Task-based FMM for heterogeneous architectures

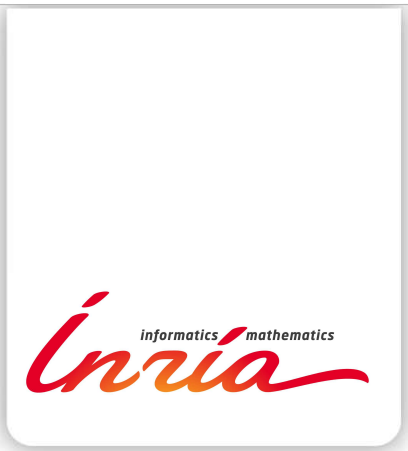
Emmanuel Agullo, Bérenger Bramas, Olivier Coulaud, Eric Darve, Matthias Messner, Toru Takahashi

Task-based FMM for heterogeneous architectures

Emmanuel Agullo, Bérenger Bramas, Olivier Coulaud, Eric Darve, Matthias Messner, Toru Takahashi

RESEARCH REPORT
N° 8513
December 2013
Project-Teams Hiepacs

ISSN 0249-6399 ISRN INRIA/RR--8513--FR+ENG



Task-based FMM for heterogeneous architectures

Emmanuel Agullo*, Bérenger Bramas*, Olivier Coulaud*,
Eric Darve†, Matthias Messner*, Toru Takahashi‡

Project-Teams Hiepacs

Research Report n° 8513 — December 2013 — 29 pages

Abstract: High performance Fast Multipole Method (FMM) is crucial for the numerical simulation of many physical problems. In a previous study [1], we have shown that task-based FMM provides the flexibility required to process a wide spectrum of particle distributions efficiently on multicore architectures. In this paper, we now show how such an approach can be extended to fully exploit heterogeneous platforms. For that, we design highly tuned GPU versions of the two dominant operators (P2P and M2L) as well as a scheduling strategy that dynamically decides which proportion of subsequent tasks are processed on regular CPU cores and on GPU accelerators. We assess our method with the StarPU runtime system for executing the resulting task flow on an Intel X5650 Nehalem multicore processor possibly enhanced with one, two or three Nvidia Fermi M2070 or M2090 GPUs. A detailed experimental study on two 30 million particle distributions (a cube and an ellipsoid) shows that the resulting software consistently achieves high performance across architectures.

Key-words: Fast multipole methods, graphics processing unit, heterogeneous architectures, runtime system, scheduling, pipeline.

* Inria, Hiepacs Project, 350 cours de la Libération, 33400 Talence, France. Email: Surname.Name@Inria.fr
† Mechanical Engineering Department, Institute for Computational and Mathematical Engineering, Stanford University, Durand-209, 496 Lomita Mall, Stanford CA 94305-4040, USA. Email: darve@stanford.edu
‡ Department of Mechanical Science and Engineering, Nagoya University, Japan. Email: ttaka@nuem.nagoya-u.ac.jp

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Méthodes multipôles rapides à base de tâches pour architectures hétérogènes

Résumé :

Développer une méthode des Multipôles Rapide (FMM) à haute performance est cruciale pour des simulations numériques dans beaucoup de problèmes physiques. Dans une étude précédente [1], nous avons montré que l'utilisation d'un paradigme à base de tâches fournit la flexibilité nécessaire pour traiter efficacement un large spectre de distributions de particules sur des architectures homogènes. Dans ce document, nous montrons maintenant comment une telle approche peut être étendue pour exploiter toutes les unités de calculs (CPU et GPU) des machines hétérogènes. Pour cela, nous présentons une version optimisée pour GPU des deux opérateurs dominants (P2P et M2L) de la FMM ainsi qu'une stratégie d'ordonnancement qui décide dynamiquement quelle proportion de tâches est traitée par les cœurs CPU et par des accélérateurs GPU. Nous évaluons notre méthode avec le moteur d'exécution StarPU pour exécuter le flot de tâches résultant sur un processeur Intel X5650 Nehalem augmenté avec un, deux ou trois Nvidia Fermi M2070 ou M2090. Une étude expérimentale détaillée sur deux distributions de 30 millions de particules (un cube et un ellipsoïde) montre que nous obtenons des résultats performants sur cette architecture.

Mots-clés : Méthodes multipôles rapides, processeur graphique, architectures hétérogènes, moteur d'exécution, ordonnancement, pipeline.

Contents

1 Introduction

The Fast Multipole Method (FMM) [2] is one of the most prominent algorithms to perform pairwise particle interactions, with applications in many physical problems such as astrophysical simulations, molecular dynamics, the boundary element method, radiosity in computer-graphics or dislocation dynamics. Its linear complexity makes it an excellent candidate for large-scale simulations [3]. With the advent of the many-core era, the High Performance Computing (HPC) community has studied the design and performance of the method on Central Processing Unit (CPU) multicore [4, 5, 6, 7] and Graphics Processing Unit (GPU) [8, 9, 10, 11, 12, 13] architectures. However, achieving high performance requires hand-tuning, both at the level of individual routines (e.g., GPU kernels) and to efficiently manage the concurrent tasks and their execution. Consequently, porting a code from one architecture to another is a time-consuming and difficult process. To cope with this drawback, the FMM may be formulated as a high-level task-based parallel algorithm independent of the underlying architecture. Such a paradigm has been studied during the past five years primarily by the dense linear algebra community [14, 15, 16, 17, 18, 19, 20] resulting in new production solvers such as Plasma, Magma [21] or Flame [22]. This paradigm has also been assessed in the context of more irregular algorithms involved in sparse direct [23, 24] and sparse iterative [25] solvers. Task-based fast solvers have also been investigated in the context of multicore architectures [1, 26, 27, 28].

In a previous study [1], we have shown in the context of multicore architectures that a task-based FMM formulation is not only competitive against more widely employed formulations specific to the architecture, but also that the flexibility provided by this paradigm enables one to achieve a near-optimum parallel efficiency on a large spectrum of particle distributions. In this article, we now show that such a task-based paradigm can be employed to make the most of heterogeneous architectures. Task-based programming is trying to find a middle ground by providing advanced parallel scheduling schemes and a flexible creation and management of concurrent tasks, while striving to be a high-level of abstraction compared to OpenMP or Pthreads programmation. Therefore, it has the potential to generate high-performance codes with high productivity. So far, that paradigm has been mostly confined to the sole runtime system community, until the linear solver community started adopting it more widely in recent years as mentioned above. As a result, many research projects propose both an API and a runtime to support task-based formulations. An exhaustive presentation is out of the scope of this paper, but we briefly mention some of the more prominent projects onto which the above mentioned solvers have been designed: SMPSs [29], StarPU [30], PaRSEC [31], CnC [32], Quark [33] and SuperMatrix [34]. These have been used in the context of dense, sparse or fast linear solvers. Following [1], we chose to illustrate our methodology with the StarPU runtime system for mainly two reasons. First, StarPU (read $\star$PU) was designed from the get-go to support heterogeneous processors (as the name indicates), and it allows for instantiating tasks on any processing unit (PU) such as a CPU core or a GPU. Second, StarPU proposes an elegant mechanism for writing parallel scheduling algorithms as third party modules (Section 2.3), which is essential for achieving high performance (Section 4.3).

We design our heterogenous task-based FMM by extending [1] as follows. We refine (Section 4.3.1) the high-level task-based expression (Section 3) proposed in [1]. We rely on the CPU kernels from [1] and we propose new GPU kernels (Section 3.1) for the two dominant operators (P2P and M2L). The P2P kernel (Section 3.1.1) is derived from [35] to cope with the constraints of the task-based expression whereas the M2L kernel (Section 3.1.2) is specifically designed for the purpose of this study. We also design a new scheduling algorithm (Section 3.2) that aims at exploiting the heterogeneity of the architecture. The P2P and M2L operators are dynamically scheduled either on CPUs or GPUs to cope with both the numerical settings (particle distribu-

tion, accuracy, octree height) and the hardware configuration (Intel X5650 Nehalem multicore processor possibly enhanced with one, two or three Nvidia Fermi M2070 or M2090 GPUs). In particular, we illustrate cases where the GPUs run out of P2P tasks and start executing M2L tasks, and other cases where the CPUs start with M2L tasks but then switch to P2P tasks. This is determined by the scheduler at runtime. We assess our method by studying the impact of four independent effects (kernel performance, task granularity, scheduling, and octree setting) on the overall FMM performance. The FMM algorithm is a non-adaptive Chebyshev FMM derived from [36]. Nevertheless, although a tree with a uniform depth is used, we do not perform computation with empty clusters, which allows us to efficiently compute cases where points are distributed on a surface. Here, we illustrate our discussion with particle distributions in the volume of a cube and on the surface of an ellipsoid.

The rest of the paper is organized as follows. In Section 2, we present the building blocks on top of which we design our method: the anti-symmetric Chebyshev FMM (Section 2.1), its task-based expression (Section 2.2) and the StarPU runtime system (Section 2.3) used to support its execution. We then present the design of our heterogeneous task-based FMM in Section 3. We finally assess our method in Section 4.

2 Background

We present the building blocks on top of which we design our method. We first provide a short introduction to the FMM [2, 36](Section 2.1). We then explain how it can be turned into a task graph according to [1] (Section 2.2). This task graph is processed by a runtime system, StarPU in our case, which we introduce in Section 2.3.

2.1 The Fast Multipole Method

Pair-wise particle interactions can be modeled mathematically as

$$f_i = \sum_{j=1}^{N} P(x_i, y_j)\, w_j \quad \text{for } i = 1, \ldots, M. \tag{1}$$

Pairs of particles, denoted as sources and targets, are represented by their spatial position $x, y \in \mathbb{R}^3$. The interaction is governed by the kernel function $P(x, y)$. The above summation can also be understood as a matrix-vector product $\mathsf{f} = \mathsf{P}\mathsf{w}$, where P is a dense matrix of size $M \times N$. Hence, if we assume $M \sim N$, the cost grows like $\mathcal{O}(N^2)$ which becomes prohibitive as $M, N \to \infty$. This is why we use the fast multipole method (FMM) as a fast summation scheme. Our approach is adapted from [36]. In this section, we present the core formulas. The mathematical details will be left out but can be found in [36].

The FMM reduces the cost of computing pair-wise particle interactions to $\mathcal{O}(N)$. It is applicable when the kernel $P(x, y)$ can be approximated using a low-rank approximation in the far-field, that is when sources and targets are well separated.

In order to compute the summation in $\mathcal{O}(N)$, we need to decompose the points x_i and y_j using an octree. Different operators are then applied at all levels of the tree and they can be categorized in two main groups: the near-field and the far-field operators.

The near field operator computes the direct interactions between particles (P2P) by applying Equation (1) between neighbor leaves.

The FMM approximates far interactions by using different operators at all levels of the tree. In this paper we refer to the widely used operators name convention: P2M, M2M, M2L, L2L and L2P, where P, M and L refer to Particle, Multipole and Local respectively.

Although in this paper we focus on the Chebyshev FMM, this work applies to most FMM formulations. Indeed, aside from the mathematical details of the formulas we use for the different operators, each step in the FMM (P2M, M2M, M2L, L2L and L2P) corresponds to matrix-vector products. Similarly P2P corresponds to a direct evaluation of the kernel $P(x_i, x_j)$. Therefore most conclusions extend to other FMM formulations and the code we wrote can be easily reused in other FMM formulations. Only the mathematical formulas for each operator needs to be redefined but all the communication and parallelization strategies would remain mostly the same.

Focusing now on the Chebyshev FMM, the approximation formula for the kernel is based on a Chebyshev interpolation, where we denote S_ℓ the resulting interpolation polynomial of order ℓ:

$$\frac{1}{|x-y|} \sim \sum_{m=1}^{\ell} S_\ell(x, \bar{x}_m) \sum_{n=1}^{\ell} \frac{1}{|\bar{x}_m - \bar{y}_m|} S_\ell(y, \bar{y}_n), \tag{2}$$

$$S_\ell(x, x_m) = \frac{1}{\ell} + \frac{2}{\ell} \sum_{n=1}^{\ell-1} T_n(\bar{x}_m) T_n(x). \tag{3}$$

Finally, the $\mathcal{O}(N^2)$ pair interactions are approximated according to the following equations:

$$f_i = \sum_{j=1}^{N} \frac{w_j}{|x_i - y_j|}, \qquad 1 \le i \le N, \tag{4}$$

$$\sim \underbrace{\sum_{m=1}^{\ell} S_\ell(x_i, \bar{x}_m)}_{\textbf{L2L}} \underbrace{\sum_{n=1}^{\ell} \frac{1}{|\bar{x}_m - \bar{y}_m|}}_{\textbf{M2L}} \underbrace{\sum_{j=1}^{N^{\text{far}}} S_\ell(y_j, \bar{y}_n)\, w_j}_{\textbf{M2M}} + \underbrace{\sum_{j=1}^{N^{\text{near}}} \frac{w_j}{|x_i - y_j|}}_{\textbf{P2P}} \tag{5}$$

Remark. *The presented FMM formulation has two approximations: 1) the interpolation of the kernel functions which is determined by the interpolation order ℓ and 2) the low-rank approximation of the M2L operators is determined by the target accuracy ε. Studies in [37, 38] have shown that the setting $(\ell, \varepsilon) = (Acc, 10^{-Acc})$ results approximately in a relative point-wise L_2 error of $\varepsilon_{L_2} = 10^{-Acc}$. In the rest of the paper and consistently with [1], we use this convention to describe the accuracy Acc.*

2.2 Task-based Fast Multipole Method

In [1], we have proposed different task-based formulation of the Chebyshev FMM presented above. Since operating on individual cells (Figure 1 (left)) does not ensure enough performance, we have defined tasks so that they operate on groups of n_g cells (Figure 1 (right)). As a consequence,

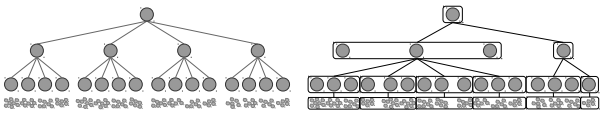


Figure 1: A tree with $n_g = 3$ (one vertex in the tree groups three cells) [1].

we rely on the standard Morton indexing [39] that is both followed within a group (to ensure kernel performance) and between groups (to ensure task locality). This granularity parameter n_g trades off concurrency (small enough n_g, many tasks) and kernel performance (large enough n_g, high kernel performance). Among all the studied task-based formulations based on this grouping

scheme (such as parallelizations per level), we have furthermore shown that the Task FLow (TFL) model provides the required flexibility to achieve a high parallel efficiency on a wide range of particle distributions (see [1] for details). The TFL model can be conveniently represented as a *directed acyclic graph* (DAG) where vertices represent individual tasks and edges correspond to data dependencies among them (Figure 2). We follow this latter scheme in the present study.

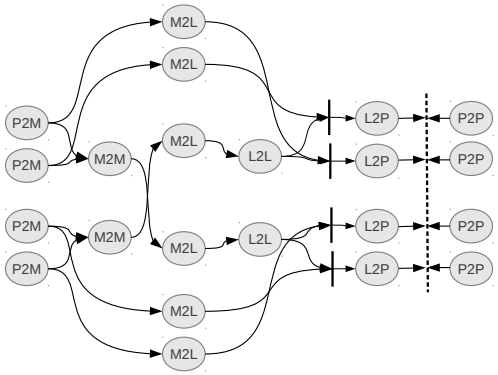


Figure 2: Task flow of the FMM algorithm viewed as a DAG where vertices represent tasks and edges dependencies between tasks. Straight lines represent a writing operation from different tasks on the same data. Dash line separates the near-field and far-field DAGs [1].

2.3 StarPU – A task-based runtime system

Designing a TFL FMM requires to describe the corresponding DAG. This DAG is a *representation* of the application and needs an intermediate software layer to support its execution, which is precisely the purpose of a runtime system. In StarPU (and many other runtime systems), one possibility for building the DAG consists of successively inserting the tasks in an order which is consistent with a valid sequential execution using the following prototype for each task insertion:

```
insertTask(codelet, access_mode_1, handle_1, access_mode_2, handle_2, ...);
```

The *access modes* (`READ`, `WRITE` or `READ_WRITE`) are then used by the runtime system to compute data hazards (race conditions) [40] (Read After Write, Write After Read, Write After Write) and hence the subsequent DAG. The *codelet* defines where a task can be implemented (for instance on CPU and CUDA GPU for P2P) and where to retrieve the code for the corresponding devices:

```
struct starpu_codelet p2p_codelet
{
    .where = STARPU_CPU | STARPU_CUDA
    .cpu_func = p2p_cpu_func,
    .gpu_func = p2p_gpu_func,
    .nbuffers = 2
};
```

The number of arguments (`nbuffers = 2` in our example) is also to be provided; indeed, a task can be executed on any device transparently, so it must respect the exact same API and have the same behavior on each of them. This might be a strong constraint as we will show in the P2P case (Section 3.1.1). As viewed in the signature of the `insertTask` function, the data on which

the tasks operate are not directly provided with pointers but with so-called `handles` which are an abstraction of the data manipulated by the runtime system to maintain all the possible valid copies on the different hardware memory banks (CPU memory, GPU memories). As a result, the algorithm is conveniently provided at a high level whereas only the individual tasks have to be provided for the target hardware. The runtime system furthermore hides the low-level architectural complexity by handling data transfers efficiently[1]. It also ensures that a task starts its execution once all its predecessors in the DAG are completed and that the input data required by the task is present on the selected device.

The selection of the device for executing a task is performed dynamically and several scheduling heuristics are available in the StarPU framework. However, when the application is irregular, generic scheduling strategies may fail to make the most of the available hardware. To comply with this limit, StarPU offers an elegant mechanism to design new scheduling strategies as illustrated in Figure 3. When the user inserts a task using the `insertTask` utility (leftmost part

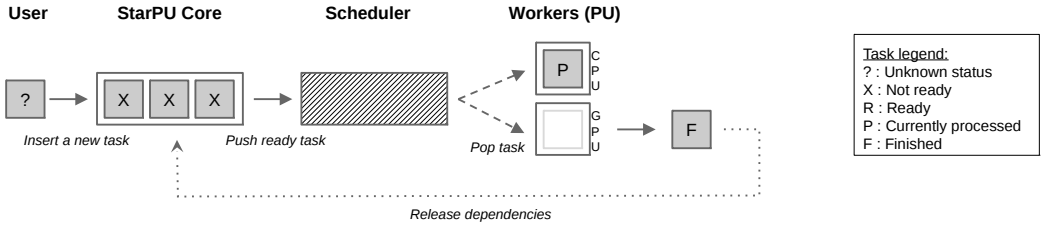


Figure 3: Generic scheduling mechanism in StarPU.

in Figure 3) it is taken in charge by the core of StarPU. When its predecessors are completed, the task becomes ready and it is pushed to another software layer, the scheduler. The scheduler maintains all ready tasks until they are popped by a PU for actual execution. This modular approach allows the programmer to redefine a scheduler by implementing the `push` and `pop` methods. He can also define its own container (dashed rectangle in Figure 3) to maintain ready tasks at its convenience. We have shown the benefits of defining dedicated scheduling strategies for processing FMM on multicore platforms in [1]. We will present a new scheduling algorithm for exploiting heterogeneous archictectures efficiently based on this mechanism in Section 3.2.

3 Heterogeneous Task-based Fast Multipole Method

In this section, we present the design of our heterogeneous task-based FMM. For that we extend the homogeneous approach proposed in [1] and recalled in Section 2, by enhancing it with two GPU kernels (Section 3.1) and a new scheduling algorithm (Section 3.2) that aims at both exploiting the heterogeneity and pipelining the task flow.

3.1 GPU kernels

3.1.1 P2P GPU kernel

P2P is the most critical operator to be ported on GPU as it is dominant in most configurations (and thus need to be accelerated) and has a high arithmetic intensity (and thus has a high

[1]For instance, StarPU detects the closest CPU core to a GPU and selects it to handle that GPU, it also performs asynchronous data transfers when the hardware allows for it and uses advanced drivers for each type of device.

potential for acceleration). We consider P2P GPU kernel from [35] and we now present how we derive it in order to cope with the group data structure introduced in Section 2.2 and efficiently support anti-symmetry in presence of a dynamic choice of the PU.

Contrary to most P2P GPU implementations where all the leaves (or a large amount of them) are available on the device, we remind that we consider a task flow of relatively fine granularity. Leaves are split in n_g consecutive non-empty leaves following the Morton index. Figures 4(a) and 4(b) show how a grid of 11 leaves is split into two groups, ABCDEF and GHIJK, if $n_g = 6$. The strategy for computing the interaction of the forces within a group depends on the type of PU. On a CPU, we exploit the property of anti-symmetry ($F_{ij} = -F_{ji}$) to actually compute only half of the interactions (F_{ij}, $i < j$) and implicitly deduce the other half (F_{ji}, $i < j$). On a GPU, exploiting this property requires to break the control flow and was shown to be slower than computing independently both F_{ij} and F_{ji}. Therefore, as most of GPU implementations do, including [35], our GPU kernel follows this latter strategy.

Some of the leaves inside a group may have neighbors outside their own group (B, C, E and F from group ABCDEF have neighbors G and I from group GHIJK and conversely). Therefore, to fully update their near-field interactions, they need to receive contributions from those neighbors. This creates a special difficulty when developing tasks appropriate for both the CPUs and GPUs. On a GPU, a natural approach would consist in duplicating the required data and using ghost leaves. However, on a CPU, such duplication is not needed and anti-symmetry should be used to reduce the number of Flops.

To resolve this, we have implemented the following strategy in order to exploit as much as possible the anti-symmetry of the kernel. A group is augmented with neighbor leaves only if the neighbor leaves have a higher Morton indexing. In our example, ABCDEF is thus augmented with G and I but GHIJK is not augmented (see Figure 4(c)). As a result, the final update requires an extra-task, that we name $P2P - Reduction$. In our example in Figure 4(c), G and I get updated at that point. We will explain in Section 4.3.1 how the $P2P - Reduction$ task can be processed efficiently in the whole TFL.

With this approach and a pure CPU execution, anti-symmetry is fully exploited. When a GPU is used, redundancy is limited only to the part of the P2P computation performed on the GPU, making the most of heterogeneity.

Note that in the sequel the number of Flopsused to assess performance (see Section 4.1.3) is the one assuming anti-symmetry is exploited (thus minimum), independently of the mapping of the tasks (and thus of the possible extra-Flop when P2P tasks are performed on GPU).

3.1.2 M2L GPU kernel

In this section we discuss our implementation of the M2L operator on the GPU. Despite the regular nature of that operator, it is not well suited for the GPU architecture. In fact, as most of the architecture, GPU are strongly limited by the memory traffic and in the other hand arithmetic throughput is rarely a limitation. If we consider for example a matrix-vector product of size $n \times n$, we need to read $n^2 + n$ numbers in order to perform n^2 multiply-add instructions (which are fused multiplication addition operation supported by GPU). Then, we need about 30 floating point operations per word of 8 bytes (Flop/word ratio) to be able to by-pass the memory bandwidth and to achieve a high Flop rate. A small matrix-vector product with a lower ratio of operations against read words will have a performance far below the peak arithmetic throughput. In contrast, matrix-matrix products, usually expressed as the computation of $C \leftarrow \alpha AB + \beta C$, are much more compute intensive. If we consider state of the art implementations of GEMM the peak performance [41] is obtained by computing a group of 64 rows of A with 64 columns of B. Then a thread is assigned to a 4×4 block of the output matrix C. The computation

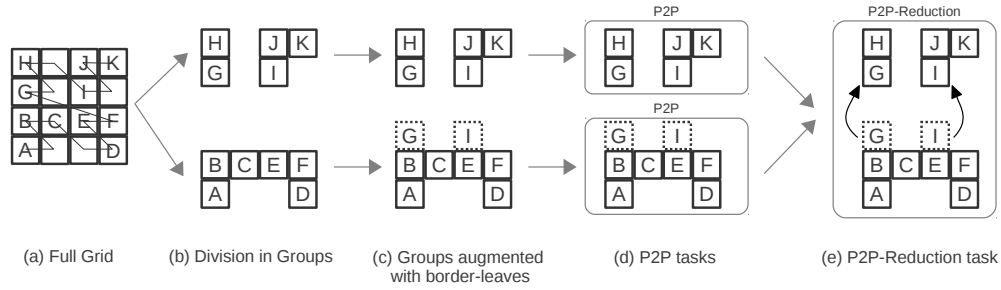


Figure 4: (a) Non-empty leaves ranging from A to K according to Morton indexing. (b) Assuming $n_g = 6$, subdivision into two groups (ABCDEF and GHIJK). The P2P computation for group ABCDEF requires G and I neighbors from the other group. (c) ABCDEF is thus augmented with G and I. Meanwhile, GHIJK is not augmented (the decision of the attribution is based on the Morton indexing). (d) Computation of P2P interactions inside each P2P set performed in concurrent tasks. (e) Updating G and I potentials using $P2P - Reduction$.

with multiple columns of B allows data reuse and increases the arithmetic intensity (Flop/word ratio). However, the performance is again adversely affected for small matrices.

In the case of our formulation of the FMM, the M2L operator is a dense matrix associated with all pairs of cells that need to interact and we denote by M_{IJ} the matrix for a pair (I, J) of cells. Coming from this definition and some numerical optimization we are faced with several difficulties. First of all, we have a large number of matrices, but each has a small size and there is no simple strategy to convert the sequence of matrix-vector products into matrix-matrix products. Then, to reduce the number of Flops, each matrix is compressed using an SVD and factored as detailed in Equation (6) where U_{IJ} and V_{IJ} are thin matrices:

$$\mathsf{M}_{IJ} \approx \mathsf{U}_{IJ}\mathsf{V}_{IJ}^{\top}. \tag{6}$$

Finally, as described in [38], it is possible to use the symmetries in the set of matrices M_{IJ} to reduce their number. For example there is a symmetry with respect to the planes $x = 0$, $y = 0$, $z = 0$ and $x = y$, $y = z$, $x = z$. Accounting for these symmetries, we can identify a set of 16 unique primitive matrices, M_p, $0 \leq p < 16$. Other M_{IJ} matrices are obtained through appropriate permutation of the entries, where the permutations are derived from the symmetry transformations [38]. This implies that the collection of 316 $(= 7^3 - 3^3)$ M_{IJ} M2L matrices can be obtained from M_p. Therefor the original M2L matrix-vector products can be re-organized into only 16 matrix-matrix products as presented in Equation (7), where W_t is a collection of vectors w_J with permutations. The statistics for the number of columns in W_t is shown in table 1.

$$\mathsf{M}_{IJ} \cdot w_J \rightarrow \qquad \mathsf{G}_t = \mathsf{V}_p^{\top} \cdot \mathsf{W}_t, \qquad \mathsf{F}_t = \mathsf{U}_p \cdot \mathsf{G}_t \tag{7}$$

We present the Flop-to-word ratio of our approach, that is, the ratio of arithmetic operations against the data traffic in the case of double precision (where one word is considered as 8 bytes). We took into account the data transaction to permute the multipole and local expansions. Specifically, each permutation uses an integer array of length ℓ^3 to exchange the row indices thus reading such an array requires $\ell^3/2$ words. Figure 5 shows the Flop-to-word ratio of the M2L kernel for various α and ℓ, where α is the maximum number of columns simultaneously processed in W_t. Even with $\alpha = 24$ and $\ell = 7$, the ratio is lower than 17. On the GPUs we use, which are defined in Section 4.1.2, this ratio is too low in order to achieve high performance as

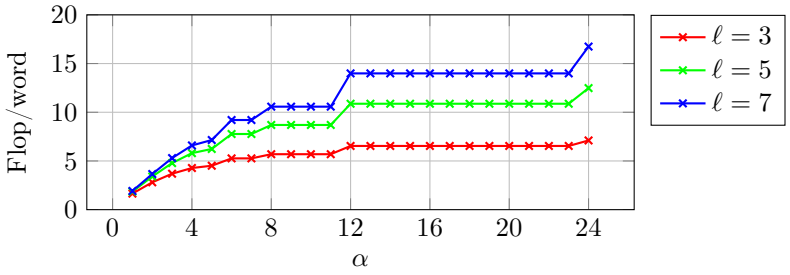


Figure 5: Flop-to-word ratio of the GPU M2L kernel against α (≤ 24). α is the maximum number of columns in W_t that are processed during one matrix-matrix multiplication by the CUDA kernel.

it is presented in Section 4.2. This means that the bottleneck of the current implementation is likely to be the memory transactions. We now assess the CUDA threads assignment where an important criterion is the size of the matrices $\mathsf{V}_p^\top$ and U_p. Recall that the matrix $\mathsf{V}_p^\top$ is fat and the matrix U_p is thin. Specifically, they have the following sizes: $\mathsf{V}_p^\top \in \mathbb{R}^{r\times\ell^3}$ and $\mathsf{U}_p \in \mathbb{R}^{\ell^3\times r}$, where r is the rank used to compress the M2L matrix M_{IJ} using the SVD, and ℓ is the order of the Chebyshev interpolation. The rank r depends on the relative position of I and J and the different values of r are shown in Table 2. The rank needs to be increased with ℓ to guarantee the prescribed tolerance while minimizing the number of Flops.

One possible implementation is to assign a 32-thread warp to perform the multiplication by $\mathsf{V}_p^\top$ and U_p. In that case, the multiplication by U_p is the easiest. In fact, the numbers of rows is equal to ℓ^3 and there are $27, 125$ or 343 rows in the matrix. Thus, each thread in a warp can process multiple rows (except for $\ell = 3$). However for $\mathsf{V}_p^\top$ the numbers of rows is only r, which varies from 4 to 45 (see Table 2). This means that in most cases few threads in a given warp have work. For that reason, it is advantageous to assign multiple threads per row and to perform a reduction at the end in order to produce the final row result. Finally, several optimizations as shared memory, loop unrolling, data prefetching, and register blocking are use to reduce the memory traffic and to improve data transfer and arithmetic performance.

3.2 Scheduling

In [1], we have shown the strong impact of scheduling strategies on performance on homogeneous machines and proposed a so-called MultiPrio heuristic where different types of tasks are assigned different priorities in order to maximize the throughput. In an heterogeneous context, scheduling strategies have potentially even more impact on performance. In the following, we present a new scheduling strategy based on two rules for efficiently processing an FMM task flow on an heterogeneous machine. The first rule is an extension of the MultiPrio heuristic to the heterogeneous case and aims at maximizing the steady state throughput. The second rule specifically takes care of termination (when few enough tasks remain in the system), which is critical on heterogeneous architectures because of potential strong imbalance of the power of the computational resources. In the sequel, this scheduling algorithm will be referred to HeteroPrio.

Steady state. As discussed in Section 2.3, StarPU allows programmers to define their own scheduling strategies by redefining the `push` and `pop` methods as well as the container used to maintain ready tasks. At insertion time, tasks that have both a CPU and a GPU implementation

Index p	Number of columns in W_t
0	6
1	24
2	24
3	12
4	24
5	8
6	3
7	12
8	12
9	12
10	24
11	12
12	3
13	6
14	6
15	1

Table 1: Number of columns in W_t as a function of p. This corresponds to an M2L operator of the form $\mathsf{G}_t = \mathsf{V}_p^\top \mathsf{W}_t$, followed by $\mathsf{F}_t = \mathsf{U}_p \mathsf{G}_t$. Using the symmetries in the M2L operators and appropriate permutation matrices, we were able to reduce the total number of M2L operators from 189 to 16 unique operators (indexed by p). This requires replacing the matrix-vector products with matrix-matrix products with permutations. See [38] and Equation (7).

Index p	Rank		
	$\ell = 3$	$\ell = 5$	$\ell = 7$
0	9	23	45
1	8	18	36
2	7	16	33
3	4	15	25
4	4	13	25
5	4	9	22
6	4	12	25
7	4	11	24
8	4	9	23
9	4	9	19
10	4	9	17
11	4	9	16
12	4	9	16
13	4	9	16
14	4	9	16
15	4	9	16

Table 2: Size of the matrices involved in the M2L operation. We have $\mathsf{V}_p \in \mathbb{R}^{\ell^3 \times r}$ and $\mathsf{U}_p \in \mathbb{R}^{\ell^3 \times r}$. The rank depends on p, which is an index indicating the relative position of the cells I and J involved in the M2L interaction.

(P2P and M2L in our case) get attributed two priorities: one related to CPU, one related to GPU. Leftmost part of Figure 6 illustrates this principle on a simplified example. A FIFO queue is accordingly associated with each pair of priorities (queues $(2,0)$, $(1,2)$ and $(0,1)$ in Figure 6). Altogether these queues (three in our illustration) form the container. As a result, when a task becomes ready, the `push` method simply consists of placing it into the corresponding queue. The `pop` method, performed by each PU, then consists of polling queues in increasing order of the corresponding priority (with the convention that a high priority task gets a low number). In our illustration, CPU cores would thus poll queues $(0,1)$, $(1,2)$ and $(2,0)$ in that order whereas GPUs would thus poll queues $(2,0)$, $(0,1)$ and $(1,2)$ in that order. If a poll is successful, the task is finally attributed to the PU for being processed as shown in the rightmost part of Figure 6.

The priority assignment is determined to make the most of the heterogeneous platforms, tasks being attributed priorities based on their relative performance on each PU. Because P2P has a better speedup than M2L, it gets a higher priority (0) on GPU and a very low priority (7) on CPU. Subsequently, as long as there are P2P tasks to be processed, they are offloaded to GPUs. On the contrary, M2L tasks get processed on GPUs only if they cannot be fulfilled with P2P tasks. Table 3 lists the detailed priority assignment.

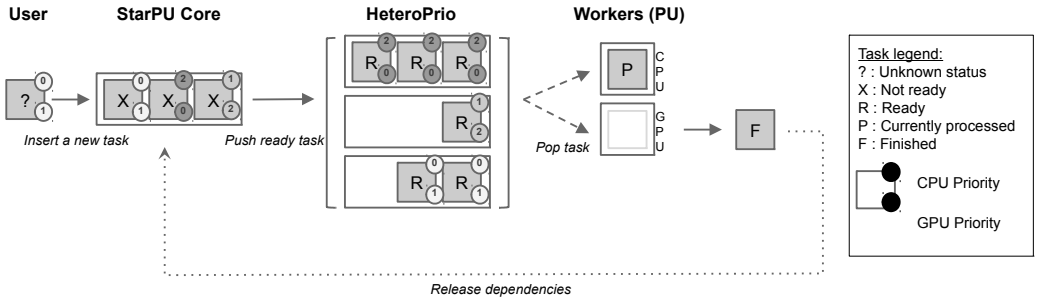


Figure 6: HeteroPrio scheduling heuristic.

Operators	Homogeneous CPU	Heterogeneous CPU	Heterogeneous GPU
P2M	0 (highest priority)	0	-
M2M	1	1	-
L2L*	2	2	-
P2P (Large)	3	7	0
M2L*	4	3	2
P2P (Small)	5	4	1
L2P	6	5	-
P2P-Reduction	7 (lowest priority)	6	-

Table 3: Tasks priority assignment of the HeteroPrio scheduling heuristic, in both the homogeneous and heterogeneous cases. * means that there is actually one priority per level in the octree.

Termination. HeteroPrio is intended to achieve a high steady-state throughput but is not protected against possible inefficient terminations. Indeed, when very few tasks remain to be processed, it might be worth postponing the execution of a task and letting a PU idle if that task can be processed much more efficiently on an other PU a few instants after. We thus use the following correcting heuristic to prevent HeteroPrio from taking such ineffective decisions. We note s the GPU speedup of a task over CPU. A PU can pop a task only if the number of tasks ($\#tasks$) is greater than that speedup ($\#tasks \geq s$). For example, assume that a CPU computes 1 task per second, whereas a GPU computes 30 tasks per second, corresponding to a speedup $s = 30$. Then, a CPU can pop that task only if there are more than $\#tasks = 30$ tasks available. To ensure a fine correction on any hardware setting, the number of accelerated PUs ($\#GPUs$) is actually also taken into account and a CPU can pop a task only if the relation $\#tasks \geq s \times \#GPUs$ is satisfied. If a PU cannot execute a task because of that exception, it simply polls another queue as if it had found the corresponding queue empty. In non uniform particle distributions, tasks may operate on very irregular amount of data. To limit the granularity of the tasks that may be processed during the termination phase, we split P2P tasks in two subset. The first subset corresponds to one half of the P2P tasks with larger granularity, whereas the second subset corresponds to the other half. P2P tasks operating on larger granularity get attributed a higher priority (priority 0 in Table 3) than the other half (priority 1 in Table 3).

4 Experiments

We now present an experimental study to assess our method. We first present the experimental environment in Section 4.1. In Section 4.2, we assess individually the GPU kernels proposed in Section 3.1. In Section 4.3, we then specifically study the ability of the HeteroPrio scheduling strategy proposed in Section 3.2 to achieve a high throughput and prevent sub-optimal termination. We finally propose an overall performance evaluation in Section 4.4.

4.1 Experimental environment

4.1.1 Test cases (particle distributions)

We consider two different example geometries with different types of particle distributions, respectively.

Cube (volume). Corresponds to random sampling particles in the unit cube as illustrated in Figure 7a. Such distributions have approximately the same number of particles per leaf cell and each cell has all its neighbors (except at the border of the cube).

Ellipsoid (surface). Corresponds to a sampling of particles on a the surface of an ellipsoid with a higher density at the extremities as illustrated in Figure 7b. This distribution leads to octree where the ratio of the leaf cell with the most particles versus the one with the least particles might be much larger than one. The aspect ratio of the ellipsoid is $1 : 5 : 1$.

In the sequel, the experiments are performed with $N = 30 \cdot 10^6$ particles both in the cube and ellipsoid cases, except in Section 4.3.2 Figure 13, where we consider an alternative ellipsoid composed of $N = 50 \cdot 10^6$ particles. Table 4 shows the details for those distributions, with a tree height $h = 7$ and $h = 11$, for the cube and ellipsoid test cases, respectively. We may notice important differences. First, in the cube test case, the number of particles per leaf is roughly constant contrary to the ellipsoid test case for which the density of particles strongly varies. Second, in the cube distribution, all the cells of the tree exist whereas in the ellipsoid distribution

the M2L interaction list is very sparse (and possibly may not exist); as a consequence, the M2L kernel performance is expected to be lower and much more sensitive to the task granularity n_g (as defined in Section 2.2). In a sparse octree, a group of n_g cells has a lower number of interactions to compute than a group of the same size from a dense configuration even though both groups take the same memory size. Thus, there is a direct relation between granularity, number of interactions and the filling of the octree. Finally, we also consider different levels of accuracy (as defined in Section 2.1). Typical low, medium and high accuracy correspond to $Acc = 3$, $Acc = 5$ and $Acc = 7$, respectively.

(a) Cube (volume) (b) Ellipsoid (surface)

Figure 7: Particles distributions of the test cases.

Property	Cube	Ellipsoid
Tree height	7	11
Number of leaves	262,144	755,815
Average particles per leaf	114.4	39.6
Minimum particles in a leaf	67	1
Maximum particles in a leaf	163	172
Average difference of particles per leaf	8.5	27.2
Average M2L per leaf-cell	176.6	46.5
Average difference of M2L per leaf-cell	20.4	4.9
Average sub-cells per cell	8	3.9

Table 4: Octree properties for the 30 million point cube (with a tree height $h = 7$) and ellipsoid (with a tree height $h = 11$) distributions.

4.1.2 Hardware configuration

We considered the two following heterogeneous platforms:

Nehalem-Fermi (M2070). This machine is a dual-socket hexa-core host machine based on Intel X5650 Nehalem processors operating at 2.67 GHz. Each socket has 12 MB of L3 cache and each core has 256 KB of L2 cache. The size of the main memory is 36 GB. It is enhanced with three Nvidia M2070 Fermi GPUs connected to the host with a 16x PCI bus. Each GPU has 448 cores and 6 GB memory with a bandwidth of 150 GB/s.

Nehalem-Fermi (M2090). This machine is based on the same CPU architecture and interconnection but is enhanced with three Nvidia M2090 Fermi GPUs which are composed of more cores (512) cores and benefit from an increased memory bandwidth (177 GB/s).

Both machines are thus composed of 12 CPU cores and 3 GPUs. However, because StarPU dedicates a CPU core to handle a GPU (see Section 2.3), they will be viewed as composed as 9 CPU cores enhanced with 3 pairs of CPU/GPU. In the sequel, the usage of such pairs will be implicit and we will simply refer to GPU usage when such a pair is used. We use the Intel compiler (version 12.0.5.220) with the compilation flags `-ip -O2`, the CUDA driver (version 4.2) and the Intel MKL library (version 10.2.7.041). All the experiments are performed in double precision.

4.1.3 Measure

The overall objective of this study is to show that a task-based approach is a valuable strategy for minimizing the FMM *completion time* on an heterogeneous platform. To compute the performance (or Flop rate) expressed in terms of floating point operations per second (Flop/s), we normalize that time with the number of operations performed. This is useful to appreciate the algorithmic effects. In an heterogeneous environment, the number of Flop required to perform certain mathematical operations (such as the evaluation of the square root of a real number) depends on the hardware unit on which it is performed. In our task-based model where the task mapping is decided dynamically, the actual number of Flop will therefore vary from one execution unit to another. In order to maintain a Flop rate inversely proportional to the elapsed time and independent of the task mapping, we computed the total number F of Flop performed in the computation when all the operations are performed on a CPU.[2] If we note t the measured elapsed time of a simulation, we thus report a performance $\mathcal{P} = F/t$ where F depends only on the numerical settings (particle distribution, accuracy and tree height) but not on the task mapping of the computational resources.

4.2 GPU kernel performance

We now present the performance of GPU kernels implementing the P2P and M2L tasks as defined in Section 3.1. The main interest of this result is not to compare our CPU and GPU kernels but rather present the acceleration factors since they are used to calibrate HeteroPrio.

We recall that we count P2P Flop assuming anti-symmetry is exploited, independently of the type of PU. The P2P CPU implementation is similar to [1] with a sequential performance of 2 GFlop/s. The P2P GPU kernel proposed in Section 3.1.1 achieves 66 GFlop/s on Fermi M2070 GPU (132 GFlop/s are actually reached because of redundancy but we do not count extra-flop).

As explained in Section 3.1.2, the M2L operator is composed of many fine grain matrix-vector products. Table 5 presents the Flop-to-word ratio, which is lower than 15 Flop/word for most of the configurations. We can compare this theoretical ratio to the GPU-specific Flop-to-word ratio capacity (i.e., ratio of the peak floating-point performance in Flop/s to the peak memory bandwidth of the device memory in words/s). On Fermi M2070 and M2090, the GPU-specific Flop-to-wordratios are 27.5 Flop/word and 30.1 Flop/word, respectively[3]. In Table 5 we present

[2] We followed http://folding.stanford.edu/home/faq/faq-flops/ for computing the cost of the basic floating point operations. For instance, the square root operator (sqrt) accounts for 15Flop.

[3] For the M2070 (resp. M2090), the peak double precision floating-point performance is 515 (resp. 665) GFlop/s and the peak memory bandwidth is 150 (resp. 177) GByte/s (ECC off). The Flop-to-word ratio is 36 for Tesla 2050 because the peak performance is 1030 GFlop/s and the peak bandwidth is 115 GB/s (ECC enabled). See `http://www.nvidia.com/object/tesla-servers.html`

the sequential Flop ratefor the M2L for different accuracies and different sizes of block. As expected from the low Flop-to-word ratio, the kernel has low performance compared to the GPU capacities. We can see that the number of cells and the accuracy have a significant impact on the GPU kernel performance.

Acc	Number of cells	CPU	M2070	M2090
3	512	1.63	3.19	4.18
	4,096	1.65	4.68	6.09
	32,768	1.68	5.26	6.83
5	512	3.82	13.8	17.7
	4,096	3.87	17.7	22.7
	32,768	3.87	19.2	24.8
7	512	5.17	22.4	29.2
	4,096	5.38	25.3	33.2
	32,768	5.46	26.4	34.6

Table 5: M2L kernel performance (GFlop/s) - assessed with the cube test case.

4.3 Scheduling efficiency

To achieve high performance, we furthermore need to schedule efficiently the execution of those kernels. In this section, based on trace analysis, we assess the quality of the HeteroPrio scheduling strategy proposed in Section 3.2. We recall that the objective is to overcome the twofold difficulty of achieving a high througput and preventing suboptimal termination due to load imbalance. The color legend defining the state of the tasks is given in Figure 8. P2P tasks are represented in white, M2L in light-gray and all other tasks (P2M, M2M, L2L and L2P) in gray. An idle computational unit is depicted in red except when it is actively waiting for a data in which case it is represented in purple.

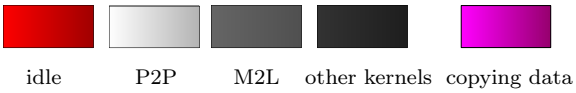


Figure 8: Color code used in Execution traces.

4.3.1 Refining the task flow for handling $P2P - Reduction$ kernel

As explained in Section 3.1.1, a new kernel $P2P - Reduction$ has been introduced to better cope with a machine using accelerators that have their own memory. Therefore the task flow presented in Figure 2 and initially proposed in [1] needs to be extended accordingly.

A natural extension consists of inserting the $P2P - Reduction$ tasks right after all the P2P tasks (and thus still before the L2P ones). We assess this strategy on the most simple heterogeneous balanced configuration where the computational load required to process P2P on the available GPUs is equivalent to the one required to process all other computations on CPUs. This configuration is matched on the *Nehalem-Fermi (M2070)* machine for a cube of $N = 30 \cdot 10^6$ particles processed with an octree of height $h = 7$ for achieving an accuracy $Acc = 5$. We observe in Figure 9 that even in this regular and well balanced load distribution, there is a period of

several seconds during which CPUs are almost not processing tasks. Indeed they are actively waiting (purple area) for the leaves to move from the GPUs memory to the host memory. The $P2P - Reduction$ tasks are not costly since they are the reduction of the duplicate border leaves. We can see in the middle of the copying data section that the $P2P - Reduction$ are performed (thin portions of black inside the purple). Finally, the L2P tasks are delayed because they depend on the $P2P - Reduction$ results.

Alternatively, we may insert the $P2P - Reduction$ tasks after the $L2P$ tasks. In this case, the L2P tasks can be run concurrently with the P2P tasks while the runtime is fetching back data resulting from GPU P2P executions to perform $P2P - Reduction$ tasks. Figure 10a shows that we obtain a very efficient termination in that case. The elapsed time is subsequently reduced from 13.9 s to 12.5 s.

4.3.2 Robustness of the scheduler with respect to the particle distribution

We now consider the exact same numerical configuration except that we use a lower accuracy ($Acc = 3$). The computational cost of the far-field being reduced, the near-field becomes dominant. Figure 11a shows the termination rule (see Section 3.2) allows HeteroPrio to detect that it can process the ultimate P2P tasks on CPU. In this particular case, it delegates no more than one task to a CPU core. As a consequence, the execution is not perfectly balanced: CPU cores are idle at the end of the execution. However, this is an optimum decision. Indeed, a P2P task lasts more than half of the overall computation when processed on a CPU core in this configuration. Processing two P2P tasks on the same CPU core would therefore necessarily increase the overall elapsed time of the simulation.

To further improve the load balancing, the only possibility consists of decreasing the task granularity. Figure 11a indeed shows that a granularity of $n_g = 1000$ (instead of $n_g = 2000$) allows to better balance the occupancy between CPU cores and GPUs, HeteroPrio having successfully adjusted its scheduling decision with the granularity. However, we may note that the overall time to completion is not improved ($t = 11.7$ s instead of $t = 11.4$ s) because the reduction of the granularity induces a slight decrease in the kernel performance that is not counterbalanced in this case by the gain obtained thanks to the better pipelining of the tasks.

Conversely, when the far-field is dominant (same configuration except that the accuracy is increased to $Acc = 7$), M2L tasks become predominant. Figure 12 shows that the steady-state rule (see Section 3.2) allows HeteroPrio to process in priority P2P tasks on GPU (their GPU priority has been set to maximum to cope with their higher acceleration factor) and that the termination rule finally delegates M2L tasks on GPU only when otherwise the GPUs would become idle. As a result, HeteroPrio takes advantage of task heterogeneity to better overcome with hardware heterogeneity, still maintaining a maximum occupancy of all computational units.

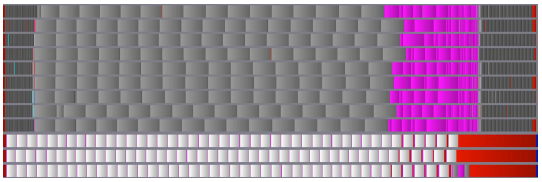
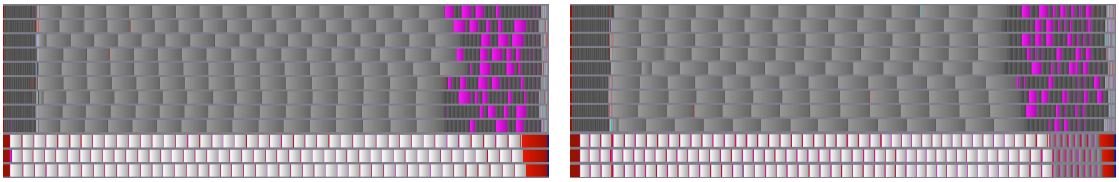
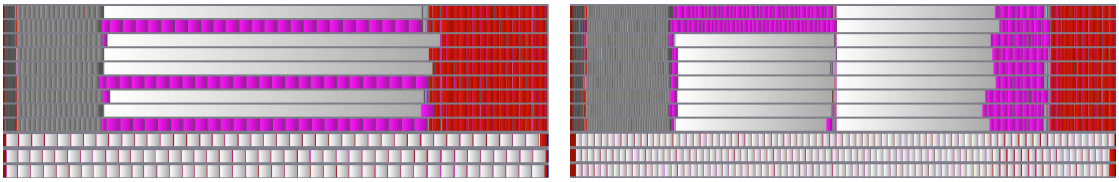


Figure 9: Execution trace when processing $P2P-Reduction$ right after P2P (and before L2P) on the *Nehalem-Fermi (M2070)* machine on a cube corresponding to a balanced execution between near field on GPU and farfield on CPU ($N = 30 \cdot 10^6$, $Acc = 5$, $h = 7$, $n_g = 2000$). The execution time is $t = 13.9\,\mathrm{s}$. The first nine lanes represent CPU cores occupancy and the three last ones GPUs.



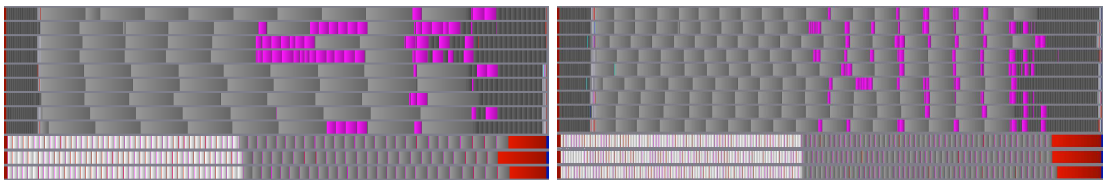
(a) *Nehalem-Fermi (M2070)* ($t = 12.5\,\mathrm{s}$) (b) *Nehalem-Fermi (M2090)* ($t = 10.9\,\mathrm{s}$)

Figure 10: Execution trace on both heterogeneous machines ($N = 30 \cdot 10^6$, $Acc = 5$, $h = 7$, $n_g = 2000$) with balanced near and far-field. Data from GPU are prefetched and $P2P-Reduction$ tasks are performed last. The nine first lanes represent CPU cores occupancy and the three last ones GPUs.



(a) $n_g = 2000$, $t = 11.4\,\mathrm{s}$ (b) $n_g = 1000$, $t = 11.7\,\mathrm{s}$

Figure 11: Execution trace on the *Nehalem-Fermi (M2070)* heterogeneous machine with a dominant near-field ($N = 30 \cdot 10^6$, $Acc = 3$, $h = 7$) for two different group sizes. The first nine lanes represent CPU cores occupancy and the three last ones GPUs.



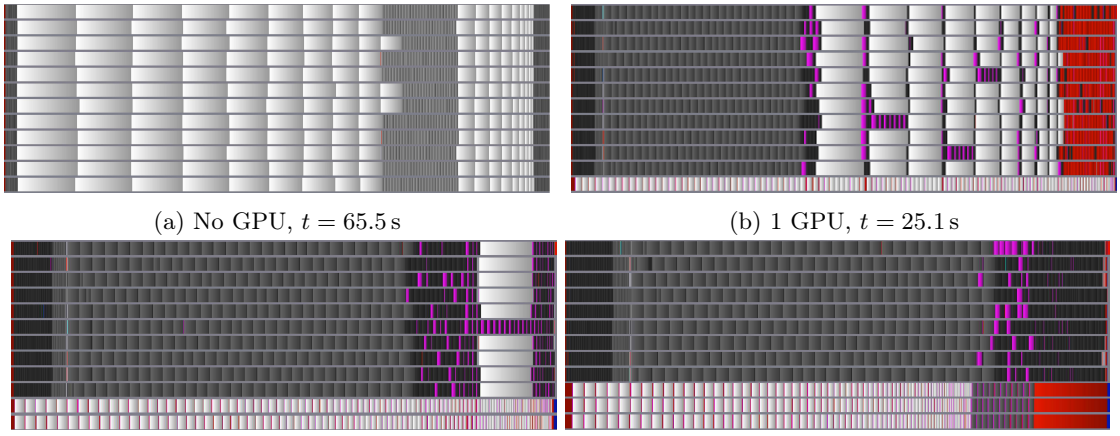
(a) $n_g = 2000$, $t = 27.6\,\mathrm{s}$ (b) $n_g = 1000$, $t = 27.9\,\mathrm{s}$

Figure 12: Execution trace on *Nehalem-Fermi (M2070)* with a dominant far-field ($N = 30 \cdot 10^6$, $Acc = 7$, $h = 7$). The first nine lanes represent CPU core occupancy and the three last ones the GPU core occupancy.

4.3.3 Robustness of the scheduler with respect to the hardware architecture

We now consider the exact same numerical setting ($N = 30 \cdot 10^6$, $Acc = 5$, $h = 7$, $n_g = 2000$) for which a perfect load balancing was achieved on *Nehalem-Fermi (M2070)* by processing the near-field on GPU and the far-field on CPU. If we now process this test case on the *Nehalem-Fermi (M2090)* platform, the extra power of the corresponding GPUs results in an acceleration of P2P computation. The HeteroPrio scheduling heuristic dynamically detects the imbalance and decides to process part of the M2L tasks on the GPU (right-most part of the GPU lanes in Figure 10b). This runtime decision allows the scheduler to maintain an almost perfect load balancing along the whole execution, showing the ability of our strategy to ensure performance portability across architectures.

We finally consider an ellipsoid distribution ($N = 50 \cdot 10^6$, $Acc = 5$, $h = 11$, $n_g = 3500$) and study the ability of HeteroPrio to cope with the number of employed GPUs. With such an irregular test case, two different P2P tasks may have a very disparate number of interactions and subsequent completion time. The granularity has thus been increased ($n_g = 3500$) so that in average tasks have enough work for achieving a decent Flop rate. The counterpart is that large P2P tasks might thus strongly unbalance the execution. However, the termination rule (see again Section 3.2) attributes the highest priority to such large P2P tasks on GPU. As a result, when the matter of balancing the load between the different PUs becomes critical at the end of the execution, only tasks of relatively small granularity (small P2P or far-field tasks) remain to be scheduled, strongly limiting the impact of potential imbalance on the overall completion time. Figure 13 shows that HeteroPrio achieves a very efficient load balancing both in the multicore case (Figure 13a) or in the accelerated cases (figures 13b, 13c and 13d). Furthermore, HeteroPrio again manages to consistently process as many P2P tasks as possible, only delegating M2L on GPUs when (Figure 13d) otherwise they would become idle, making the most of the heterogeneous platform.



(a) No GPU, $t = 65.5\,\mathrm{s}$ (b) 1 GPU, $t = 25.1\,\mathrm{s}$

(c) 2 GPUs, $t = 15.5\,\mathrm{s}$ (d) 3 GPUs, $t = 14.0\,\mathrm{s}$

Figure 13: Execution trace on *Nehalem-Fermi (M2070)* for an ellipsoid test case ($N = 50 \cdot 10^6$, $Acc = 5$, $h = 11$, $n_g = 3500$) using 0 to 3 GPUs. We also report the execution time t.

4.4 Overall performance evaluation

4.4.1 Method

We have proposed a consistent measure of the performance with respect to the architectural heterogeneity in Section 4.1.3. However, such a raw performance, even reported to the overall computational power of the machine, may not be a very tight estimation of the efficiency of the algorithms to harness the platforms we focus on in the present study. As a result, we propose to provide two tighter upper bounds together with raw performance numbers. The first upper bound (that we name LP hereafter) consists of the performance of the optimum schedule of the task graph based on the actual (measured) speed of the kernels for the considered numerical setting. This bound still depends on the chosen granularity parameter (n_g) and on the type of distribution (uniform or non uniform). We thus also propose an additional (looser) upper bound (that we name $\widetilde{LP}$ hereafter) that consists of the performance of the optimum schedule of the task graph if kernels were consistently running at their maximum speed (independently of the numerical setting). As a result, matching (or approaching) these bounds first mean that HeteroPrio successfully made the most of heterogeneity (steady-state rule in Section 3.2). Second, both bounds are computed assuming there are no dependencies between tasks (and thus in particular that communications are free). Indeed, FMM being very much parallel, a good schedule shall not be much penalized by dependencies and communications. Third, both bounds are computed assuming moldable tasks, which means that reaching the bounds furthermore implies that the termination rule allowed to handle efficiently possible load imbalance due to architectural heterogeneity (and to the presence of irregular tasks in the ellipsoid test case). We now present in details how these bounds are computed.

LP performance upper bound: Optimum schedule of the tasks at actual speed of the kernels. LP upper bound consists of the performance of the optimum schedule of the task graph based on the actual (measured) speed of the kernels for the considered numerical setting, assuming moldable tasks and no dependencies between them. Here is how we empirically construct the corresponding linear program.

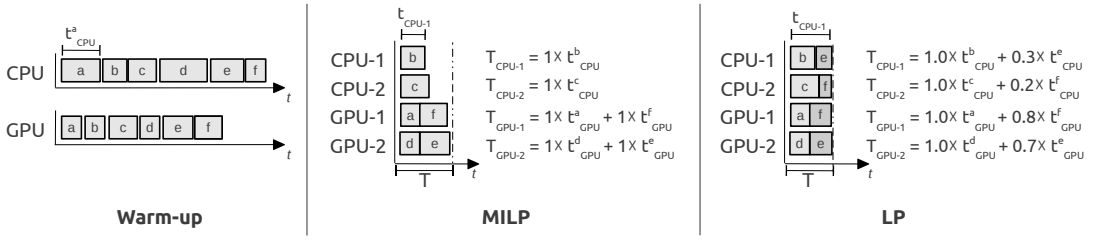


Figure 14: Empirical construction of LP upper bound for a set of tasks $\Omega = \{a, b, c, d, e, f\}$. In the sequel, $MILP$ is not considered and we only study LP.

For each type of PU (CPU and GPU here), we perform a warm-up stage where the execution time of all tasks is measured on the PU. The leftmost part of Figure 14 illustrates this warm-up stage assuming a set of tasks $\Omega = \{a, b, c, d, e, f\}$. The empirical time spent for a task ω are noted t^{ω}_{CPU} and t^{ω}_{GPU} on CPU and GPU, respectively. We note α^{ω}_{p} the proportion of workload of task ω processed on PU p. Because a task is in practice assigned to a unique PU p, α^{ω}_{p} should be equal to one on that unit and to zero on the other units. The subsequent objective function would aim at finding the schedule minimizing the elapsed time T such as in the center

part of Figure 14. However such a problem corresponds to Mixed Integer Linear Programming (MILP) and the bound for large task graphs could not be computed. Instead, we consider the corresponding (real case) Linear Program (LP) based on moldable tasks, *i.e.*, a single task ω could be processed on multiple PUs as it satisfies: $\sum_{p=1}^{P} \alpha_p^\omega = 1, \alpha_p^\omega \in \mathbb{R}^+$ where P represents the total number of PUs. This is why matching LP performance with an actual execution means that the overall workload was split into tasks of fine enough granularity in order to prevent imbalance at termination. The rightmost part of Figure 14 shows such a schedule where task e would be processed both on CPU-1 and GPU-2 and task f by both CPU-2 and GPU-1. We present the corresponding formal linear program in Figure 15, which we solve in practice with *lp_solve* [42].

$$\begin{cases} \text{Objective function: } \min(T) \\ \sum_{\omega\in\Omega} \alpha_1^\omega \, t_1^\omega = t_1 \leq T \\ \sum_{\omega\in\Omega} \alpha_2^\omega \, t_2^\omega = t_2 \leq T \\ \dots \\ \sum_{\omega\in\Omega} \alpha_P^\omega \, t_P^\omega = t_P \leq T \end{cases} \tag{8}$$

(a) Work assignment

$$\begin{cases} \sum_{p=1}^{P} \alpha_p^1 = 1 \\ \sum_{p=1}^{P} \alpha_p^2 = 1 \\ \dots \\ \sum_{p=1}^{P} \alpha_p^{|\Omega|} = 1 \end{cases} \tag{9}$$

(b) Constraints

Figure 15: LP linear program.

$\widetilde{LP}$ performance upper bound: Optimum schedule of the tasks at maximum speed of the kernels. $\widetilde{LP}$ upper bound consists of the performance of the optimum schedule if the kernels were consistently running at their maximum speed as reported in Section 4.2. It can be computed with the same linear program except that the estimated time $\widetilde{t_p^\omega}$ for processing task ω on PU p is defined as $\widetilde{t_p^\omega} = F(\omega)/\mathcal{P}$ where $F(\omega)$ is the number of Flop performed by task ω and $\mathcal{P}$ is the maximum performance of the kernel, which only depends on the type of task (and on the accuracy for M2L, see Table 5) but not on the type of distribution nor on the granularity.

4.4.2 Results

We consider the cube (Figure 16) and the ellipsoid (Figure 17) distributions composed of $N = 30 \cdot 10^6$ particles and executed on the *Nehalem-Fermi (M2070)* platform to illustrate our discussion. Depending on the accuracy (*Acc*, x-axis), the optimum tree height may vary from $h = 6$ (Figure 16a) to $h = 8$ (Figure 16c) and from $h = 10$ (Figure 17a) to $h = 12$ (Figure 17c) for the cube and the ellipsoid, respectively. We recall that increasing the tree height h leads to a decrement of work spent for the near field (P2P operator) but an increment for the far field (other operators including M2L). Furthermore, for a given tree height, requesting a higher accuracy *Acc* increases the cost of the far field evaluation (while the near-field cost is unchanged). Note that, for large tree heights ($h = 8$ and $h = 12$ for the cube and the ellipsoid, respectively), the number of cells may be prohibitive in terms of memory. Subsequently, the results could not be computed for large accuracies (which are memory demanding) in those cases (rightmost parts of figures 16c and 17c, respectively), especially when multiple GPUs are used (as buffers need to be allocated for them).

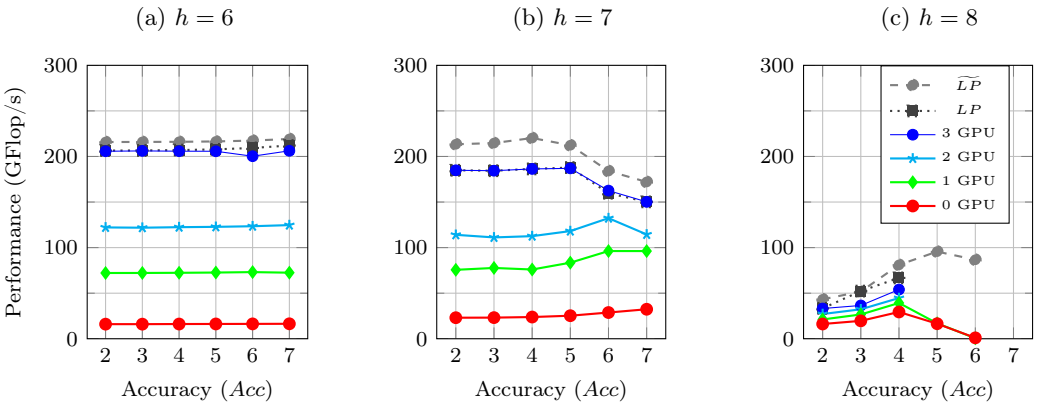


Figure 16: Performance on *Nehalem-Fermi (M2070)* for a cube test case ($N = 30{\cdot}10^6$, $n_g = 2000$).

We may first observe that the use of GPUs significantly or moderately increases the overall performance when the near-field or the far-field dominates, respectively. Indeed, we have shown in Section 4.2 that the P2P kernel is much more accelerated with a Fermi GPU than the M2L Chebyshev kernel. However, when all PUs are used (plain blue plots), the LP bound (dotted black plots) is almost consistently achieved. This means that HeteroPrio could effectively make the most of heterogeneity thanks to the steady-state rule proposed in Section 3.2. HeteroPrio could furthermore schedule the refined task graph (see Section 4.3.1) to efficiently overlap the far field with the near field. In the cube test case (Figure 16), it results in not only approaching the LP bound but almost consistently reaching it (figures 16a, and 16b as well as $Acc = 2$ case in Figure 16c). In the ellipsoid test case (Figure 17), the total workload is much smaller. As a result, dependencies and communications may not be fully hidden. Nevertheless, the overall performance remains close to the LP bound (ranging from 85% to 98%).

We may also observe the obtained performance is close to the $\widetilde{LP}$ bound in the cube distribution case, showing that an optimum performance could be achieved, without penalty due to granularity. This latter bound is looser in the ellipsoid case. Indeed, first, the maximum kernel speed (used to compute $\widetilde{LP}$) cannot be reached for a particle distribution on a surface. Second, because the workload of this relatively small workload compared to the computational power of the platform, the granularity ($n_g = 1800$) was traded off to deliver enough task concurrency at the cost of limited kernel performance.

For each considered accuracy and number of GPUs, the overall objective is to achieve the lowest time to completion. Thus, to conclude our study we present the corresponding timings in Figure 18. For instance, for the cube distribution at accuracy $Acc = 4$, a tree height $h = 7$ results in an execution of 79.7 s and 10.2 s in the CPU only case (0 GPU) and fully accelerated case (3 GPUs), respectively. On the other hand, a tree height $h = 8$ results in an execution of 42.2 s and 23.0 s in the 0 and 3 GPU cases, respectively. We have reported for that accuracy a time to completion of 42.2 s and 10.2 s in the 0 and 3 GPU cases, respectively ($Acc = 4$ in Figure 18a).

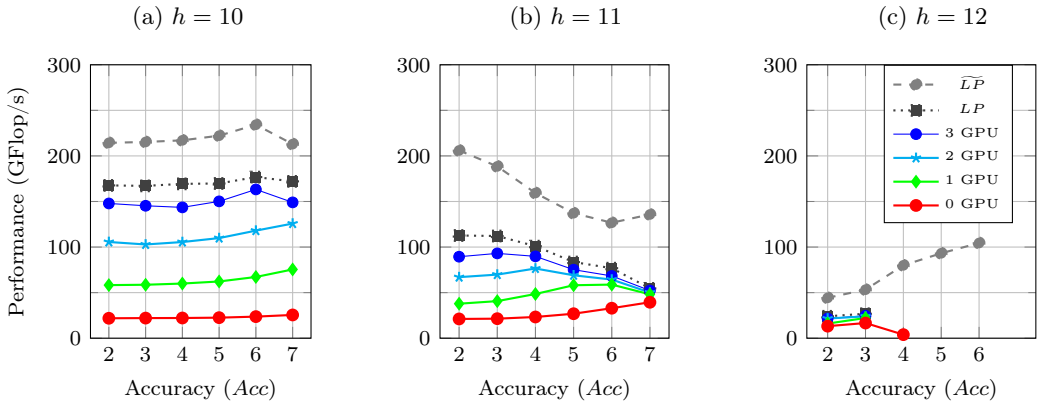


Figure 17: Performance on *Nehalem-Fermi (M2070)* for an ellipsoid test case ($N = 30 \cdot 10^6$, $n_g = 1800$).

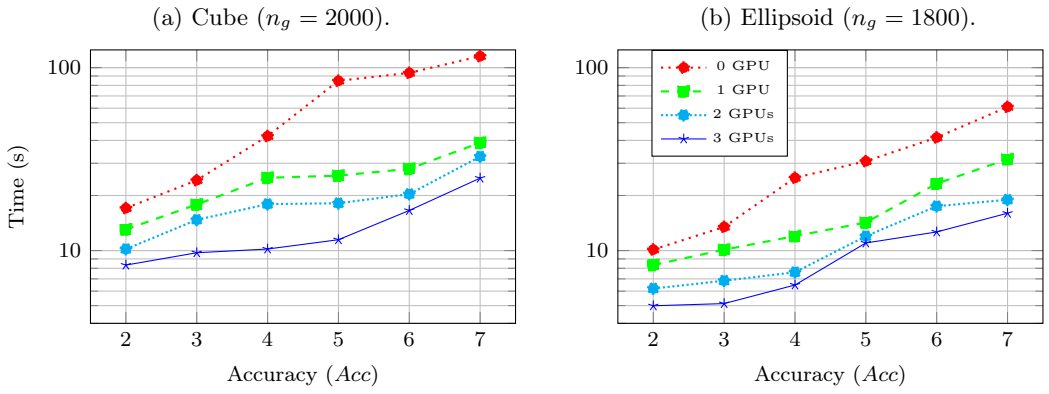


Figure 18: Time to completion (log scale) for distributions of $N = 30 \cdot 10^6$ particles. For a given accuracy and number of GPUs, the tree height h minimizing the time to completion was selected.

5 Conclusion

We have designed a task-based FMM for heterogeneous architectures. Because most of the work load is shared by P2P and M2L tasks, we have implemented a GPU version for both these kernels, the P2P kernel being derived from [11] and the M2L kernel being written from scratch. In most FMM codes, the work distribution among the processing units is static. In the present study, we have shown that an optimal performance can be achieved by moving on demand P2P tasks to the GPUs when the near-field is dominant or balanced with the far-field. Furthermore, when the far-field is larger with respect to the near-field, we have demonstrated the benefits of executing part of the M2L tasks on the GPUs too. Combined with our HeteroPrio dynamic scheduling strategy (Section 3.2), we have shown that we can exploit this flexibility to consistently achieve a near-optimal performance. We have indeed carefully assessed the impact on performance of four separate effects: kernel performance, task granularity, scheduling and octree settings. All in all, based on the task flow proposed in [1] (and refined in Section 4.3.1), the successive optimizations proposed in the present paper to fully exploit heterogeneity result in a single source code which consistently achieve high-performance with respect to both particle distribution and hardware configuration.

As discussed in Section 2.2, the degree of parallelism is governed by the granularity parameter n_g. In this study, we have considered a 30 million particle distribution (except in Figure 13 where we study a 50 million particle distribution) and demonstrated that they could be efficiently processed on a multicore processor enhanced with three Fermi GPUs. The parameter n_g was large enough to allow near-optimal kernel performance, especially for the cube particle distribution. Processing smaller test cases with the same computing power requires reducing the task granularity and eventually leads to a degradation of the kernel performance. CUDA streams, which allow multiple kernels to be executed concurrently on the same GPU device, could be employed to reduce task granularity without penalty of the efficient use of GPUs. The amount of data transfers between the main memory and the GPU memories are also likely to become prohibitive in such a situation. Providing a GPU implementation of all six[4] kernels would enable the scheduler to perform successive tasks on the same GPU and reduce this volume. The steady state rule could also be improved with a deeper static preliminary analysis such as [43]. Combining static pre-processing with dynamic corrections is a promising avenue for future work. Although assessed in the specific context of non directional, non adaptive, anti-symmetric Chebyshev FMM, the method proposed in this study may be applied to any FMM algorithm. In particular, we plan to extend this work to the adaptive FMM [44], as well as to the formulation for oscillatory kernels proposed in [37]. We also plan to apply this approach to clusters of heterogeneous nodes. Assuming a given data mapping, one approach would consist in performing explicitly inter-node communications. Alternatively, we could let the runtime system instantiate these communications [45] based on the task flow and the data mapping.

Acknowledgment

The authors would like to thank Raymond Namyst and Samuel Thibault from the Inria Runtime project for their advice on performance optimization with StarPU as well as A. Etcheverry and L. Giraud for reviewing a preliminary version of this manuscript. Experiments presented in this paper were carried out using the PLAFRIM experimental testbed, being developed under the Inria PlaFRIM development action with support from LABRI and IMB and other entities: Conseil Régional d'Aquitaine, FeDER, Université de Bordeaux and CNRS (see

[4] seven if we include the P2Preduce operator.

https://plafrim.bordeaux.inria.fr/).

References

[1] Emmanuel Agullo, Bérenger Bramas, Olivier Coulaud, Eric Darve, Matthias Messner, and Toru Takahashi. Task-Based FMM for Multicore Architectures. *SIAM Journal on Scientific Computing*, 36(1):66–93, 2014.

[2] Leslie Greengard and Vladimir Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73(2):325 – 348, December 1987.

[3] Leslie Greengard and Vladimir Rokhlin. A new version of the Fast Multipole Method for the Laplace equation in three dimensions. *Acta Numerica*, 6:229–269, 1997.

[4] L. Greengard and W. D. Gropp. A Parallel Version of the Fast Multipole Method. *Computers & Mathematics with Applications*, 20(7):63 – 71, 1990.

[5] Aparna Chandramowlishwaran, Samuel Williams, Leonid Oliker, and George Biros Ilya Lashuk, and Richard Vuduc. Optimizing and Tuning the Fast Multipole Method for state-of-the-art Multicore Architectures. In *Proceedings of the 2010 IEEE conference of IPDPS*, pages 1–15, 2010.

[6] Felipe A. Cruz, Matthew G. Knepley, and L. A. Barba. PetFMM—A dynamically load-balancing parallel fast multipole library. *Int. J. Numer. Meth. Engng.*, 85(4):403–428, 2011.

[7] Eric Darve, Cris Cecka, and Toru Takahashi. The fast multipole method on parallel clusters, multicore processors, and graphics processing units. *Comptes Rendus Mécanique*, 339(2-3):185–193, 2011.

[8] R. Yokota, T. Narumi, R. Sakamaki, S. Kameoka, S. Obi, and K. Yasuoka. Fast multipole methods on a cluster of GPUs for the meshless simulation of turbulence. *Computer Physics Communications*, 180(11):2066 – 2078, 2009.

[9] Nail A. Gumerov and Ramani Duraiswami. Fast multipole methods on graphics processors. *Journal of Computational Physics*, 227(18):8290 – 8313, 2008.

[10] Tsuyoshi Hamada, Tetsu Narumi, Rio Yokota, Kenji Yasuoka, Keigo Nitadori, and Makoto Taiji. *42 TFlops hierarchical N-body simulations on GPUs with applications in both astrophysics and turbulence*, pages 62:1—62:12. ACM, 2009.

[11] Toru Takahashi, Cris Cecka, William Fong, and Eric Darve. Optimizing the multipole-to-local operator in the fast multipole method for graphical processing units. *International Journal for Numerical Methods in Engineering*, 89(1):105–133, 2012.

[12] Qi Hu, Nail A. Gumerov, and Ramani Duraiswami. Scalable Fast Multipole Methods on Distributed Heterogeneous Architectures. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 36:1–36:12, New York, NY, USA, 2011. ACM.

[13] Ilya Lashuk, Chandramowlishwaran Aparna, Harper Langston, Tuan-Anh Nguyen, Rahul Sampath, Aashay Shringarpure, Richard Vuduc, lexing Ying, Denis Zorin, and George Biros. A massively parallel adaptive fast multipole method on heterogeneous architectures. In *Proceedings of the 2009 ACM/IEEE conference on Supercomputing*, page 1–11, 2009.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[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] 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.

[21] Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180:012037, July 2009.

[22] 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.

[23] Xavier Lacoste, Mathieu Faverge, Pierre Ramet, Samuel Thibault, and George Bosilca. Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes. In *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing Workshops and Phd Forum (IPDPSW'14), HCW 2014*, Phoenix, United-States, 2014.

[24] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, and Florent Lopez. Multifrontal qr factorization for multicore architectures over runtime systems. In Felix Wolf, Bernd Mohr, and Dieter Mey, editors, *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 521–532. Springer Berlin Heidelberg, 2013.

[25] Emmanuel Agullo, Luc Giraud, Abdou Guermouche, Stojce Nakov, and Jean Roman. Task-based Conjugate-Gradient for multi-GPUs platforms. Rapport de recherche RR-8192, INRIA, 2012.

[26] B. Lize, G. Sylvand, E. Agullo, and S. Thibault. A task-based H-matrix solver for acoustic and electromagnetic problems on multicore architectures. In *SciCADE, the International Conference on Scientific Computation and Differential Equations*, Valladolid, Spain, 2013.

[27] Hatem Ltaief and Rio Yokota. Data-driven execution of fast multipole methods. *CoRR*, arXiv:1203.0889, 2012. http://arxiv.org/abs/1203.0889.

[28] R. Kriemann. *H*-LU Factorization on Many-Core Systems. Preprint 5, Max-Planck-Institut für Mathematik in den Naturwissenschaften Leipzig, 2014. http://www.mis.mpg.de/preprints/2014/preprint2014_5.pdf.

[29] A. Duran, J. M. Perez, R. M. Ayguadé, E. amd Badia, and J. Labarta. Extending the OpenMP tasking model to allow dependent tasks. In *OpenMP in a New Era of Parallelism, 4th International Workshop, IWOMP 2008*, West Lafayette, IN, May 12-14 2008. Lecture Notes in Computer Science 5004:111-122. DOI: 10.1007/978-3-540-79561-2_10.

[30] 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.

[31] George Bosilca, Aurélien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack Dongarra. PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability. *Computing in Science and Engineering*, 99:1, 2013.

[32] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sağnak Taşirlar. Concurrent collections. *Sci. Program.*, 18(3-4):203–217, August 2010.

[33] A. YarKhan, J. Kurzak, and J. Dongarra. QUARK users' guide: QUeueing And Runtime for Kernels. Technical Report ICL-UT-11-02, Innovative Computing Laboratory, University of Tennessee, April 2011. `http://icl.cs.utk.edu/projectsfiles/plasma/pubs/56-quark_users_guide.pdf`.

[34] E. Chan, E. S. Quintana-Orti, G. Gregorio Quintana-Orti, and R. van de Geijn. Supermatrix Out-of-Order Scheduling of Matrix Operations for SMP and Multi-Core Architectures. In *Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures SPAA'07*, pages 116–125, June 2007.

[35] T. Takahashi, C. Cecka, and E. Darve. Optimization of the parallel black-box fast multipole method on CUDA. In *Innovative Parallel Computing (InPar), 2012*, pages 1–14, May 2012.

[36] William Fong and Eric Darve. The black-box fast multipole method. *Journal of Computational Physics*, 228(23):8712 – 8725, 2009.

[37] Matthias Messner, Martin Schanz, and Eric Darve. Fast directional multilevel summation for oscillatory kernels based on Chebyshev interpolation. *Journal of Computational Physics*, 231(4):1175 – 1196, 2012.

[38] M. Messner, B. Bramas, O. Coulaud, and E. Darve. Optimized M2L Kernels for the Chebyshev Interpolation based Fast Multipole Method. *ArXiv e-prints*, October 2012.

[39] G.M. Morton. *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*. International Business Machines Company, 1966.

[40] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

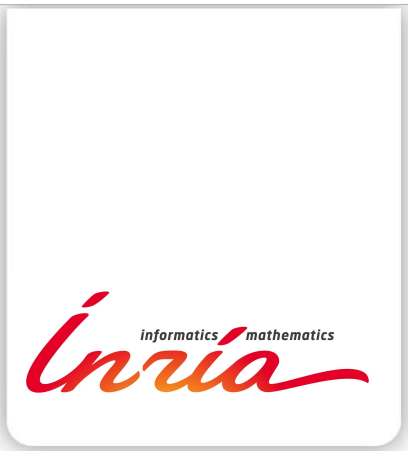
[41] Guangming Tan, Linchuan Li, Sean Triechle, Everett Phillips, Yungang Bao, and Ninghui Sun. Fast implementation of DGEMM on Fermi GPU. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 35. ACM, 2011.

[42] lp_solve mixed integer linear programming solver. `http://lpsolve.sourceforge.net/`.

[43] Jee Choi, Aparna Chandramowlishwaran, Kamesh Madduri, and Richard Vuduc. A cpu: Gpu hybrid implementation and model-driven scheduling of the fast multipole method. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, GPGPU-7, pages 64:64–64:71, New York, NY, USA, 2014. ACM.

[44] K Nabors, F T Korsmeyer, F T Leighton, and J White. Preconditioned, Adaptive, Multipole-Accelerated Iterative Methods for Three-Dimensional First-Kind Integral Equations of Potential Theory. *Journal on Scientific Computing*, 15(3):713–735, 1994.

[45] Cédric Augonnet, Olivier Aumage, Nathalie Furmento, Raymond Namyst, and Samuel Thibault. StarPU-MPI: Task Programming over Clusters of Machines Enhanced with Accelerators. In Siegfried Benkner Jesper Larsson Träff and Jack Dongarra, editors, *The 19th European MPI Users' Group Meeting (EuroMPI 2012)*, volume 7490 of *LNCS*, Vienna, Autriche, 2012. Springer.



RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

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

ISSN 0249-6399