Pipelining the Fast Multipole Method over a Runtime System

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

Pipelining the Fast Multipole Method over a Runtime System

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

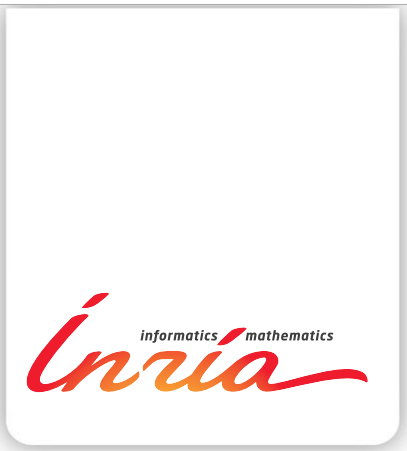
RESEARCH REPORT

N° 7981

May 2012

Project-Teams Hiepacs

ISSN 0249-6399 ISRN INRIA/RR--7981--FR+ENG



Pipelining the Fast Multipole Method over a Runtime System

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

Project-Teams Hiepacs

Research Report n° 7981 — May 2012 — 24 pages

Abstract: Fast Multipole Methods (FMM) are a fundamental operation for the simulation of many physical problems. The high performance design of such methods usually requires to carefully tune the algorithm for both the targeted physics and the hardware. In this paper, we propose a new approach that achieves high performance across architectures. Our method consists of expressing the FMM algorithm as a task flow and employing a state-of-the-art runtime system, StarPU, in order to process the tasks on the different processing units. We carefully design the task flow, the mathematical operators, their Central Processing Unit (CPU) and Graphics Processing Unit (GPU) implementations, as well as scheduling schemes. We compute potentials and forces of 200 million particles in 48.7 seconds on a homogeneous 160 cores SGI Altix UV 100 and of 38 million particles in 13.34 seconds on a heterogeneous 12 cores Intel Nehalem processor enhanced with 3 Nvidia M2090 Fermi GPUs.

Key-words: Fast multipole methods, graphics processing unit, heterogeneous architectures, runtime system, 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, 94305-3030 Stanford, CA, 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

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

Pipeline de la méthode multipôle rapide sur un moteur d'exécution

Résumé : Les méthodes multipoles rapides (FMM) sont une opération fondamentale pour la simulation de nombreux problèmes physiques. Leur mise en œuvre haute performance requiert habituellement d'optimiser attentivement l'algorithme à la fois pour la physique visée et la matériel utilisé. Dans ce papier, nous proposons une nouvelle approche qui atteint une performance élevée et portable. Note méthode consiste à exprimer l'algorithme FMM comme un flot de tâches et d'employer un moteur d'exécution, StarPU, afin de traiter les tâches sur les différentes unités d'exécution. Nous concevons précisément le flot de tâches, les opérateurs mathématiques, leur implémentations sur unité centrale de traitement (CPU) et processeur graphique (GPU) ainsi que les schémas d'ordonnancement. Nous calculons les potentiels et forces pour des problèmes de 200 millions de particules en 48.7 secondes sur une machine homogène SGI Altix UV 100 comportant 16O cœurs et de 38 millions de particules en 13.34 secondes sur une machine hétérogène composée d'un processeur Intel Nehalem accéléré avec 3 GPUs Nvidia Fermi M2090.

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

Contents

1 Introduction

Pair-wise particle interactions play an important role in many physical problems. Examples are astrophysical simulations, molecular dynamics, the boundary element method, radiosity in computer-graphics and dislocation dynamics. In the last decades numerous algorithms have been developed in order to reduce the quadratic complexity of a direct computation. The fast multipole method (FMM), first presented in [1], are probably the most prominent one. The fact that they have linear complexity makes them candidates of first choice for processing large-scale simulations of physical problems [2]. Thus, the design of efficient FMM implementations is crucial for the high performance computing (HPC) community. They have been ported to multicore processors [3, 4, 5, 6], graphical processing units (GPUs) clusters [7, 8, 9, 10, 11, 12]. Because these codes were tightly coupled with the targeted architectures, they could achieve excellent performance, sometimes beyond 10^6 floating point operations per second ($^{\mathrm{GFlop}}/_{\mathrm{s}}$). On the other hand, porting a code from one architecture to another was a commitment requiring very important (valuable) human resources. In this paper, we consider an alternative approach for achieving high performance across architectures. For that purpose, we turn the FMM algorithm into a task flow and employ a runtime system in order to dispatch the tasks on the actual computational units.

Processing HPC algorithms over runtime systems was successfully studied in the case of dense linear algebra algorithms during the past five years [13, 14, 15, 16, 17, 18, 19, 20] and is now a common utility for related state-of-the-art libraries such as Plasma [21], Magma [22] and Flame [23]. Dense linear algebra were excellent candidates for pioneering this path. First, the related task graph is very wide and therefore allows for many computational units to run concurrently. Second, the relative regularity of the tasks makes them particularly easy to schedule and achieve optimum performance [24]. On the contrary, pipelining the FMM over a runtime system is much more challenging. Indeed, the pattern of an FMM task flow is much more irregular and, hence, involving to handle. Moreover, the nature of the different computational steps and the subsequent granularity of tasks add complexity for ensuring an efficient scheduling. To tackle these challenges, we carefully analyze the FMM task flow such that it can be efficiently pipelined, we define fast mathematical operators and implement them on central processing units (CPU) and graphics processing units (GPUs), and we construct empirical performance models together with appropriate scheduling algorithms.

We propose an innovative methodology for designing HPC FMM codes. To our knowledge, only two other (very recent and not yet published) attempts have been made for processing the FMM over runtime systems. Ltaief and Yokota [25] have studied the feasibility of the approach on a 16 cores homogeneous machine. Bordage [26] obtained preliminary results for the Helmholtz kernel on heterogeneous architectures. In the present work, we not only show that the FMM can be highly pipelined, but also obtain performance numbers across architectures which are comparable to well established and heavily tuned methods for specific architectures [9, 12, 11, 6].

The paper is organized as follows. In Section 2, we briefly introduce the FMM algorithm in use and present an improved M2L kernel. In Section 3, we introduce the StarPU [27] runtime system and explain how to build a naive FMM task flow to be processed by the runtime system. In Section 4, we carefully design an improved task flow and present scheduling strategies for homogeneous architectures and assess their impact on multicore platforms. Finally, we tackle heterogeneous architectures in Section 5 before concluding in Section 7.

2 Fast Multipole Method

Pair-wise particle interactions can be modeled mathematically as

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

Here, pairs of particles, in the following denoted as sources and targets, are represented by their spatial position $x, y \in \mathbb{R}^3$, respectively. The interaction is governed by the kernel function $P(x, y)$. The above summation can also be understood as matrix-vector product $\mathsf{f} = \mathsf{Pw}$, where P is a dense matrix of size $M \times N$. Hence, if we assume $M \sim N$, the cost grows like $\mathcal{O}(N^2)$. The FMM reduces the cost of computing such summations to $\mathcal{O}(N)$. Its idea is to approximate the kernel function by a low-rank representation whose error decays exponentially if sources and targets are well separated (far-field). If sources and targets are not well separated (near-field) the kernel function will be used in its original form. These facts are exploited by hierarchically partitioning the computational domain into an octree (in $\mathbb{R}^3$) an subsequently identifying the near and the far-field. Finally, we end up with a data-sparse approximation of the dense matrix P.

2.1 The black-box FMM algorithm

Many FMM algorithms are kernel specific, meaning, they require a distinct analytic treatment for each kernel function. Our approach is adopted from [28] and can deal with broad range of kernel functions. Examples for the Laplace and the Stokes kernel and for various multiquadric basis functions have been presented. Another kernel independent FMM with a lower computational cost is presented in [29]. However, it is less general, since it works only for functions which are fundamental solutions of second-order constant coefficient non-oscillatory elliptic partial differential equations. Our approach has also been extended to the oscillatory Helmholtz kernel in [30].

In the paper at hand we present results for the Laplace kernel function

$$P(x, y) = \frac{1}{|x - y|} \quad \text{and} \quad F(x, y) = \frac{x - y}{|x - y|^3}. \tag{2}$$

Note, the second one can be written as $F(x, y) = \nabla_x P(x, y)$. And we use a Chebyshev interpolation scheme to interpolate these kernel functions as

$$\begin{aligned} P(x, y) &\sim \sum_{m=1}^{\ell} S_m(x) \sum_{n=1}^{\ell} P(\bar{x}_m, \bar{y}_n) S_n(y) \quad \text{and} \\ F(x, y) &\sim \sum_{m=1}^{\ell} \nabla_x S_m(x) \sum_{n=1}^{\ell} P(\bar{x}_m, \bar{y}_n) S_n(y). \end{aligned} \tag{3}$$

by shifting the gradient ∇_x from $F(x, y)$ to the interpolation polynomial $S_m(x)$. In the following the interpolation polynomial is referred to as P2M (particle-to-moment), M2M (moment-to-moment), L2L (local-to-local) and L2P (local-to-particle) operator and the point-wise evaluated kernel function as the M2L (moment-to-local) operator. Note, the P2M, M2M, M2L and L2L operators are the same for Equation (3), only the L2P operator differs.

In the remainder of this section, we present technical details on how we improved the black-box FMM. Note, they are not required for the understanding of the main scope of the paper and readers not familiar with the FMM may proceed to Section 3.

2.1.1 Efficient implementation of the P2M, M2M, L2L and L2P operators

In $\mathbb{R}^3$ we use the tensor-product ansatz for the translation operators

$$S_m(x) = S_{m,1}(x_1) \otimes S_{m,2}(x_2) \otimes S_{m,3}(x_3). \tag{4}$$

where each

$$S_{m,i}(x_i) = \frac{1}{\ell} + \frac{2}{\ell} \sum_{n=1}^{\ell-1} T_n(\bar{x}_{m,i}) T_n(x_i), \tag{5}$$

is a polynomial of order $\ell - 1$ and $T_n(x)$ with $x \in [-1, 1]$ are Chebyshev polynomials of first kind and $\bar{x}$ are interpolation points chosen to be Chebyshev roots.

The application of the M2P and P2L operator requires $\mathcal{O}(M\ell^4)$ floating point operations. However, by exploiting the outer-product-like representation from Equation (5), the cost can be reduced to $\mathcal{O}(M\ell^3 + \ell^4)$. Moreover, due to the tensor-product ansatz in Equation (4) the matrix-vector-product can be reformulated as a sequence of smaller matrix-matrix-products coupled with permutations (use of Blas3). In this way the cost for applying the M2M and L2L operator can be reduced from $\mathcal{O}(\ell^6)$ to $\mathcal{O}(\ell^4)$ only.

2.1.2 Improved M2L operator

Normally, in $\mathbb{R}^3$ the far-field is limited to the at most 27 near-field interactions of the parent-cell, only. This leads to at most 189 far-field interactions for one cell. Most kernel functions are homogeneous (if we scale the distance between source y and target x by a factor of α the resulting potential is scaled by α^n, where n is a constant and depends on the kernel function). Hence, the M2L kernels of all 316 possible far-field interactions for all cells need to be computed only once.

The authors of [28] proposed to compress all possible M2L operators P_t with $t = 1, \ldots, 316$, each of size $\ell^3 \times \ell^3$, via two big singular value decompositions (SVD). Subsequent algebraic transformations lead to an efficient representation as $\mathsf{P}_t \sim \mathsf{U}\mathsf{C}_t\mathsf{V}^\top$, where C_t is a matrix of size $r \times r$ (low-rank obtained via the SVD based on a prescribed accuracy ε_{SVD}). The matrices U and V, both of size $\ell^3 \times r$, are the same for all t, hence, their application can be shifted to the M2M and L2L operator, respectively.

We propose another approach. First, by exploiting symmetries, we can represent the 316 M2L operators by permutations of 16, only. Second, we apply an individual SVD to each of the 16 operators. In other words, instead of computing 316 matrix-vector products we compute 16 matrix-matrix-products coupled with permutations.

Table 1: Comparison of both M2L optimizations

Acc	r_{316}	weighted r_{16}	$\text{cost}_{16}/\ \text{cost}_{316}$
3	19	4.6	0.69
5	67	11.2	0.62
7	150	22.2	0.67

We compare both approaches in Tab. 1. The first column shows three different accuracies of the method given by *Acc*, i.e., $(\ell, \varepsilon_{\text{SVD}}) = (Acc, 10^{-Acc})$. Studies in [30] have shown that it can be useful to correlate ℓ and ε_{SVD}. In other words, by solely reducing ε_{SVD} or increasing ℓ, the method does not give more accurate results. The accuracies $Acc = 3, 5, 7$ lead to an relative error

of $\varepsilon_{L_2} \sim 10^{-5}, 10^{-7}, 10^{-9}$, respectively. The second column in Tab. 1 shows the obtained low-rank r_{316} by the first approach and the third column the weighted low-rank r_{16} of our approach. The overall cost is given by $\text{cost}_{316} = \mathcal{O}\left(316 \cdot r_{316}^2\right)$ and $\text{cost}_{16} = \mathcal{O}\left(316 \cdot 2\ell^3 r_{16}\right)$, respectively. Column four shows that our approach is favorable in terms of

- reduced precomputation cost and memory requirement: 16 small SVD, each of size $\ell^3 \times \ell^3$ instead of 2 big SVDs, both of size $316\,\ell^3 \times \ell^3$
- better compression (weighted $r_{16} \ll r_{316}$) reduces cost of applying the M2L operators
- better cache reuse: 16 matrix-matrix products instead of 316 matrix-vector products

3 FMM over a runtime system

A runtime system is a software component that aims at supporting the execution of an algorithm written in a relatively high-level language. Different runtime systems were designed to support accelerator-based platforms. StarSs [31] is an annotation-based language that can execute programs either on CPUs, GPUs or Cell processors, respectively, using SMPSs, GPUSs or CellSs. Initially designed for distributed memory machines, DAGuE [32] has been extended to handle GPUs too. The Harmony runtime system proposes features similar to StarPU [33]. Sequoia [34] statically maps hierarchical applications on top of clusters of hybrid machines. Charm++ can also support GPUs [35].

StarPU was originally designed for handling heterogeneous platforms such as multicore chips accelerated with GPUs. It is a natural candidate if one aims at achieving performance across architectures.

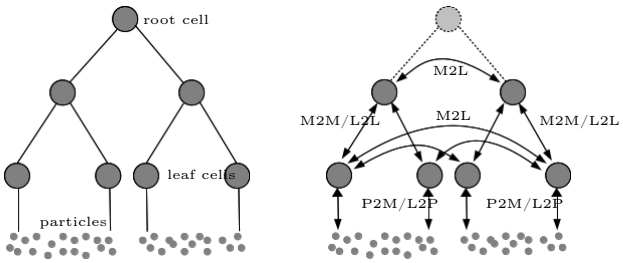
3.1 StarPU runtime system

StarPU was essentially designed to execute codes on heterogeneous platforms. The algorithm (FMM in our case) to be processed by StarPU is written in a high-level language, oblivious to the underneath platform. It is expressed as a so-called *task flow* consisting of a set of tasks with dependencies among them. Task flows can conveniently be represented as *directed acyclic graphs* (DAGs) where vertices represent individual tasks and edges dependencies among them. A multi-device implementation of the tasks, so-called *codelet*, is then provided by the programmer. It gathers the kernel implementations available to the different devices (CPU core and GPUs in our case). The data on which the tasks operate may need to be moved between the different computational units. StarPU ensures the coherency of these data. For that, data are registered to the runtime, which is accessing them not through their memory address anymore but through a StarPU abstraction, the *handle*, returned by registration. StarPU transparently guarantees that a task that needs to access a piece of data will be given a pointer to a valid data replicate. It will take care of the data movements and therefore relieve programmers from the burden of explicit data transfers. StarPU also detects the CPU core the closest to a GPU and selects it to handle that GPU.

In the next section, we show how the FMM, usually viewed as a tree algorithm, can be turned into a task flow suitable for execution over a runtime system.

3.2 FMM task flow

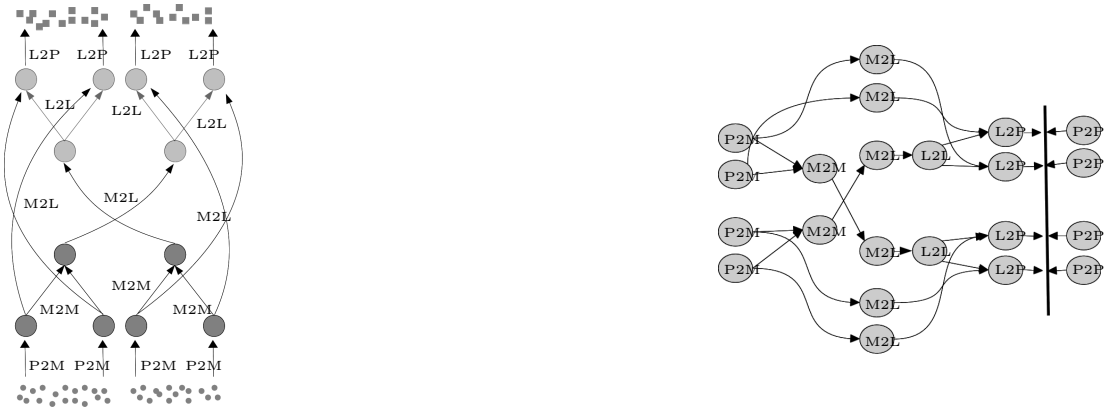
We start by subdividing the computational domain hierarchically into subdomains, hereafter, referred to as cells. The resulting data structure is a non-directed tree as presented in Figure 1a.



(a) Tree view (b) Cells and operations

Figure 1: Simplified FMM tree and subsequent operations.

We let vertices represent cells (which contain data such as multipole or local expansions) and edges represent the operations (P2P, P2M, L2L, M2M, L2P) applied on them (see Figure 1b). The tree is traversed twice. A bottom up traversal performs the P2M and M2M operations and a top down traversal performs the M2L followed by the L2L and L2P operations. P2P are usually associated to leaf cells and can be executed at any time except when the corresponding L2P is being executed. M2L can be represented with additional edges between siblings (see again Figure 1b). However, the resulting graph is neither a tree nor a DAG anymore since. The most straightforward way to derive a DAG from the tree would consist of mirroring the tree as illustrated in Figure 2a. The resulting DAG, where tasks are carried by edges, can finally be turned into an appropriate DAG shown in Figure 2b where tasks are carried by vertices (as required by the runtime system, see Section 3.1).



(a) Mirrored tree view (b) FMM task flow

Figure 2: Simplified FMM DAG.

3.3 Pipelining the FMM DAG

In a naive implementation of the FMM DAG each task works on one cell. For example a P2P tasks computes the at most 26 near-field interactions of one cell at the leaf level, a M2M task computes the equivalent source values based on the source values of its 8 child cells or an M2L tasks computes the at most 189 far-field interactions. This approach leads to

- a large number of tasks (there are at most 8^ν cells at level ν, hence, the same number of

tasks for all kernels)

- a large number of dependencies (a cell has at most 189 far-field interactions or a leaf cell has at most 26 near-field interactions and all of them must be made available to each task)
- a very small granularity (depends on *Acc*)

4 Homogeneous case

In the previous section, we have shown how the FMM can be turned into a task flow that can be executed by a runtime system. Here, we consider multicore platforms (Section 4.1) and design a variant of the task flow for which we tune the granularity of the tasks in order to efficiently exploit homogeneous multicore architectures. We show that very high parallel efficiency is achieved, which we explain thanks to a theoretical breakdown of the computational costs.

4.1 Experimental setup

We consider two homogeneous platforms for assessing our algorithms. The first platform is composed of four deca-core Intel Xeon E7-4870 processors running at 2.40 GHz (40 cores total). This machine is cache-coherent with Non Uniform Memory Access (ccNUMA), which means that each core can access the memory available on all sockets, with a higher latency if the accessed data is on another socket. Each socket has 256 MB of random access memory (RAM) and a 30 MB L3 cache. Each CPU core has its own L1 and L2 caches of size 32 KB and 256 KB, respectively.

The second machine is an SGI Altix UV 100. It is also a ccNUMA machine, composed of twenty octa-core Intel Xeon E7-8837 processors running at 2.67 GHz (160 cores total). Each socket has 32 GB of RAM and 24 MB of L3 cache. Each CPU core has its own L1 and L2 caches of size 32 KB and 256 KB, respectively.

In the rest of the paper, we will refer to these platforms as the *four deca-core Intel Xeon E7-4870* and *twenty octa-core Intel Xeon E7-8837* machines, respectively.

4.2 Pipelining strategies

The task flow proposed in Section 3 works on extremely fine grained data which induces a large amount of tasks. The overhead for the runtime system to handle all these tasks may become non negligible and produce a large penalty on the total execution time. Preliminary experiments (not reported here) showed that this approach does not scale. Assume, t_i is the time needed to schedule a task, t_e the time to execute a task and n_p the number of available threads. If $t_e < n_p t_i$ then not sufficient work for all threads will be available. We overcome this problem by increasing the granularity of the tasks by letting them operate on groups of cells at the same level as shown in Figure 3. On one hand, an increased task granularity leads to a higher performance of the

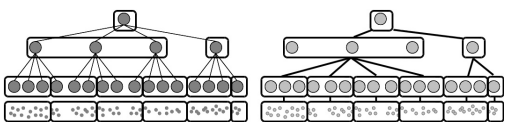


Figure 3: A tree split in groups of three elements

FMM kernels. On the other hand, fewer tasks are available and that limits the concurrency.

Thus, we parameterize the granularity which allows us to find a trade-off between performance and concurrency. The parameter is the size n_g of the group of cells we let a task work on. With this strategy we obtain relatively regular tasks even for very irregular problems. Moreover, it reduces the overall number of dependencies (see Fig. 14 for some studies). We use Morton ordering [36] to group cells. That is why they tend to be grouped also locally and, hence, they very likely share common near and far-field interactions.

4.3 Breakdown of the computational work

In Figure 4 we present the breakdown of the computational work for the different FMM kernels based on an example with $N = 20 \cdot 10^6$ uniformly distributed particles, an octree of height $h = 7$ and an accuracy $Acc = 7$. It shows the level-wise grouped far-field operators (P2M, M2M, M2L, L2L and L2P) and the respective number of required floating point operations. Edges indicate dependencies: The algorithm starts with the P2M at level 6 (the leaf level), once it is finished it gives the go to the M2L at the leaf level and the M2M at level 5, and so on. Evidently, the work of the M2L kernel is predominant. Moreover, most of the work is done by the kernels at the leaf level. Indeed, it decreases exponentially (we have a uniform octree; hence, by a factor of 8 per level).

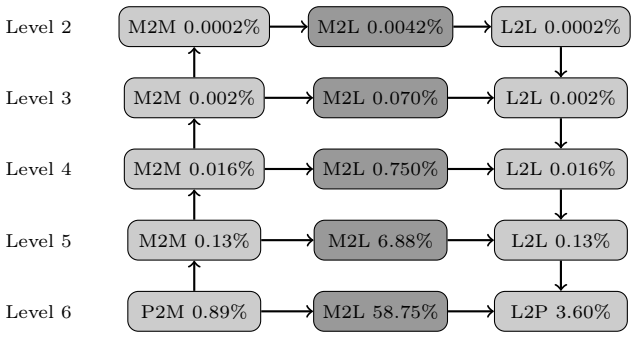


Figure 4: The breakdown of the computational work of the FMM kernels ($N = 20 \cdot 10^6$ uniformly in the unit-cube distributed particles, $h = 7$, $Acc = 7$) shows dependencies of the kernels at different levels and the percentage of the overall work which sums up to $2.71 \cdot 10^{12}$ floating point operations (P2P 28.75%)

4.4 Parallel efficiency

We now study the *parallel efficiency* of our approach, defined as

$$e_n = \frac{t_1}{nt_n},$$

where t_n denotes the measured execution time with n computational devices. In the Figures 5 and 6 we present studies obtained on the *four deca-core Intel Xeon E7-4870* and *twenty octa-core Intel Xeon E7-8837* machines, respectively. We present studies for uniformly and non-uniformly distributed particles and for different accuracies (defined as $(\ell, \varepsilon_{\mathrm{SVD}}) = (Acc, 10^{-Acc})$). All studies feature an extraordinary good scaling: between 80% and 98% efficiency at 40 CPUs in Figure 5 and between 60% and 86% efficiency at 160 CPUs in Figure 6. In order to understand the scaling behaviour more in detail we need to analyze the influence of the granularity of the tasks

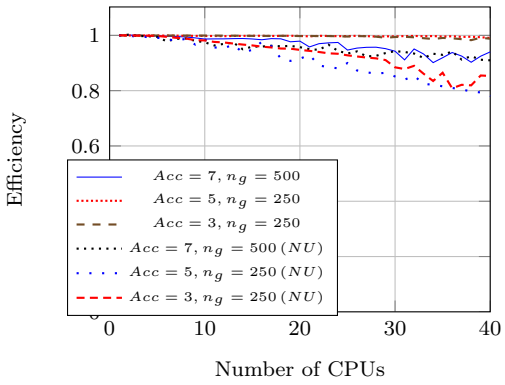


Figure 5: Parallel efficiency on up to 40 CPUs on the *four deca-core Intel Xeon E7-4870* machine for $20 \cdot 10^6$ particles (uniform distribution with $h = 7$ and non-uniform (NU) with $h = 8$); the granularity of tasks is parameterized by n_g, the number of cells it works on.

to be scheduled by the runtime system. In Section 4.2 we have parameterized the granularity of tasks by means of the number of cells n_g they work on (see Fig. 14 for some studies). Obviously, efficiency is related to concurrency, which can be increased by adding another level to the tree or by reducing the granularity of the tasks. In the uniform case in Figure 5 the high accuracy example is less efficient than the other two. The reason is that it does not provide enough concurrency due to a larger granularity ($n_g = 500$ compared to 250). In the non-uniform case the high accuracy example is the most efficient one. We have added one more level to the tree, hence, there is enough concurrency even for tasks of larger granularity. In Figure 6 we kept the

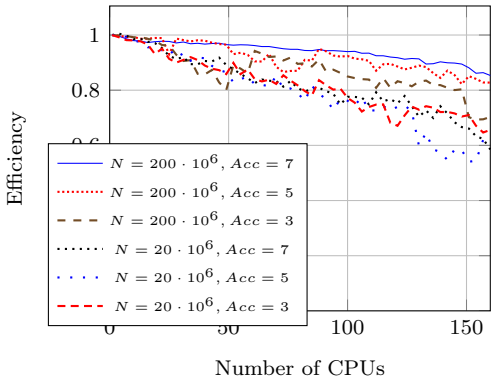


Figure 6: Parallel efficiency on up to 160 CPUs on the *twenty octa-core Intel Xeon E7-8837* machine for uniformly distributed particles (for $N = 20 \cdot 10^6$ we use $n_g = 500$ and $h = 7$, for $N = 200 \cdot 10^6$ we use $n_g = 1000$ and $h = 8$); the granularity of tasks is parameterized by n_g, the number of cells it works on.

granularity constant. The concurrency does not change but the granularity changes depending on the chosen accuracy (higher accuracy leads to larger granularity). All studies behave as expected, those with larger granularity are more efficient.

Figure 8 shows the execution trace obtained by the runtime system StarPU. Each horizontal lane shows the occupancy of a particular processing unit (here 40 CPUs) as a function of the

time. As illustrated in Figure 7, red (dark) sections denote idle time, white sections denote the

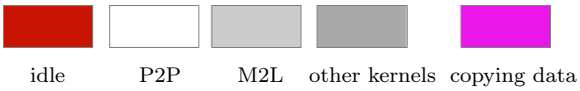


Figure 7: Color legend for traces

P2P kernel, light gray sections denote the M2L kernel and medium gray sections denote all other kernels (P2M, M2M, L2L, L2P; as can be seen in Figure 4 their work share is vanishingly small compared to P2P and M2L). The purple color denotes that a tasks is available but not the data. The trace shows a highly pipelined execution. Depending on the execution, a barrier can be observed before the L2P. This is due to data dependencies in the L2L meanly at leaves level.

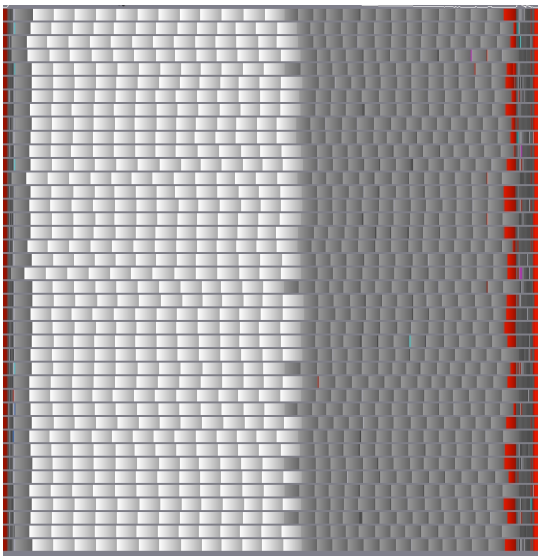


Figure 8: Trace for the study in Figure 5 with $N = 20 \cdot 10^6$ and $Acc = 7$ (execution time $t = 23\,\mathrm{s}$).

5 Heterogeneous case

If GPUs are now available on the computational node, the execution of a task can be deported on them by the runtime system (see Section 3.1). As discussed above, most FMM codes for heterogeneous machines decide how to distribute the work load before starting the actual parallel execution. With such a static approach, it is very hard to consistently achieve efficient load balancing. StarPU allows for a dynamic distribution of the work load based on given scheduling strategies (see [37]). We use the *Eager* scheduler for all computations. Other schedulers are available on StarPU, but they are not yet considered as stable. Some tests have been performed with the *Heft* scheduler which is able to estimate and use the duration of tasks to schedule them optimally. Such estimation is based on preliminary runs where the duration of tasks is stored in a database. In Fig. 9 calibration results for the *Heft* scheduler of the P2P kernel are presented. During two preliminary runs the duration of all P2P tasks as a function of the number of interactions are stored in a database and based thereon the scheduler is able to estimate the duration of tasks in future runs.

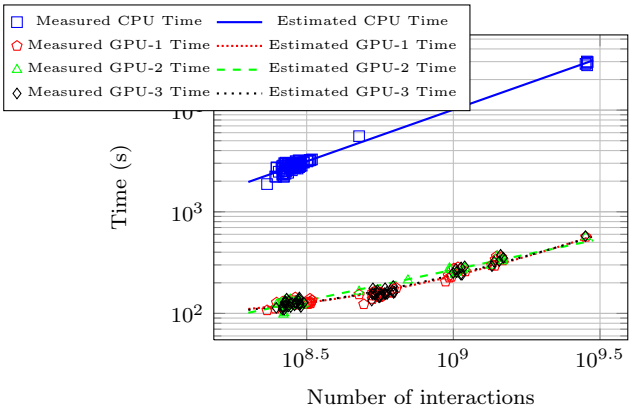


Figure 9: Calibration results of the *Heft* scheduler after two simulations for the CPU and the three GPU.

In Section 5.2, we consider executions where near-field (P2P) and far-field (other kernels) interactions are well balanced. Relying on a high performance blocked P2P kernel, we show that processing P2P (and only P2P) on GPUs allows for achieving well balanced executions. We then consider the case where the near-field is dominant in Section 5.3 and we show that we continue to achieve a good load balance by scheduling P2P tasks on demand on CPU or GPU. Finally, we study the case where the far-field is dominant in Section 5.4. By relying on a M2L GPU kernel (not yet optimized), we manage to maintain a high pipeline and well balanced execution.

5.1 Experimental setup

We consider 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 48 GB. Moreover, we have two different GPU sets associated to it. They consist of three Nvidia M2070, respectively, M2090 Fermi accelerators and are connected to the host with a 16x PCI bus. Each GPU has 6 GB of GDDR-5 of memory and 14 processors (448, respectively 512 cores), operating at 1.15 GHz. In the rest of the paper, we will refer to this platforms as the *Nehalem-Fermi (M2070)*, respectively, *Nehalem-Fermi (M2090)* machine. If we do not need to explicitely refer to one of these clusters we simply call them heterogeneous machines since they are composed of CPUs and GPUs. Because StarPU dedicates a CPU core in order to handle a GPU (see Section 3.1), both machine will be viewed as a nine CPU cores node enhanced with three GPUs.

5.2 Balancing near and far-field

We consider executions where near-field (P2P) and far-field (other kernels) computational costs are well balanced. Relying on a high performance blocked P2P kernel, we design a first scheme that processes all P2P (and only P2P) on GPUs and performs highly pipelined executions.

We have derived from [10] a GPU kernel that operates on groups of n_g cells. Figure 9 compares the execution times of our CPU and GPU P2P kernels. Because we group multiple cells to be processed into a single kernel call, not all particles interact directly with each other. The time spent for executing this kernel is thus neither proportional to the number of cells nor

particles. Figure 9 indeed shows that the execution time of the kernel is proportional to the number of effective interactions (x-axis).

Our main goal is to obtain perfectly piplined executions. Hence, we choose the reference example settings such that the computational load is well balanced between the three available GPUs and nine available CPUs. Initially we process the near-field (P2P kernels) on the GPUs and the far-field (all other kernels) on the CPUs. In order to end up with a roughly matching execution time we choose $N = 38 \cdot 10^6$, $Acc = 5$ and $h = 7$. Moreover, we chose a large block size ($n_g = 1500$) in order to achieve a high flop-rate for the P2P GPU kernel ($102\,\mathrm{GFlop/s}$).

Figure 10 shows the execution trace of the task flow proposed in Section 4 on *Nehalem-Fermi (M2070)*, the heterogeneous machine presented in Section 5.1. We recall that for the moment we force all P2P tasks to be executed on GPUs. As expected, they are perfectly pipelined (last three lanes in Figure 10) since they are all independent one from another. However, CPU cores (first nine lanes) are not fully occupied (purple color means that a task is available but its data not). The reason is that P2P tasks, executed on GPUs, are followed by P2Preduce tasks which are executed on CPUs. What is the P2Preduce tasks for? It basically exploits the fact that if source and target particles are the same the resulting matrix becomes symmetric. Hence, we do not only write the resulting potential and forces on the target particles but also on source particles. The P2Preduce tasks perform this data reduction and cannot be executed at the same time as the corresponding L2P tasks.

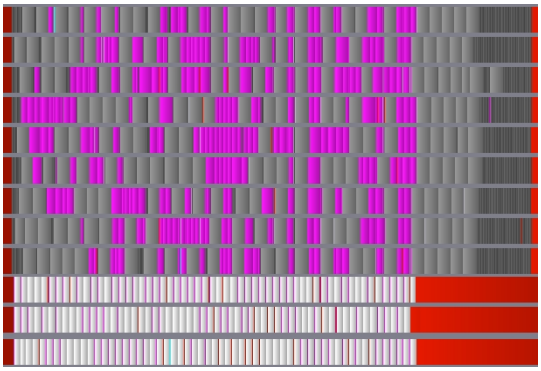
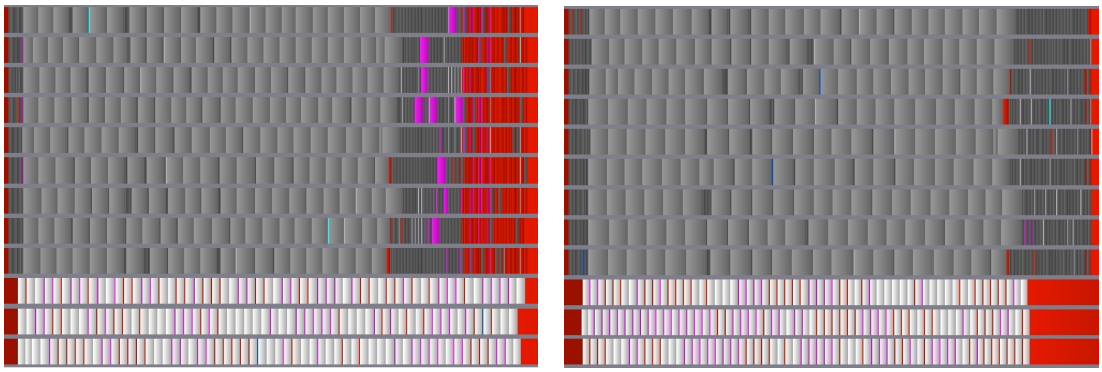


Figure 10: Execution trace on the *Nehalem-Fermi (M2070)* heterogeneous machine ($N = 38 \cdot 10^6$, $Acc = 5$, $h = 7$, $n_g = 1500$) with balanced near and far-field. Each P2P task is followed by the P2Preduce tasks. The execution time is $t = 18.50\,\mathrm{s}$. The nine first lanes represent CPU cores occupancy and the three last ones GPUs.

Finally, by forcing P2Preduce tasks to be processed after L2P tasks and prefetching all data movements between CPUs and GPUs, we obtain a perfect pipeline (see Figure 11). It reduces the walltime from $18.50\,\mathrm{s}$ to $14.78\,\mathrm{s}$.

5.3 Dominant near-field

Next, we consider an execution with a lower accuracy ($Acc = 3$). M2L computational cost is reduced (M2L is the most sensitive kernel to the accuracy) and therefore P2P strongly dominates. If P2P is fully processed on GPU, CPUs remain idle most of the time (trace not reported here). We therefore allow for scheduling dynamically P2P on CPU or GPU. In order ensure prefetching, the scheduling decision has to be taken ahead of time (so that data have time to be moved between the different computational units). However, because of the heterogeneity of the computational



(a) *Nehalem-Fermi (M2070)* ($t = 14.78\,\mathrm{s}$) (b) *Nehalem-Fermi (M2090)* ($t = 13.34\,\mathrm{s}$)

Figure 11: Execution trace on both heterogeneous machines ($N = 38 \cdot 10^6$, $Acc = 5$, $h = 7$, $n_g = 1500$) with balanced near and far-field. Data from GPU are prefetched and P2Preduce tasks are performed last. The nine first lanes represent CPU cores occupancy and the three last ones GPUs.

power of CPUs and GPUs, this may still lead to unbalanced executions (Figure 12. By limiting the anticipation of such decisions, we managed to balance correctly the load (trace not reported here).

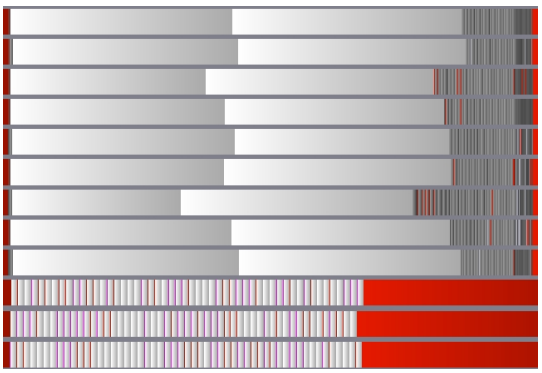
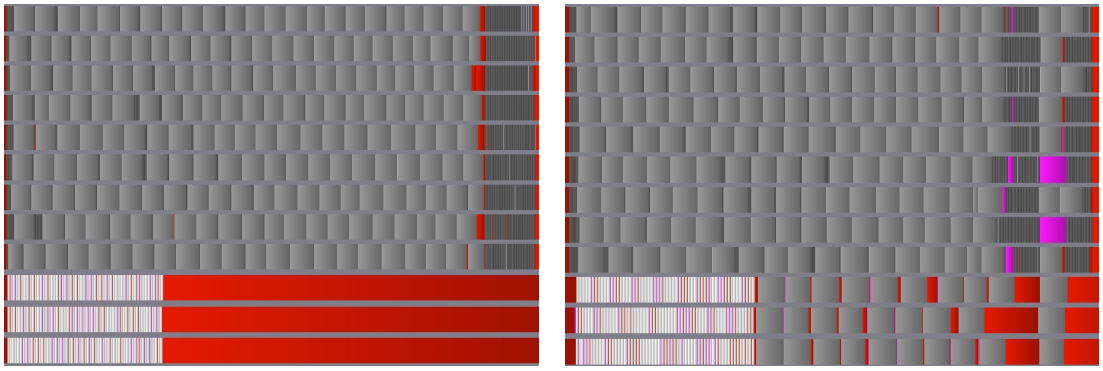


Figure 12: Execution trace on the *Nehalem-Fermi (M2070)* heterogeneous machine with a dominant near-field ($N = 38 \cdot 10^6$, $Acc = 3$, $h = 7$, $n_g = 1500$). The execution time is $t = 19.12\,\mathrm{s}$. The nine first lanes represent CPU cores occupancy and the three last ones GPUs.

5.4 Dominant far-field

We finally, by increasing the accuracy ($Acc = 7$), we consider the case where the far-field dominates. Since M2L tasks become dominant, the computation is unbalanced (Figure 13a). Thus, we have designed a (non optimized) GPU M2L kernel in order to show that we can efficiently balance the load by scheduling M2L tasks on demand (P2P tasks are processed on GPUs). Figure 13b shows the resulting trace. The pipeline is again very efficient and the load well balanced (but the performance of the M2L kernel would remain to be optimized).



(a) No M2L tasks on GPU ($t = 49.02\,\mathrm{s}$) (b) Some M2L tasks on GPU ($t = 42.09\,\mathrm{s}$)

Figure 13: Execution trace on *Nehalem-Fermi (M2070)* with a dominant far-field ($N = 38 \cdot 10^6$, $Acc = 7$, $h = 7$, $n_g = 1500$). The nine first lanes represent CPU core occupancy and the three last ones the GPU core occupancy.

6 Performance benchmarks

Several parameters influence the achieved flop-rates. One is the group-size n_g, but Fig. 14 shows that its influence is less critical. In other words, a broad range of n_g leads to the same good flop-rates. Interesting is the benchmark on the 20 octa-core Intel Xeon machine for $N = 38 \cdot 10^6$: it is best performing for $1000 < n_g < 3000$. This is a rather small interval compared to the other three benchmarks. The reason is that larger n_g does not provide enough parallelism anymore.

In Fig. 15 we show flop–rates studies for uniform (in the unit-cube) and in Fig. 16 for non-uniform (on the unit-sphere) particle distributions. The number of particles is in either case $N = 38 \cdot 10^6$. We vary the octree height $h = 6, 7, 8$. By doing so we can redeploy work between near- and far-field. A smaller h means more near- and less far-field and a larger h the other way around. Moreover, we vary the accuracies $Acc = 2, \ldots, 7$. By doing so, the cost for evaluating the near-field does not change but higher Acc increases the cost for evaluating the far-field. All, these effects can obviously be recognized in the figures 15 and 16.

In the figures 15a, 16a and 16b the computations are clearly dominated by the near-field, i.e., the P2P kernels. Except in the 0 GPU case they are all performed on the GPU. In either case we observe the linear scaling for additional GPUs. In the figures 15b and 16c we have a larger h and after $Acc = 5$, respectively, $Acc = 3$ the computations are dominated by the far-field, i.e., mainly the M2L kernels. They are scheduled by StarPU dynamically to available CPUs and GPUs. However, since our GPU implementation of the M2L kernel is not yet optimized, the flop-rate breaks down once the far-field becomes dominant. In Fig. 15c we have the uniform case with $h = 8$, which is clearly to large (about 8^7 leafs which contain only about 18 particles). The flop-rate is one order of magnitude smaller then for $h = 6, 7$. In Fig. 15c the examples for $Acc = 6, 7$ have not been computed.

What is the main difference between uniform and non-uniform particle distributions? In the uniform case particles are distributed in a tree-dimensional subset of $\mathbb{R}^3$, in the non-uniform case they are distributed in a two-dimensional subset of $\mathbb{R}^3$, only. This leads to an octree in the uniform case and to a quadtree in the non-uniform case. Since N is the same and the convex-hull is of the same order in both cases, at a given level of the tree a non-empty cluster in the non-uniform case tends to contain an order of magnitude more particles compared to the uniform

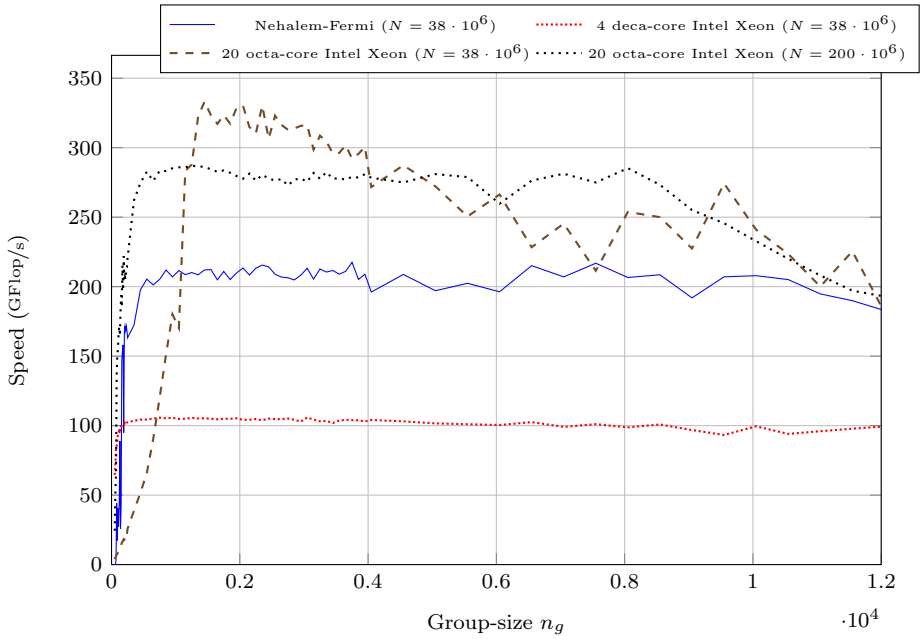


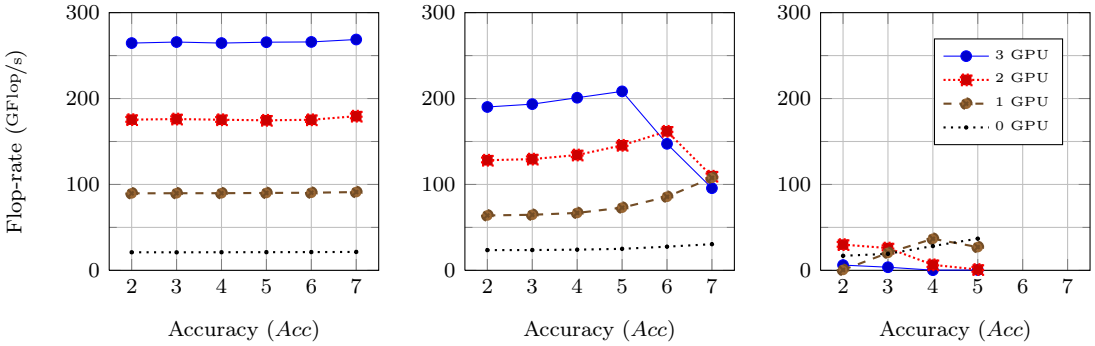
Figure 14: Flop-rates on three different computers for different group sizes $n_g < 12000$ (particles per group) for $Acc = 5$ and $h = 7$ (uniform particle distribution)

case. That is why if we compare Fig. 15a and Fig. 16b we obtain the same flop-rates even though h is different.

In Fig. 17 we compare absolute timing results corresponding to the flop-rates from Fig. 15 and Fig. 16. We omit results for the uniform case with $h = 8$. In Fig. 17a and 17b the times for the non-uniform case and $h = 6$ (dominated by near-field) are outside of the presentable range. All figures show that in the uniform case a octree height of $h = 7$ leads to the shortest computation times. In the non-uniform case for accuracies $Acc < 6$ an octree $h = 8$ otherwise $h = 7$ provides the best setting.

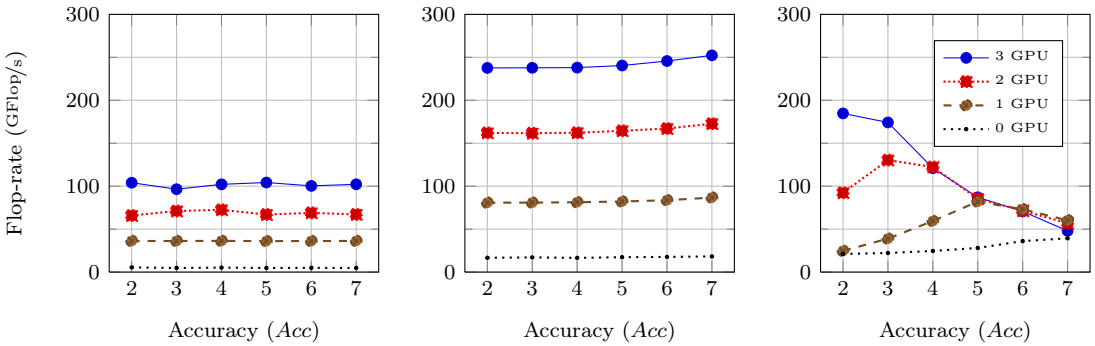
Figure 18 addresses the scaling behavior of our dynamically scheduled FMM implementation in the case of heterogeneous architectures. We take the results from Figure 17, choose the minimal timings for each h and plot them over Acc. In the uniform case (Figure 18a) they correspond to $h = 7$. In the non-uniform case (Figure 18b) and with 0 and 1 GPU they correspond to $h = 7$, too. However, with $2, 3$ GPUs and up an accuracy $Acc = 5, 6$ an octree height $h = 8$ and for higher accuracies $h = 7$ leads to minimal timings. As expected, we get excellent scaling for low accuracies ($Acc < 5$) where the near-field is dominating (we have an optimized GPU implementation of the P2P kernel). For example, let us look at $Acc = 3$: without GPU the computation takes 120.8 s, with $1, 2, 3$ GPUs it takes 44.1 s, 22.1 s and 14.7 s, respectively. The scaling for high accuracies can be improved by implementing optimized far-field kernels (P2M, M2M, M2L, L2L and L2P).

A comment on the 1 GPU case in Figure 18b: the timings for $Acc = 2, 3$ are not as expected. They should be the same as for $Acc = 4, 5$, because the near-field is dominating up to $Acc \leq 5$. In our understanding the reason for that is the fact that the *Eager* scheduler (see Section 5) has problems to schedule very small tasks correctly. However, by using the *Heft* scheduler we were able to reproduce the expected results.



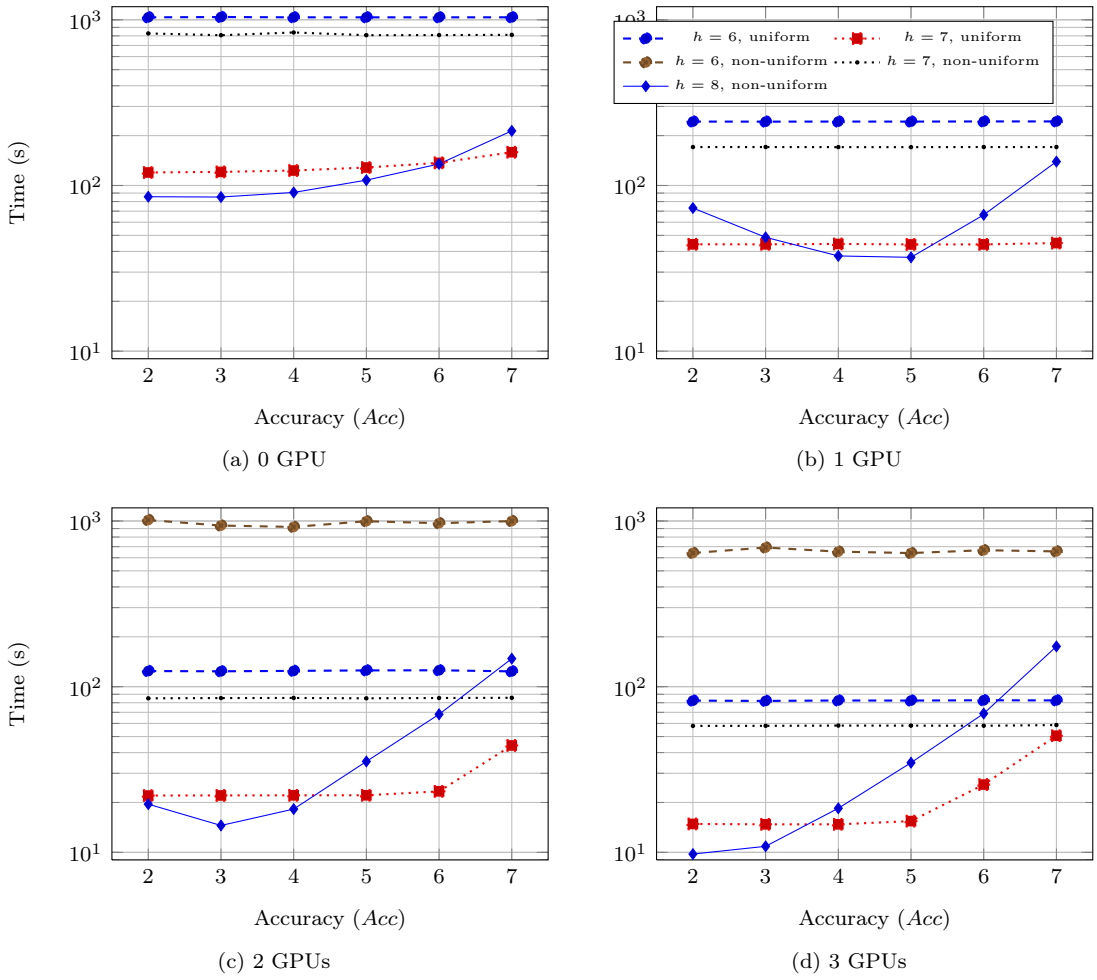
(a) $h = 6$ (b) $h = 7$ (c) $h = 8$

Figure 15: Flop-rates for uniform distribution (in the unit-cube) of $38 \cdot 10^6$ particles and group size $n_g = 1000$



(a) $h = 6$ (b) $h = 7$ (c) $h = 8$

Figure 16: Flop-rates for non-uniform distribution (on the unit-sphere) of $38 \cdot 10^6$ particles and group size $n_g = 1000$



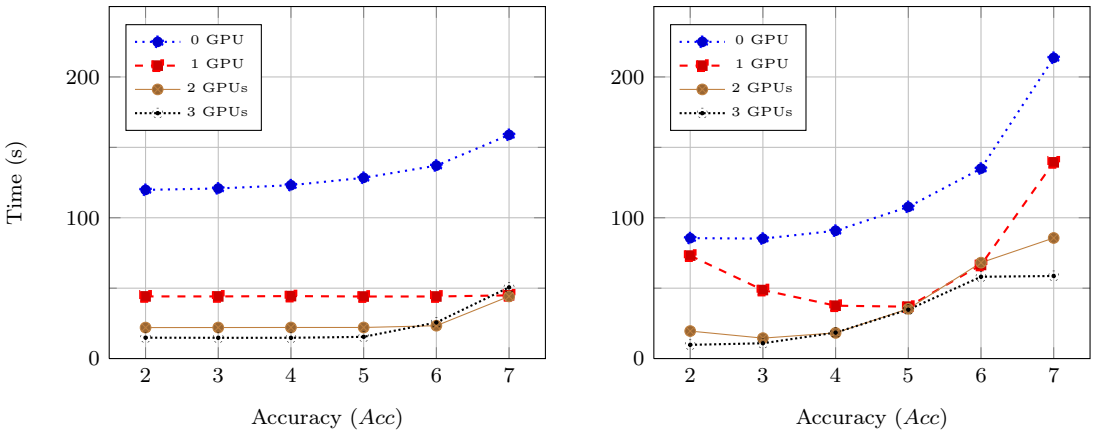
(a) 0 GPU

(b) 1 GPU

(c) 2 GPUs

(d) 3 GPUs

Figure 17: Comparison of timing results (logarithmic scale) for uniform and non-uniform distribution of $N = 38 \cdot 10^6$ particles on $0, 1, 2$ or 3 GPUs and $n_g = 1000$



(a) Uniform particle distribution ($h = 7$)

(b) Non-uniform particle distribution

Figure 18: Minimal timing over all h from Fig. 17

7 Conclusion

We have proposed an original FMM implementation over a runtime system. Thanks to a breakdown of the computational work (Section 4.3), we have shown that FMM presents excellent properties for being efficiently pipelined. We have succeeded to design a task flow that can exploit these properties and achieve a very high parallel efficiency up to 160 CPU cores on a SGI Altix UV 100 machine (Section 4.4). Because almost the entire work load is shared by P2P and M2L tasks (Section 4.3), we have implemented a GPU version of both these kernels. We have shown that when the near-field is dominant (Section 5.3) or comparable to the far-field (Section 5.2), we manage to consistently pipeline the task flow and achieve very well balanced executions by deporting P2P to GPU on demand. Thanks to a highly optimized P2P kernel, we have achieved very high performance. For instance, potentials and forces for 38 million particles could be processed in 13.34 s on our 12 cores Nehalem processor enhanced with 3 M2090 Fermi GPUs. When the far-field is dominant, some M2L tasks also have to be deported to GPUs in order to balance the load. We have shown that an efficient pipeline could be performed even in that case (although the impact on performance itself is less impressive since we did not optimize the GPU M2L kernel).

The successive optimizations discussed in this study result in a single code that gets highly pipelined and balanced across architectures. Since we rely on the so-called black-box method, our code furthermore has a broad range of applications (Section 2.1).

We plan to apply this approach to clusters of heterogeneous nodes. One possibility will consist of statically distributing the work load to the different nodes of the cluster and explicitly handling the inter-node communications. Alternatively, we might also let the runtime handle these communications thanks to the task flow. We also plan to design optimized GPU kernels for the six FMM tasks. Indeed, if all GPU kernels are provided to the runtime, it can limit data movement by processing on individual GPUs connected subparts of the task flow. It will be interesting to assess whether such an approach allows for a better strong scaling.

Acknowledgment

The authors would like to thank Raymond Namyst and Samuel Thibault for their advice on performance optimization with StarPU. 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 https://plafrim.bordeaux.inria.fr/).

References

[1] L. Greengard and V. Rokhlin, "A fast algorithm for particle simulations," *Journal of Computational Physics*, vol. 73, no. 2, pp. 325 – 348, Dec. 1987. [Online]. Available: http://dx.doi.org/10.1016/0021-9991(87)90140-9

[2] ——, "A new version of the fast multipole method for the laplace equation in three dimensions," *Acta Numerica*, vol. 6, pp. 229–269, 1997.

[3] L. Greengard and W. D. Gropp, "A parallel version of the fast multipole method," *Computers & Mathematics with Applications*, vol. 20, no. 7, pp. 63 – 71, 1990.

[4] A. Chandramowlishwaran, S. Williams, L. Oliker, a. G. B. Ilya Lashuk, and R. Vuduc, "Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures," in *Proceedings of the 2010 IEEE conference of IPDPS*, 2010, pp. 1–15.

[5] F. A. Cruz, M. G. Knepley, and L. A. Barba, "Petfmm—a dynamically load-balancing parallel fast multipole library," *Int. J. Numer. Meth. Engng.*, vol. 85, no. 4, p. 403–428, 2011.

[6] E. Darve, C. Cecka, and T. Takahashi, "The fast multipole method on parallel clusters, multicore processors, and graphics processing units," *Comptes Rendus Mécanique*, vol. 339, no. 2-3, pp. 185–193, 2011.

[7] 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*, vol. 180, no. 11, pp. 2066 – 2078, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010465509001891

[8] N. A. Gumerov and R. Duraiswami, "Fast multipole methods on graphics processors," *Journal of Computational Physics*, vol. 227, no. 18, pp. 8290 – 8313, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0021999108002921

[9] T. Hamada, T. Narumi, R. Yokota, K. Yasuoka, K. Nitadori, and M. Taiji, *42 TFlops hierarchical N-body simulations on GPUs with applications in both astrophysics and turbulence*. ACM, 2009, pp. 62:1—62:12. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654123

[10] T. Takahashi, C. Cecka, W. Fong, and E. Darve, "Optimizing the multipole-to-local operator in the fast multipole method for graphical processing units," *International Journal for Numerical Methods in Engineering*, vol. 89, no. 1, pp. 105–133, 2012. [Online]. Available: http://dx.doi.org/10.1002/nme.3240

[11] Q. Hu, N. A. Gumerov, and R. Duraiswami, "Scalable fast multipole methods on distributed heterogeneous architectures," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 36:1–36:12. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063432

[12] I. Lashuk, C. Aparna, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, l. Ying, D. Zorin, and G. Biros, "A massively parallel adaptive fast-multipole method on heterogeneous architectures," in *Proceedings of the 2009 ACM/IEEE conference on Supercomputing*, 2009, p. 1–11. [Online]. Available: http://www.cc.gatech.edu/~gbiros/

[13] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov, "Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs," in *GPU Computing Gems*, W. mei W. Hwu, Ed. Morgan Kaufmann, Sep. 2010, vol. 2. [Online]. Available: http://hal.inria.fr/inria-00547847

[14] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, J. Langou, H. Ltaief, and S. Tomov, "LU factorization for accelerator-based systems," in *The 9th IEEE/ACS International Conference on Computer Systems and Applications, AICCSA 2011, Sharm El-Sheikh, Egypt, December 27-30, 2011*, H. J. Siegel and A. El-Kadi, Eds. IEEE, 2011, pp. 217–224. [Online]. Available: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6122397

[15] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov, "QR factorization on a multicore node enhanced with multiple GPU accelerators," in *IPDPS*. IEEE, 2011, pp. 932–943. [Online]. Available: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6011824

[16] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. A. van de Geijn, "Solving dense linear systems on platforms with multiple hardware accelerators," *ACM SIGPLAN Notices*, vol. 44, no. 4, pp. 121–130, Apr. 2009.

[17] G. Quintana-Ortí, E. S. Quintana-Ortí, E. Chan, F. G. V. 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.

[18] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "Parallel tiled QR factorization for multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 13, pp. 1573–1590, 2008.

[19] J. Kurzak, H. Ltaief, J. Dongarra, and R. M. Badia, "Scheduling dense linear algebra operations on multicore processors," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 1, pp. 15–44, 2010. [Online]. Available: http://dx.doi.org/10.1002/cpe.1467

[20] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Hérault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra, "Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA," in *IPDPS Workshops*. IEEE, 2011, pp. 1432–1441. [Online]. Available: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6008655

[21] "PLASMA users' guide, parallel linear algebra software for multicore architectures, version 2.0," http://icl.cs.utk.edu/plasma, University of Tennessee, November 2009.

[22] "MAGMA users' guide, version 0.2," http://icl.cs.utk.edu/magma, University of Tennessee, November 2009.

[23] F. G. Van Zee, E. Chan, R. A. van de Geijn, E. S. Quintana-Orti, and G. Quintana-Orti, "The `libflame` library for dense matrix computations," *Computing in Science and Engineering*, vol. 11, no. 6, pp. 56–63, Nov./Dec. 2009.

[24] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov, "QR factorization on a multicore node enhanced with multiple GPU accelerators," *Accepted to the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2011)*, 2011.

[25] H. Ltaief and R. Yokota, "Data-driven execution of fast multipole methods," *CoRR*, vol. abs/1203.0889, 2012.

[26] C. Bordage, 2012, personal communication.

[27] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience, Euro-Par 2009 best papers issue*, 2010.

[28] W. Fong and E. Darve, "The black-box fast multipole method," *Journal of Computational Physics*, vol. 228, no. 23, pp. 8712 – 8725, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0021999109004665

[29] L. Ying, G. Biros, and D. Zorin, "A kernel-independent adaptive fast multipole algorithm in two and three dimensions," *J. Comput. Phys.*, vol. 196, no. 2, pp. 591–626, May 2004. [Online]. Available: http://dx.doi.org/10.1016/j.jcp.2003.11.021

[30] M. Messner, M. Schanz, and E. Darve, "Fast directional multilevel summation for oscillatory kernels based on chebyshev interpolation," *Journal of Computational Physics*, vol. 231, no. 4, pp. 1175 – 1196, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0021999111005705

[31] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí, "An Extension of the StarSs Programming Model for Platforms with Multiple GPUs," in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 851–862.

[32] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "Dague: A generic distributed dag engine for high performance computing," *Parallel Computing*, vol. 38, no. 1–2, pp. 37 – 51, 2012, <ce:title>Extensions for Next-Generation Parallel Programming Models</ce:title>. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819111001347

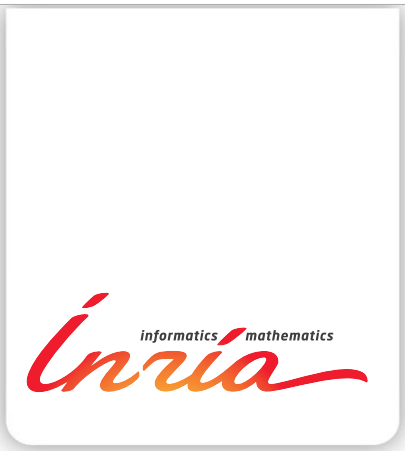
[33] G. F. Diamos and S. Yalamanchili, "Harmony: an execution model and runtime for heterogeneous many core systems," in *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*. New York, NY, USA: ACM, 2008, pp. 197–200.

[34] K. Fatahalian, T. Knight, M. Houston, M. Erez, D. Horn, L. Leem, J. Park, M. Ren, A. Aiken, W. Dally, and P. Hanrahan, "Sequoia: Programming the memory hierarchy," in *ACM/IEEE SC'06 Conference*, 2006.

[35] P. Jetley, L. Wesolowski, F. Gioachin, L. V. Kale, and T. R. Quinn, "Scaling hierarchical N-body simulations on GPU clusters," in *SC'10 USB Key*. New Orleans, LA, USA: ACM/IEEE, Nov. 2010.

[36] M. S. Warren and J. K. Salmon, "A parallel hashed oct-tree n-body algorithm," in *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, ser. Supercomputing '93. New York, NY, USA: ACM, 1993, pp. 12–21. [Online]. Available: http://doi.acm.org/10.1145/169627.169640

[37] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, vol. 23, pp. 187–198, Feb. 2011. [Online]. Available: http://hal.inria.fr/inria-00550877



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