt, December 27-30, 2011*, pages 217–224. IEEE, 2011.

[5] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Hatem Ltaief, Samuel Thibault, and Stanimire Tomov. QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators. In *IPDPS*, pages 932–943. IEEE, 2011.

[6] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In Wen mei W. Hwu, editor, *GPU Computing Gems*, volume 2. Morgan Kaufmann, September 2010.

[7] Gabrielle Allen, Jaroslaw Nabrzyski, Edward Seidel, G. Dick van Albada, Jack Dongarra, and Peter M. A. Sloot, editors. *Computational Science - ICCS 2009, 9th International Conference, Baton Rouge, LA, USA, May 25-27, 2009, Proceedings, Part I*, volume 5544 of *Lecture Notes in Computer Science*. Springer, 2009.

[8] Marco Ament, Günter Knittel, Daniel Weiskopf, and Wolfgang Straßer. A parallel preconditioned conjugate gradient solver for the Poisson problem on a multi-GPU platform. In Marco Danelutto, Julien Bourgeois, and Tom Gross, editors, *PDP*, pages 583–592. IEEE Computer Society, 2010.

[9] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, February 2011.

[10] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, February 2011.

[11] Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí. An extension of the StarSs programming model for platforms with multiple GPUs. In Henk J. Sips, Dick H. J. Epema, and Hai-Xiang Lin, editors, *Euro-Par*, volume 5704 of *Lecture Notes in Computer Science*, pages 851–862. Springer, 2009.

[12] Rosa M. Badia, José R. Herrero, Jesús Labarta, Josep M. Pérez, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. Parallelizing dense and banded linear algebra libraries using SMPSs. *Concurrency and Computation: Practice and Experience*, 21(18):2438–2456, 2009.

[13] Jacques Bahi, Raphaël Couturier, and Lilia Ziane Khodja. Parallel sparse linear solver GMRES for GPU clusters with compression of exchanged data. In *HeteroPar'11, 9-th Int. Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms*, LNCS, Bordeaux, France, August 2011. Springer. To appear.

[14] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008.

[15] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC*. ACM, 2009.

[16] Pieter Bellens, Josep M. Pérez, Felipe Cabarcas, Alex Ramírez, Rosa M. Badia, and Jesús Labarta. CellSs: Scheduling techniques to better exploit memory hierarchy. *Scientific Programming*, 17(1-2):77–95, 2009.

[17] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Hérault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, Piotr Luszczek, Asim YarKhan, and Jack Dongarra. Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA. In *IPDPS Workshops*, pages 1432–1441. IEEE, 2011.

[18] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Hérault, Pierre Lemarinier, and Jack Dongarra. DAGuE: A generic distributed dag engine for high performance computing. *Parallel Computing*, 38(1-2):37–51, 2012.

[19] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurrency and Computation: Practice and Experience*, 20(13):1573–1590, 2008.

[20] Ali Cevahir, Akira Nukada, and Satoshi Matsuoka. Fast conjugate gradients with multiple GPUs. In Allen et al. [7], pages 893–903.

[21] Ali Cevahir, Akira Nukada, and Satoshi Matsuoka. High performance conjugate gradient solver on multi-GPU clusters using hypergraph partitioning. *Computer Science - R&D*, 25(1-2):83–91, 2010.

[22] Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. Technical report, 2010.

[23] NVIDIA Corporation. NVIDIA CUDA best practices guide version 5.0, 2012.

[24] Raphaël Couturier and Stéphane Domas. Sparse systems solving on GPUs with GMRES. *The Journal of Supercomputing*, 59(3):1504–1516, 2012.

[25] Serban Georgescu and Hiroshi Okuda. Conjugate gradients on multiple GPUs. *International Journal for Numerical Methods in Fluids*, 64(10-12):1254–1273, 2010.

[26] P. Ghysels, T.J. Ashby, K. Meerbergen, and W. Vanroose. Hiding global communication latency in the GMRES algorithm on massively parallel machines. to appear in SIAM SISC.

[27] P. Ghysels and W. Vanroose. Hiding global synchronization latency in the preconditioned Conjugate Gradient algorithm. Technical Report 12.2012.1, Intel ExaScience Lab Flanders, Leuven, December 2012.

[28] Azzam Haidar. *On the parallel scalability of hybrid linear solvers for large 3D problems*. PhD thesis, HAL - CCSD, December 17 2008.

[29] Timothy D. R. Hartley, Erik Saule, and Ümit V. Çatalyürek. Improving performance of adaptive component-based dataflow middleware. *Parallel Computing*, 38(6-7):289–309, 2012.

[30] Everton Hermann, Bruno Raffin, François Faure, Thierry Gautier, and Jérémie Allard. Multi-GPU and multi-CPU parallelization for interactive physics simulations. In Pasqua D'Ambra, Mario Rosario Guarracino, and Domenico Talia, editors, *Euro-Par (2)*, volume 6272 of *Lecture Notes in Computer Science*, pages 235–246. Springer, 2010.

[31] Laxmikant V. Kalé and Sanjeev Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In *OOPSLA*, pages 91–108, 1993.

[32] David M. Kunzman and Laxmikant V. Kalé. Programming heterogeneous clusters with accelerators using object-based programming. *Scientific Programming*, 19(1):47–62, 2011.

[33] Jakub Kurzak, Hatem Ltaief, Jack Dongarra, and Rosa M. Badia. Scheduling dense linear algebra operations on multicore processors. *Concurrency and Computation: Practice and Experience*, 22(1):15–44, 2010.

[34] Ruipeng Li, Hector Klie, Hari Sudan, and Yousef Saad. Towards Realistic Reservoir Simulations on Manycore Platforms. *SPE Journal*, pages 1–23, 2010.

[35] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In David H. Albonesi, Margaret Martonosi, David I. August, and José F. Martínez, editors, *MICRO*, pages 45–55. ACM, 2009.

[36] Alexander Monakov and Arutyun Avetisyan. Implementing blocked sparse matrix-vector multiplication on NVIDIA GPUs. In Koen Bertels, Nikitas J. Dimopoulos, Cristina Silvano, and Stephan Wong, editors, *SAMOS*, volume 5657 of *Lecture Notes in Computer Science*, pages 289–297. Springer, 2009.

[37] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. Automatically tuning sparse matrix-vector multiplication for GPU architectures. In Yale N. Patt, Pierfrancesco Foglia, Evelyn Duesterwald, Paolo Faraboschi, and Xavier Martorell, editors, *HiPEAC*, volume 5952 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 2010.

[38] A. Munshi. The OpenCL specification, khronos opencl working group, version 1.1, revision 44, 2011.

[39] Tomás Oberhuber, Atsushi Suzuki, and Jan Vacata. New row-grouped CSR format for storing the sparse matrices on GPU with implementation in CUDA. *CoRR*, abs/1012.2270, 2010.

[40] G. Quintana-Ortí, E. S. Quintana-Ortí, E. Chan, F. G. Van Zee, and R. A. van de Geijn. Scheduling of QR factorization algorithms on SMP and multi-core architectures. In *Proceedings of PDP'08*, 2008. FLAME Working Note #24.

[41] Gregorio Quintana-Orti, Francisco D. Igual, Enrique S. Quintana-Orti, and Robert van de Geijn. Solving dense linear algebra problems on platforms with multiple hardware accelerators. *FLAME Working Notes*, flawn32, 2008.

[42] Gregorio Quintana-Ortí, Francisco D. Igual, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Solving dense linear systems on platforms with multiple hardware accelerators. *ACM SIGPLAN Notices*, 44(4):121–130, April 2009.

[43] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, 2007.

[44] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[45] H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, Mar 2002.

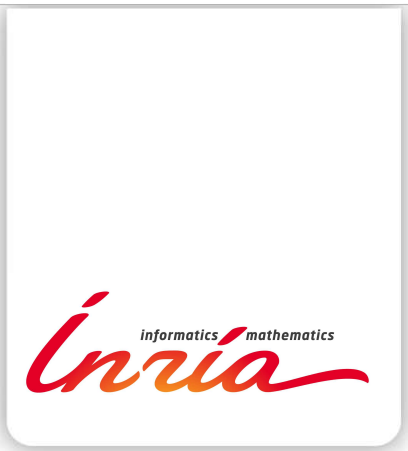
[46] Field G. Van Zee, Ernie Chan, Robert A. van de Geijn, Enrique S. Quintana-Orti, and Gregorio Quintana-Orti. The `libflame` Library for Dense Matrix Computations. *Computing in Science and Engineering*, 11(6):56–63, November/December 2009.

[47] Mickeal Verschoor and Andrei C. Jalba. Analysis and performance estimation of the conjugate gradient method on multiple GPUs. *Parallel Computing*, 38:552 – 575, 2012.

[48] Mingliang Wang, Hector Klie, Manish Parashar, and Hari Sudan. Solving sparse linear systems on NVIDIA Tesla GPUs. In Allen et al. [7], pages 864–873.

[49] W.A. Wiggers, V. Bakker, A.B.J. Kokkeler, and G.J.M. Smit. Implementing the Conjugate Gradient algorithm on multi-core systems. In *System-on-Chip, 2007 International Symposium on*, pages 1 –4, nov. 2007.

[50] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 38:1–38:12, New York, NY, USA, 2007. ACM.



RESEARCH CENTRE
BORDEAUX – SUD-OUEST

351, Cours de la Libération
Bâtiment A 29
33405 Talence Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399