The Fast Multipole Method on the Cell processor

Pierre Fortin, Jean-Luc Lamotte

The Fast Multipole Method on the Cell processor

Pierre Fortin and Jean-Luc Lamotte
UPMC Univ Paris 06 and CNRS UMR 7606, LIP6,
4 place Jussieu, F-75252, Paris cedex 05, France
Contact: pierre.fortin@lip6.fr

Abstract

This paper presents the first deployment of the Fast Multipole Method on the Cell processor (PowerXCell 8i). We rely on the matrix formulation with BLAS routines of the FMB code (Fast Multipole with BLAS) in order to directly and efficiently offload the most time consuming operators of both far field and near field computations on the Cell heterogeneous cores. We detail the difficulties that had to be solved first, and we finally obtain a deployment in single and double precisions, which scales linearly on several Cell blades and which is able to handle both uniform and non-uniform distributions of particles. We also present our performance results and comparisons with multicore CPUs, as well as the limitations of our deployment on the Cell processor.

1 Introduction

Motivation. The *Fast Multipole Method* (FMM) [1] solves the *N*-body problem for any given accuracy with linear runtime complexity against quadratic complexity for the direct evaluation. This algorithm is considered as one of the most important in scientific and engineering computing [2] and is a key improvement for particle simulations in molecular dynamics, astrophysics, electromagnetics, fluid mechanics and many more. Thanks to an octree data structure, the potential or force field is decomposed in a near field part, directly computed, and a far field part approximated with multipole and local expansions. The FMM is considered as a challenging application since this algorithm is highly non-trivial and presents several phases with different computational intensities and different (possibly irregular) memory access patterns [3]. Efficiently deploying such complex algorithm on different hardware accelerators (HWAs) such as the SPE (Synergistic Processing Element) cores of the Cell processor, Graphics Processing Units (GPUs) and the Xeon Phi, is thus an important challenge.

Related work. *N*-body simulations via direct computation [4, 5] or with cut-off radius [6, 7, 8] have already been implemented on the Cell processor. To our knowledge no FMM deployment has ever been done on the Cell processor. In our opinion, this is due to these challenging features of the FMM algorithm and to the important programming efforts required to efficiently optimize the FMM far field operators on the Cell SPE cores.

The FMM has however been recently deployed on GPUs for both uniform [9, 10] and non-uniform [3, 11, 12, 13, 14, 15] distributions of particles. These implementations also scale on multiple CPU-GPU nodes. The far field part is generally less efficiently implemented on GPU than the near field part (direct computation) which results in an octree decomposition on GPU which favors the direct computation [3, 9, 11]. Moreover, such implementations are based on a thorough deployment of the near field and/or far field computations of the FMM on the GPU, along with all the data structures, which requires important algorithmic changes and programming efforts. This results in GPU or hybrid CPU-GPU implementations that outperform CPU implementations (it can also be noticed that highly optimized CPU implementations can diminish the CPU-GPU performance gap [11]).

Approach and contributions. Our approach will rather focus on offloading only the most time consuming operations on the HWA in a straightforward and efficient way. In this purpose, we rely on the FMB code (Fast Multipole with BLAS) [16, 17] which presents a matrix formulation of the multipole-to-local (*M2L*) operator for Laplace equation. In the far field computation this *M2L* operator, which converts a multipole expansion into a local expansion, corresponds indeed to the most time consuming part.

Thanks to the level 3 BLAS (Basic Linear Algebra Subprograms) routines, which are highly efficient routines performing matrix-matrix operations, this code offers substantial runtime speedup on CPUs for the targeted precisions in astrophysics and in molecular dynamics.

Provided that BLAS routines are available on a HWA (which is usually the case), we can directly use these BLAS implementations in order to efficiently perform the *M2L* computations on this HWA, for any required precision. Contrary to other FMM implementations on HWA, we do not have to write and highly optimize specific FMM operators for each new HWA. Moreover, we rely on the ability of the FMB code to group multiple *M2L* operations into one single matrix-matrix product: this enables to increase the computation grain and thus offset the cost of offloading *M2L* operations on the HWA. It has to be noticed that a similar idea has been concurrently developed for GPUs in [18], where, referencing the same FMB approach [16, 17], multiple *M2L* operations are performed at once (but without BLAS routines) on the GPU. The portability and straightforwardness of our approach may however lead to a lower overall performance for the FMM compared to thorough, non portable, deployments such as those done on GPUs.

The other most time consuming part of the FMM is the direct computation of the *N*-body problem, which is also a key application for new HWAs and is therefore among the first applications to be efficiently implemented on these HWAs: see for example [19] for GPU. Therefore the direct computation part of the FMM can usually also be efficiently performed on a new HWA.

Finally, the most time consuming operations of the near and far field computations correspond to small or medium computation grains, which are moreover involved in irregular computation schemes due to the possible non-uniform distributions of particles. That is why we have first targeted in this paper the Cell processor, and its SPE cores, whose internal bus has lower latency and higher bandwidth than the PCI Express bus of GPUs.

This paper presents thus the following contributions.

- To our knowledge, this is the first deployment of the FMM on the Cell processor. Our implementation scales efficiently on several Cell blades, supports both single and double precision computations and can handle both uniform and non-uniform distributions of particles.
- We extend our near field part computation in single precision on the Cell processor [20] to its double precision version.
- Since the latest IBM Cell SDK [21] does not provide efficient BLAS routines for complex numbers on the Cell processor, we had to write our own implementations. We thus extend our efficient implementation of the CGEMM BLAS routine (single precision complex matrix-matrix multiplication) [22] to its (previously unpublished) double precision version (ZGEMM).
- We also present here the scheduling of these matrix products in the far field computation in order to efficiently perform the *M2L* computations on the SPEs. Moreover, we show how data movements in the Cell main memory can efficiently be performed directly by the SPEs.
- We detail our performance results and comparisons with multicore CPUs, as well as the limitations of our approach on the Cell processor.

Even if IBM has announced in November 2009 that the next Cell processor with 32 SPEs will not be released, we believe that the results and feedback of this work will be useful for the deployment of the FMM (as well as of other challenging scientific applications) on current and forthcoming HWAs.

In the following, we will first present the FMM and the FMB code in Sect. 2. Then, in Sect. 3 we will detail our deployment of the FMB code on the Cell processor, and we will present performance results in Sect. 4. Finally, concluding remarks will be presented in Sect. 5 and future work will be discussed.

2 The parallel Fast Multipole Method with BLAS routines

In this section, we briefly present the Fast Multipole Method and the FMB implementation which relies on BLAS routines. More details can be found in [1, 16, 17, 23].

2.1 The Fast Multipole Method

In the FMM algorithm, a hierarchical decomposition of the particle space thanks to an octree enables to efficiently divide the potential or force computation into a near field part (directly computed) and a far field part (approximately computed). For Laplace

equation in astrophysics and in molecular dynamics, the far field is computed via multipole and local expansions based on spherical harmonics. The maximum expansion degree, denoted by P, determines the accuracy of the computation. Greater P values imply more precise approximations of the far field, but the classic FMM operation count grows as $\mathcal{O}(P^4N)$ for N particles.

The algorithm requires first an upward pass of the octree to compute the multipole expansions of all cells in the octree (particle-to-multipole, or *P2M*, and multipole-to-multipole, or *M2M*, operations). During a downward pass of the octree, the local expansion of each cell c is then computed from the conversion of the multipole expansions of all cells in its *interaction list* (multipole-to-local, or *M2L* operation). All cells in the interaction list of c are indeed *well-separated* from c which allows the *M2L* conversion. The interactions due to cells further than the interaction list of c are taken into account thanks to the local expansion translation of the parent cell of c (local-to-local, or *L2L* operation). After the downward pass, the far field is deduced in each octree leaf c_l from the evaluation of the local expansion of c_l (local-to-particle, or *L2P* operation). The near field part (particle-to-particle, or *P2P* operation) for particles within c_l is finally added to the far field part. More precisely, we directly compute all the interactions between each particle contained in c_l and each particle contained in each of its nearest neighbors: these are the direct *pair* computations. Interactions between each pair of particles contained in c_l must then also be computed directly: this is the direct *own* computation. The mutuality of gravity (or mutual interaction principle) is used for both *pair* and *own* computations to halve the direct computation cost.

As far as non-uniform distributions of particles are concerned, the FMB code relies on an adaptation of the algorithm presented in [24] where the octree height is fixed by the user: the possible numerous empty cells are simply skipped (see [23] for more details). It can be noticed that a more adaptive algorithm has been introduced in [1], but in the context of parallel computing in multi-process mode the FMB algorithm offers predictable communication patterns [23]. The octree height H is optimally set by the user (unless otherwise mentioned) in order to minimize the overall computation time: the H value must balance at best the near field and far field computations, depending on P and on the particle distribution.

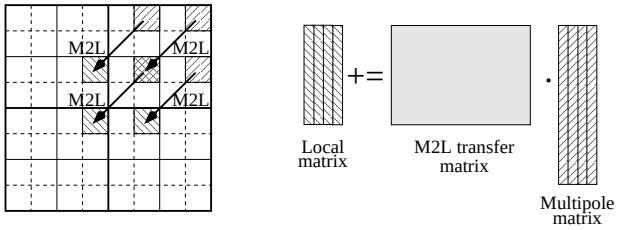


Figure 1: Matrix-matrix product resulting from the grouping of multiple *M2L* operations

2.2 Matrix formulation

Matrix formulations of some of the FMM operations usually correspond to matrix-vector operations [3, 9]. The FMB code relies on its matrix formulation to group multiple *M2L* operations in matrix-matrix products with complex elements [16]. Since matrix-matrix products require $\mathcal{O}(N^2)$ memory storage relatively to $\mathcal{O}(N^3)$ operation count, it is indeed easier to overlap memory latency with computation with level 3 BLAS than with level 2 BLAS, and thus to reach the CPU peak performance. More precisely, the FMB code offers three different computation schemes with level 3 BLAS. The first scheme uses *recopies* to gather and scatter multipole and local matrices from and to multipole and local expansions. As presented in Fig. 1, the multipole matrix is then multiplied by a *M2L* transfer matrix and the result is added to the corresponding local matrix. The other computation schemes correspond to specific data storage modes (by *rows* or *slices*) which can avoid these recopies in uniform areas of the particle distribution [17]. As a reference implementation [12], the $\mathcal{O}(P^3)$ computation scheme with rotations has also been implemented for *M2L* in FMB [16].

2.3 Hybrid MPI-thread parallelization

An hybrid MPI-thread parallelization of FMB has been proposed in [23], where the multi-thread parallelization is based on POSIX threads. The data locality and the load balancing among the processors are ensured by a static octree decomposition based on Morton ordering and appropriate cost functions. Because of the mutual interaction principle, write/write conflicts can sometimes occur between two threads. These conflicts are treated through mutual exclusion mechanisms for all particles of an octree leaf at a time: a single bit per leaf (*lock bit*) is used to detect

conflicts, and FIFO (*First In First Out*) data structures enable to postpone the conflicting operation. As far as BLAS routines are concerned in multi-thread mode, each BLAS call is performed sequentially since the matrix sizes are relatively small or medium. MPI communications are *sender-driven* thanks to the algorithm chosen for non-uniform distributions of particles, small messages are aggregated into bigger ones and communications are overlapped with computation.

Compared to a pure MPI code, this hybrid MPI-thread parallelization of FMB enables a gain in parallel efficiency thanks to a better load balancing between threads and to more use of the mutual interaction principle in the direct computation [23]. The memory scalability is also improved since the octree data structure is shared by all threads inside each MPI process.

3 Deployment on the Cell processor

We now detail how we adapt the matrix formulation and the MPI-thread parallelization of the FMB code in order to deploy it on the Cell processor. We consider in this paper only the force computation (no potential computation) in both single precision (SP) and double precision (DP) floating point arithmetic.

As presented in Sect. 1, our deployment will consist in offloading only the two most time consuming operators of the FMM, namely *P2P* and *M2L*. Since the near field and far field computations should be balanced, the *P2P* operator represents indeed roughly half of the total computation time. Moreover, the *M2L* operator is clearly the most time consuming operator of the far field computation since it has to be applied to all members (up to 189) of the interaction list of each cell. We will thus offload all *P2P* and *M2L* computations on the SPE (Synergistic Processing Element) cores of the Cell processor, which are specialized for high performance computing, while keeping the other computations on the PPE (PowerPC Processing Element), which is a general-purpose core.

The main features of the Cell architecture are detailed below.

3.1 The Cell processor

The Cell processor is composed of one PPE core and eight SPE cores. In this paper, we focus on the PowerXCell 8i version which offers efficient computations in double precision. All cores have an in-order execution and runs at 3.2 GHz. The PPE has a two-way simultaneous multi-threading, while each SPE has a Synergistic Processor Unit (SPU) and a 256 KB local store (LS) to store both data and SPE kernel code. This SPU has 128 128-bit SIMD (Single-Instruction, Multiple-Data) registers and two instruction pipelines, referred to as pipeline 0 (even) and pipeline 1 (odd), which can each issue and complete one instruction per cycle. The even pipeline handles mainly the integer and floating-point units, whereas the odd pipeline handles the remaining instructions, including load, store and shuffle. In the PowerXCell 8i, all SP and DP floating-point operations are fully pipelined, with a latency of 6 cycles in SP and 9 cycles in DP. Loads and stores from/to the LS also have a fixed latency of 6 cycles which enables the programmer to completely control the LS memory accesses to load/store data to/from the SIMD vector units: with these two independent pipelines, memory accesses can thus be overlapped with computation. Moreover, communications are required between each LS and the Cell main memory through explicit DMA (Direct Memory Access) instructions over the EIB (Element Interconnect Bus). DMA transfers are performed concurrently with SPU computation, which allows very efficient overlap of DMA transfers with computation. The total peak performance of the eight SPEs of a PowerXCell 8i is $8 \times 12.8 = 102.4$ Gflop/s in double precision and $8 \times 25.6 = 204.8$ Gflop/s in single precision. It also has to be noticed that while the DP floating-point operations are IEEE-compliant on the PowerXCell 8i SPUs, the SP floating-point operations only support the truncation rounding mode. More details about the Cell architecture and programming can be found in [25, 26].

The overall Cell architecture presents thus three levels of parallelism: SIMD parallelism on the SPU vector units, multi-thread parallelism among the SPEs, and MPI multi-process parallelism on multiple processors or on multiple IBM QS22 blade servers. These QS22 blades are NUMA architectures which contain two PowerXCell 8i processors and up to 32 GB of DDR2 memory shared by the two processors. A process running on a QS22 blade can thus access the 16 SPEs of the two PowerXCell 8i.

3.2 Direct computations on the SPEs

We now present the deployment of the direct computation (*P2P* operator) on the SPE of the Cell processor, starting with the SPE computation kernels.

3.2.1 Direct computation kernel in single precision

The efficient implementation of the *P2P* kernel on the SPE, for both the *pair* and *own* computations, has been detailed in single precision in [20], and is recalled here. We emphasize that in our implementation we aim at exploiting at most the mutual interaction principle. Firstly, as we may have to treat low numbers of particles per leaf, we have chosen to compute each *pair* or *own* computation on one SPE only. In practice, multiple *pair* or *own* computations will therefore be performed in parallel on the up to 16 SPEs of one QS22 blade. Secondly, SIMD vectorization for the SPE code requires a "structure of arrays" (SOA) data layout [8] which had to be implemented in the FMB code. The SPE vector register size leads us to compute together *blocks* of 4 bodies: this implies array padding with zero mass bodies so that the array sizes are multiples of 4. The key insight here is to have enough instructions in the body of the internal loop. The compiler can then reorder instructions in order to overlap at best LS memory accesses with computation.

For the *pair* computation code between two leafs, this is accomplished thanks to the numerous SPE vector registers which enable us to compute together all the 16 computations among the two *blocks* of 4 bodies. The required quadword rotates are dual-issued with floating point instructions, and we hence obtain 8 body loads (from local store to registers) for 16 computations. The internal loop has also been unrolled manually which eventually offers 32 computations for 12 body loads, and the instructions have been manually interleaved in the C code.

The *own* computation requires special treatment when computing interactions among the same 4 bodies (*own block* computation): only 12 interactions are then performed without the mutual interaction principle. Interactions between distinct *blocks* are treated with the *pair* computation code and with the mutual interaction principle.

Finally, we use the IBM SDK `_rsqrtf4` inlined vector function for the square root reciprocal which uses a floating-point reciprocal square root estimate (which has low latency and can be pipelined and dual-issued with floating point instructions) and one Newton-Raphson iteration to match floating point single precision.

For each interaction in the *pair* computation, the final code requires hence 27 flops[1]. Since we use the mutual interaction principle in this *pair* computation, this results in 13.5 flops per interaction. Among these 27 floating point operations, 14 are written with 7 fused multiply-add (FMA) instructions. The theoretical peak performance of such computation in SP is thus 67.5% of the SPE peak performance, namely 17.28 Gflop/s on one SPE. It can be noticed that an *own* block computation leads to 24 flops per interaction since no mutual interaction is computed in this case. We will consider 13.5 flops per interaction in the overall near field computation which will thus underestimates the flops used in the *own* block computations.

It can also be noticed that the same techniques have been applied concurrently in [4] to perform one *own* computation (with thousands of bodies) on several SPEs, but without the mutual interaction principle and without the Newton-Raphson iteration.

3.2.2 Direct computation kernel in double precision

We now present how we have adapted the *P2P* kernel to double precision for this paper. We use the `_rsqrtd2()` inlined vector function of the IBM SDK 3.1 which uses the same floating-point reciprocal square root estimate as `_rsqrtf4`, but with three Newton-Raphson iterations in order to match floating point double precision.

For the sake of simplicity, we pad our array in DP with zero mass bodies so that their sizes are multiples of 4, exactly like in SP. This enables us to rely on the SP *pair* computation code with two *blocks* of 4 bodies. The internal loop unrolling does not improve performance here.

In DP the *pair* computation kernel requires finally $39/2 = 19.5$ flops per interaction thanks to the mutual interaction principle. According to the number of FMA used, the theoretical peak performance of such computation in DP is 65% of the SPE peak performance, namely 8.32 Gflop/s on one SPE.

3.2.3 Task scheduling

We briefly recall here the task scheduling of the *P2P* computations on the SPEs presented in [20], which has to yield to a responsive PPE code that will minimize the time where the SPEs are idle. We define one *P2P task* as the *own* computation of a target leaf T

[1] Following [8], we do not count the reciprocal square root estimate which can be dual-issued with floating point instructions.

along with all the *pair* computations between T and its nearest neighbors.

First, it has to be noticed that in the FMB octree data structure all bodies of an octree leaf are contiguous in memory, which is suitable for efficient DMA transfers. The bodies of a leaf are then transferred by chunk of at most 512 bodies. DMA transfers are overlapped with computation on the SPE thanks to three shared I/O buffers (each containing at most 512 bodies) in the LS and *fenced* DMA operations [26]. When considering a given *P2P task*, we manage to have only the first target chunk read and the last target and neighbor chunk writes not overlapped with SPE computation.

All *P2P* computations will be performed on the SPEs: the static load balancing of the multi-thread parallelization of FMB [23] is therefore suitable for balancing *P2P* computations among the homogeneous SPEs. In the multi-thread parallelization presented in [23], the write/write conflicts are however solved thanks to lock bits which are set and unset for each *pair* or *own* computation. This corresponds to fine-grained locks and fined-grained computations which may imply too strong synchronization overhead for our Cell deployment. We have thus move our locking strategy to the *P2P task* level: we now set together the lock bits of the current leaf and of all its required nearest neighbors. If some lock bits are already set for another SPE, meaning that the force vectors of these bodies are currently being updated, we use the same FIFO data structures as in [23] to postpone the whole conflicting *P2P task*. This increases the computation grain but may lead to deadlocks. Instead of using multiple POSIX threads on the PPE (one for each SPE), we therefore prefer to have one single thread on the PPE that will manage all SPEs. This way, the deadlocks are indeed easily avoided, no mutexes are required to set or unset the lock bits and this also avoids costly context switches between multiple threads on the PPE. All this results in a fast single thread code which will help the PPE to be more responsive to all SPEs.

Finally, we can use up to 4 *slots* (2 in practice) to be able to assign (with mailbox messages of the Cell processor) several *P2P tasks* to each SPE at any time. The end of each *P2P task* is notified by the SPE to the PPE thanks to DMA writes in the Cell main memory, which is the fastest notification approach [25].

As shown in Fig. 2, all this results in an efficient overall near field computation on one SPE for both single and double precisions: we reach up to 14.5 Gflop/s in SP, and up to 5.5 Gflop/s in DP. On one Cell processor, we obtain up to 115.8 Gflop/s in SP and up to 230.4 Gflop/s on one blade. Detailed performance results can be found in [20]. This compares favorably with results from the literature for the direct computation with 8 SPEs: depending on the considered forces and on the number of particles, the performance results in SP vary between 45 Gflop/s [6], 60 Gflop/s (for 6 SPEs) [7] and 83 Gflop/s [5]. 61% of peak has been obtained in SP in [4] when considering 20 flops per interaction. In DP, 34 Gflop/s performance on a PowerXCell 8i processor has been presented in [8].

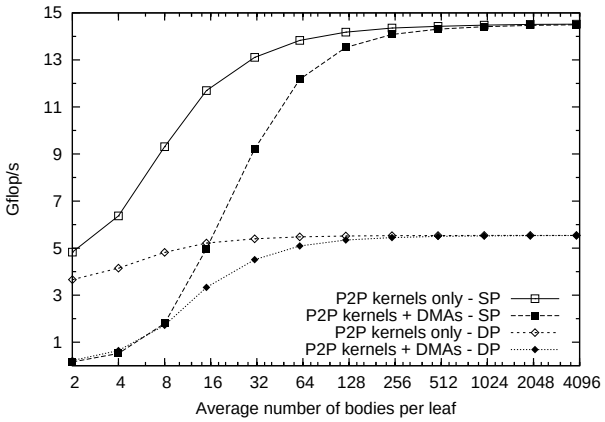


Figure 2: Near field computation on one SPE for increasing average number of particles per leaf with uniform distributions. We use several distributions with different number of particles (from 500 000 to 2 millions) and different octree heights (from 3 to 6) in order to obtain various average numbers of particles per leaf. Gflop/s rates are computed by considering 13.5 flops per interaction in SP, and 19.5 in DP

Moreover, for an average number of particles per leaf close to 128 or more, the $\mathcal{O}(N)$ DMA transfer times are overlapped with the $\mathcal{O}(N^2)$ computation times, and the overall near field computation performs like the *P2P* computation kernel. As detailed in [20], for such average numbers of particles per leaf we also have very good parallel speedups with multiple SPEs (up to 16) for uniform distributions, as well as for non-uniform distributions like cylinders.

3.3 Multipole-to-local computations on the SPEs

The latest IBM Software Development Kit (SDK 3.1, see [21]) provides complex BLAS routines that run only on the PPE, not on the SPEs, yielding limited performance. We therefore have to implement our own SP and DP complex matrix products for the SPEs in order to offload the *M2L* computations on the SPEs.

3.3.1 Design considerations

We first present our design considerations according to the FMM and FMB requirements.

We will consider the following matrix-matrix product: $C+ = A \times B$, where C is a $M \times N$ matrix of complex elements, and A and B are respectively $M \times K$ and $K \times N$ matrices of complex elements. As presented in Sect. 2.2, B will correspond to the multipole matrix, C to the local matrix and A to the *M2L* transfer matrix in the FMB code. This imposes $M = (P+1)(P+2)/2$ and $K = (P+1)^2$.

Firstly it has to be noticed that with a complete BLAS library optimized for SPE, we could have been able to treat all P values, as already done on CPU [16]. Since we have to develop our own optimized BLAS routines, we will focus only on few specific P values, which yield to specific matrix sizes. We will thus focus on the following P values and the corresponding M and K values:

- for medium precisions we choose $P = 7$, which implies $M = 36$ and $K = 64$,
- for high precisions we choose $P = 15$, which implies $M = 136$ and $K = 256$,
- for very high precisions we choose $P = 23$, which implies $M = 300$ and $K = 576$.

It can be noticed that P values lower than 7 (for low precisions) can be used in astrophysics for example. However as detailed in [17], no BLAS computation are performed in such simulations since the BLAS routines are mainly efficient for greater P values or within large uniform areas, which are not present in astrophysical particle distributions. As far as N is concerned, it is independent of the required precision and can be considered as much greater than K. In this deployment on the Cell processor, we focus indeed only on the scheme with *recopies* (see Sect. 2.2) since it can be used for both uniform and non-uniform particle distributions, and since it offers greater N values: up to 2744 (respectively $27,000$) for an uniform octree with height 5 (resp. 6) [16]. In practice, we use two buffers of sizes $M \times N_{max}$ and $K \times N_{max}$, with $N_{max} = 2048$, for respectively the local matrix (C) and the mutipole matrix (B). With the scheme with *recopies*, these buffers are filled with multipole and local expansions that share the same *M2L* transfer matrix, and a matrix product is performed when all possible expansions have been recopied, or each time these buffers are full. Since these are relatively small or medium sizes for M and K, we plan to use only one SPE per matrix product. However, we will consider multiple matrix products in parallel (up to 16 on one QS22 blade), since many matrix products can be performed concurrently in the downward pass of FMB.

Secondly, in [16] two different *M2L* kernel heights (*single* or *double*) were presented which lead to two different *M2L* transfer matrices (*sparse* or *dense*). For our targeted P values, BLAS computations are more efficient with the *double height* M2L *kernel* [16]. Moreover this latter eases the implementation of the matrix product since the corresponding *M2L* transfer matrix is dense. We will therefore only consider the double height *M2L* kernel in this paper.

Thirdly, we detail our matrix data layout. For portability purposes, the FMB code relies on the standard C BLAS interface[2], where each complex matrix is considered to be stored as a single memory array of complex elements. The real and imaginary parts of the different complex elements are thus interleaved in memory. Both multipole and local matrices are built by concatenating multipole (respectively local) expansions as column vectors. We therefore favor the column-major format. However, as pointed out by several authors [27, 28], the BDL (Block Data Layout) format enables to save DMA transfers, to exploit at best the memory banks of the main memory and to minimize TLB (Translation Look-aside Buffer) misses which are especially costly when generated by the SPE [25]. Fortunately, this conversion to BDL format can be performed at no cost in FMB: the *M2L* transfer matrices are built and precomputed at the begin of the algorithm, and the gather operations of the scheme with *recopies* can be directly performed with multipole and local matrices in BDL storage mode. We therefore consider that the matrices are stored in BDL format, with column-major order within each block, as well as among the different blocks. Moreover, as the *M2L* transfer matrices are

[2]See: http://www.netlib.org/blas/blast-forum/cinterface.pdf

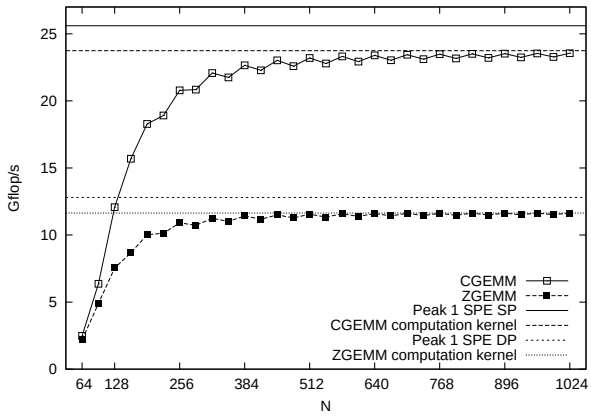


Figure 3: Performance on one SPE for our CGEMM and ZGEMM implementation with square matrices of size $N \times N$

precomputed, we could have freely chosen between $C+ = A \times B$ (1) and $C+ = A^T \times B$ (2). If (2) is usually preferred on classic CPU architectures since the inner loop then matches the cache lines, there is no such cache effect in the SPE local store. Moreover the SIMD computation necessitates additional floating point operations with (2), which prevents to reach optimal peak performance in this case [29]. That is why we have chosen to build the *M2L* transfer matrices in FMB so that we can consider $C+ = A \times B$.

We finally present the programming choice for our computation kernel that runs on the SPE. In the case of real matrix products on the Cell processor, excellent performance in single precision (for example 89.88% of peak in [30]) has already been obtained when programming in C (with C SPU intrinsics), and almost peak performance (99.80% of peak) has been obtained when writing in assembly code [29]. As far as real double precision matrix products are concerned, a kernel in assembly has been developed in [31] using techniques similar to those presented in [29], reaching 99.87% of the SPE peak. We focus here on C programming only (with SPU intrinsics) in order to build efficient complex matrix products within a reasonable development time, and with performance similar to the real matrix products.

3.3.2 Matrix multiplication in single precision

Following [29], we have implemented in [22] a single precision complex matrix product (CGEMM BLAS routine) as a tile (or block) operation. We have chosen a tile size of 32×32, mainly for optimal DMA and memory bandwidths. This also enables K to be a multiple of the tile size for our targeted precisions ($P = 7$, $P = 15$, $P = 23$). Moreover, for M we limit to 4, 8 or 12 (all multiples of 4) the number of extra lines of the C matrix to compute, for respectively $P = 7$, $P = 15$, and $P = 23$. These extra lines are currently computed with a complete extra tile, but this could be improved by avoiding the useless computations with a specific kernel.

At first glance, performing a $N \times N$ complex matrix product involves twice more data than a $N \times N$ real matrix product, while requiring four times more floating point operations. Compared to the real case, the complex case thus seems more favorable to an high performance implementation on the Cell SPE but numerous additional shuffle operations (permutations of vector elements) are required to separate real and imaginary parts for the SIMD computation on the SPE. We have thus introduced in [22] an original computation scheme that can reduce the impact of such extra operations depending on the loop unrolling amount. More precisely, the choice of the unrolling amount for each of the three loops of the tile computation kernel is equivalent to determine a sub-tile size (m, n, k) whose computation will be completely unrolled. Setting $m = 4$ to keep multiples of 4 in the M loop, our original computation scheme enables to reduce the number of odd pipeline operations: more details can be found in [22]. When considering load, store and shuffle operations on the odd pipeline, a sub-tile size of ($m = 4$, $n = 8$, $k = 32$) then enables to dual-issue the 640 odd pipeline operations with 1024 floating point operations on the even pipeline, while keeping 384 spare slots on the odd pipeline for extra operations (like jump instructions, or pointer and index arithmetic). Thanks to software pipelining, loop linearization and double-buffering technique to overlap LS memory accesses with computation, we have managed to reach 23.74 Gflop/s for the computation kernel of a 32×32 tile, which is 92.7% of the SPE peak performance. When adding DMA transfers to perform a complete matrix product from the Cell main memory, we maximize the use of double-buffering (with shared I/O buffers for C) to overlap DMA transfers with tile computation on the SPE. We also combine huge (16 MB) pages with NUMA node and memory bindings [25]. As shown in Fig. 3, for increasing matrix size N the $\mathcal{O}(N^2)$ DMA transfer times rapidly become completely overlapped with the $\mathcal{O}(N^3)$ computation time. Detailed performance

results can be found in [22]. Moreover, multiple matrix products can be computed concurrently and efficiently on the SPEs and, for the relatively small or medium sizes of the matrix used in FMB (see Sect. 3.3.1), it is actually more efficient to perform multiple matrix products with one matrix product per SPE, than to perform one single matrix product on multiple SPEs [22].

3.3.3 Matrix multiplication in double precision

We now focus on adapting our single precision CGEMM implementation written in C to an efficient double precision ZGEMM implementation.

We first keep the 32×32 tile for the same reasons as presented in Sect. 3.3.2 and since such tile size still fits in a single DMA transfer (16 KB). We then choose a sub-tile size of ($m = 4$, $n = 4$, $k = 32$) which offers 320 spare slots on the odd pipeline for 1024 floating point operations on the even pipeline: all required odd pipeline operations can thus be dual-issued with even pipeline operations. The same techniques presented for CGEMM in Sect. 3.3.2 apply to ZGEMM, taking into account that the pipeline latency is 9 cycles for DP instructions (against 6 cycles for SP instructions).

Our ZGEMM computation kernel written in C reaches 11.64 Gflop/s on one SPE, which corresponds to 90.95 % of the 12.8 Gflop/s peak performance of the SPE in DP. Like our CGEMM implementation, our ZGEMM implementation offers very good performance as presented in Fig. 3. Like in SP, the DMA transfer times become rapidly minority in DP for increasing matrix size.

3.3.4 Task scheduling and data movements

The task scheduling of the *M2L* computations on the SPEs directly rely on the multi-thread parallelization presented in [23]. We therefore use here multiple POSIX threads on the PPE, namely one POSIX thread for each SPE. Each thread offloads on its SPE all *M2L* computations in its octree part, as determined by the static octree decomposition. The static load balancing is indeed suitable for balancing all *M2L* computations among the homogeneous SPEs. An *M2L task* corresponds thus to a complete matrix product between a multipole matrix, the corresponding *M2L* transfer matrix and the local matrix (see Sect. 3.3.1). The end of each *M2L task* is notified by the SPE to the PPE thanks to a mailbox message, since the computation grain of the matrix products is usually coarse enough. Moreover, there is no possible write/write conflicts between threads for the *M2L* computations, and thus no need for locking.

However, this implies that all *recopies* have to be performed on the PPE which can bottleneck the overall performance with multiple SPEs. We therefore present an alternative version where the SPE directly perform the *recopies*: each SPE gathers on the fly all necessary multipole and local expansions for one tile product in its local store, and then scatters back the local expansions in main memory after the tile product. We thus remove the cost of performing the *recopies* first on the PPE, at the expense of more scattered data accesses for the SPEs in the main memory. Moreover, the SPEs can perform these gather and scatter operations in parallel, which can solve the possible bottleneck of the *recopies* on the PPE. This version implies greater memory requirements since all multipole and local expansions in FMB now have to be aligned on a 128-byte boundary for DMA peak performance, and must be padded to a multiple of the CGEMM/ZGEMM tile size (32 in practice here). In this version, we also have to use DMA list commands to perform both gather and scatter operations on the multipole and local expansions. As presented for example in [32], we therefore have to ensure that all memory buffers in the FMB code, each containing one expansion, never cross a 4 GB boundary. We also have to issue multiple DMA list commands when the expansion buffers of a single gather or scatter operation are stored in multiple 4 GB regions.

We have also considered the use of 16MB memory huge pages. When the *recopies* are performed on the PPE, these huge pages can be used to store the multipole and local matrices, which improve performance for large matrix products [22]. When the SPEs directly perform the *recopies* on the fly, we can also use huge pages to store all local and multipole expansions in FMB. This may require hundreds of 16MB memory huge pages, but enables to reduce page table and TLB (Translation Look-aside Buffer) thrashing [25]. Indeed the TLB of the SPE has only 256 entries, which may be insufficient for the gather and scatter operations performed by the SPEs in all the Cell main memory.

Finally, code overlays are used to successively run the *P2P* and *M2L* kernels on the SPEs, and we directly rely on the FMB MPI parallelization (see Sect. 2.3) to obtain a complete fast multipole method that runs on multiple QS22 blades.

4 Performance results

All tests are performed on four QS22 blades located at the HPC@LR Competence Center in High-Performance Computing (Languedoc-Roussillon region, France). Each QS22 blade has 16 GB of memory and the four blades are linked with a gigabit Ethernet network. A Yellow Dog Linux distribution with 4KB memory pages is installed on each blade. We use the IBM SDK 3.1, the Open-MPI library (version 1.4.2) and the `gcc` compiler for the PowerXCell 8i (version 4.3.2). The *P2P* kernel is compiled with the classical optimization options (`-O3 -funroll-loops -fmodulo-sched -ftree-vectorize -ffast-math`, see [25]) whereas optimal performance is obtained for the *M2L* kernel with the `-Os` option (optimal code size). As far as NUMA node and memory bindings are concerned, we always prefer the node 0 of the QS22 when using 8 SPEs (or less) since it delivers better memory access performance [25]. When using the 16 SPEs of the QS22, we use an interleaved NUMA policy. The QS22 blades have been configured so that the number of available 16MB memory huge pages (HP) is 768.

We will study here both uniform and non-uniform distributions of particles. The uniform distribution will consist in a 3D cube, whereas the non-uniform distribution will consist in a cylinder in a 3D space where the particles are uniformly distributed on the 2D surface of the cylinder. Moreover, in the FMM the norms of the multipole and local expansion terms present large varying magnitudes when P increases (see for example [16]). For numerical stability purposes, further detailed in Sect. 4.3, we will thus use single precision computations only for $P = 7$ (in much the same way as in [12]), and we will use double precision computation for $P = 15$, $P = 23$ as well as also for $P = 7$.

Finally, for FMB timings we consider only one FMM computation, and we do not take into account the octree's construction and the parallel decomposition since these can be considered as precomputation steps, whose costs can be amortized over several simulation time-steps.

4.1 The downward pass

We first study the downward pass computation times on one QS22 blade. In our tests, these times depend on the octree height only, not on the number of particles. We emphasize that these timings encompass both the *M2L* and *L2L* operations: while the *M2L* operations are offloaded on the SPEs, the *L2L* operations are all performed on the PPE single core.

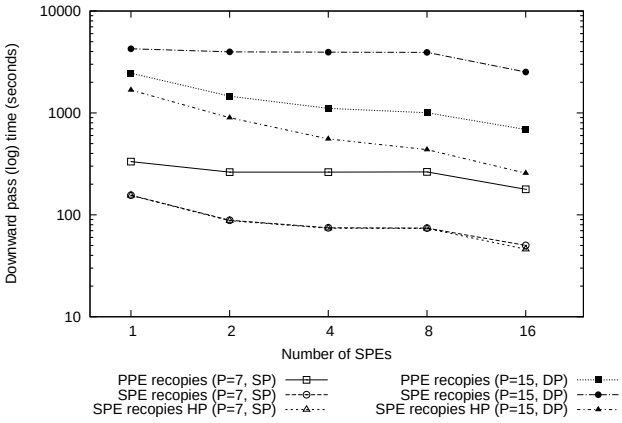
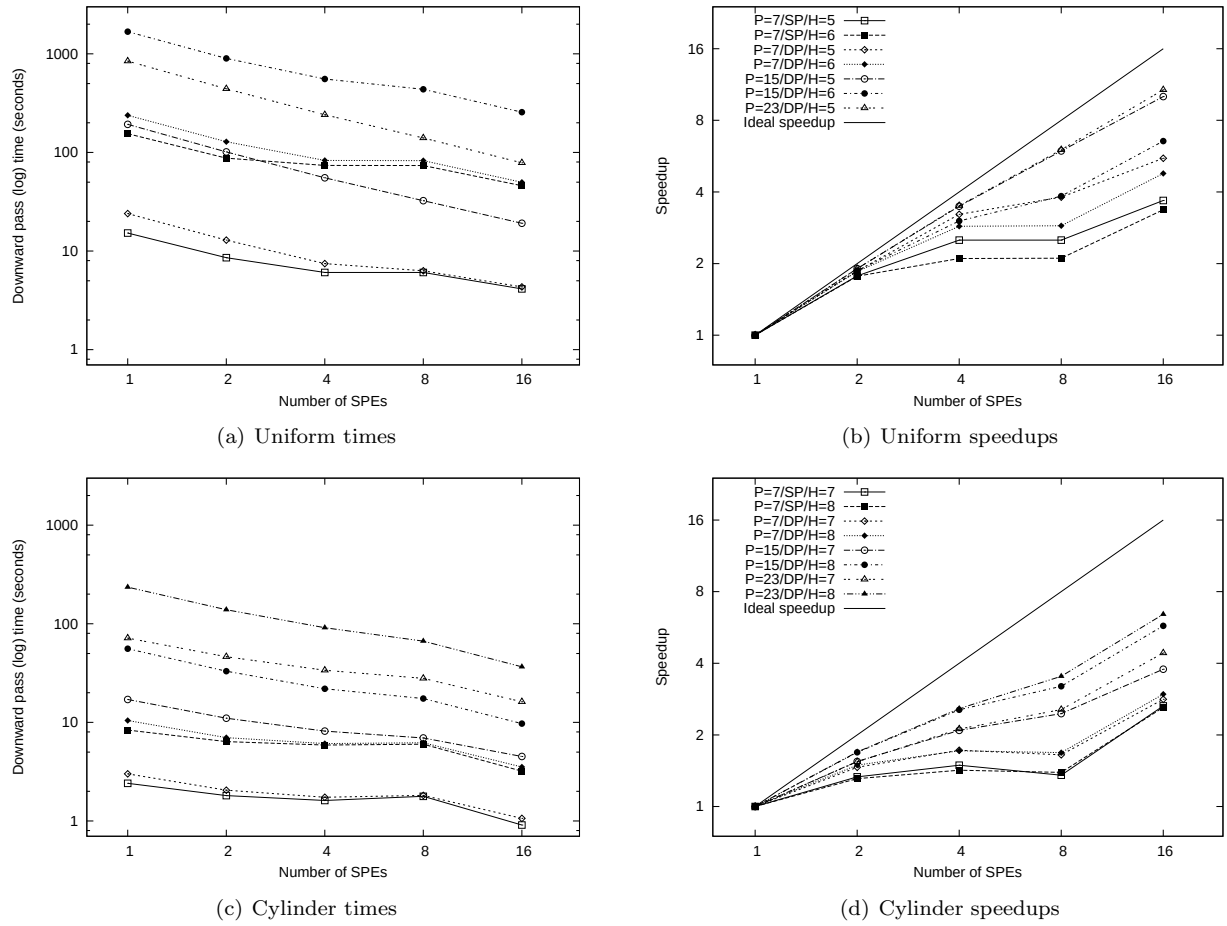


Figure 4: Downward pass times for an uniform distribution (octree height set to 6), with different *recopies* versions, and with or without 16MB memory huge pages (HP)

As shown in Fig. 4, with *recopies* performed on the PPE (*PPE recopies*) we have a performance bottleneck for multiple SPEs within one Cell processor. On the contrary, when the SPE performs directly the *recopies* on the fly in their LS (*SPE recopies*), the single SPE performance is clearly improved since there is no intermediate *recopies* on the PPE, and we obtain better speedups with multiple SPEs since the *recopies* can be performed concurrently by the SPEs. Combining 16MB huge pages with *SPE recopies* can also clearly reduce the computation times when the memory requirements become large (e.g. for $P = 15$ in DP), since we reduce the TLB thrashing (see Sect. 3.3.4). We therefore use *SPE recopies* with huge pages in all the following tests.

It can be noticed that an intermediate scheme where *recopies* are performed by the SPE in the main memory has also been implemented, but this did not improved the performance. Likewise, a single thread PPE code managing all SPEs via DMA writes (similar to the PPE code for the near field computation) has also been implemented and tested, but this delivers similar or lower performance (especially with 16 SPEs).

When considering different octree heights H and P values, in SP and DP, for both uniform and non-uniform distributions, one can see in Fig. 5 that for a given H value the parallel speedup increases with



(a) Uniform times

(b) Uniform speedups

(c) Cylinder times

(d) Cylinder speedups

Figure 5: Downward pass times and speedups on one QS22 blade for both uniform and cylinder distributions with different H and P values

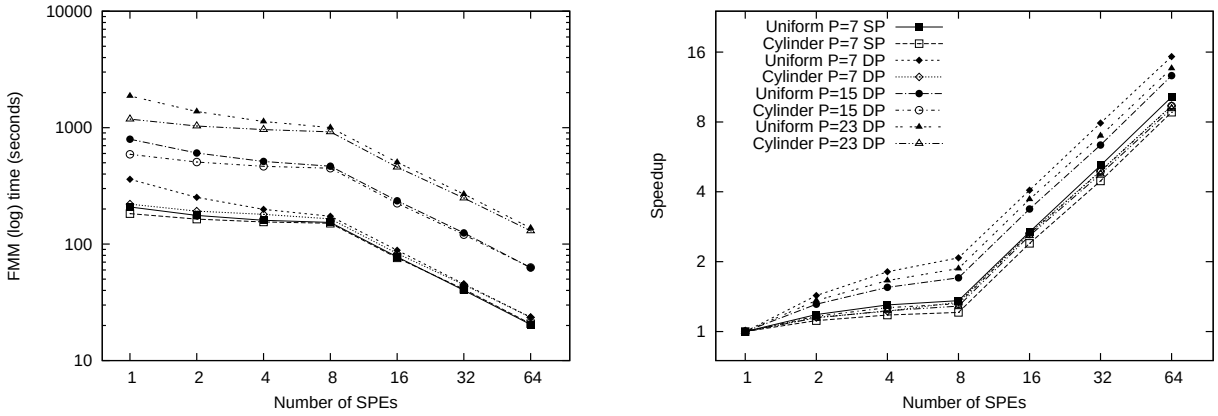
the computation grain (that is to say with DP or when P increases). We obtain reasonable speedups for uniform distributions (up to 10.7 for 16 SPEs), whereas the non-uniform cylinder distributions lead to lower speedups (up to 6.4 for 16 SPEs) since the non-uniformity implies less *M2L* operations (on the SPEs) for each *L2L* operation (on the PPE). Better speedups, or at least lower HP usage, may possibly be obtain with a Linux kernel with 64KB memory pages [25].

4.2 The complete FMM

We now present in Fig. 6, the complete FMM computation times on one and on several QS22 blades, for distributions with $2^{23} = 8$ Mi particles. The octree height H is optimally chosen according to the distribution, the P value, the single or double precision, and the number of SPEs used.

The limited speedups for 2 to 8 SPEs are due to the fact that the most time consuming parts of the FMM (*P2P* and *M2L* operators) have become minority in the overall computation time thanks to their efficient offloading on the SPEs. In addition to *L2L*, the others operators involved in the upward pass (*P2M* and *M2M* operators) and in the evaluation of the local expansions (*L2P* operator) are not offloaded on the SPE and are thus all performed on the PPE single core which leads to a performance bottleneck when increasing the number of SPEs.

With 16 SPEs (two Cell processors on one single QS22 blade), and more (up to 4 blades), we obtain linear speedups for both uniform and non-uniform distributions and for all required precisions. It has to be noticed that we have always used here only one MPI process per blade since this offers better



(a) Times

(b) Speedups

Figure 6: Complete FMM times and speedups on one QS22 blade (up to 16 SPEs) and on four QS22 blades (up to 64 SPEs), for both uniform and cylinder distribution with 8Mi particles

memory scalability, and since using for example two MPI processes per blade (one per Cell processor) does not improve the performance.

4.3 Comparison with multicore CPUs

We finally compare performance and precision between one PowerXCell 8i and two multicore CPUs located at Polytech'Paris-UPMC, Paris, France. The first CPU is a low-cost quad-core Intel Q8200 processor running at 2.33 GHz, released in the same year as the PowerXCell 8i (2008). The second CPU is an high-end hex-core Intel Xeon X5650 processor running at 2.67 GHz, released in 2010. When considering concurrent multiply and add operations, the Q8200 (resp. X5650) has a theoretical peak performance of 74.56 Gflop/s (resp. 128.16 Gflop/s). We use GotoBLAS2[3] as the BLAS library on the CPUs.

The comparison results are presented in Fig. 7 for both uniform and non-uniform distributions and for different values of P. We distinguish the *downward+direct* times and the complete FMM times, as well as serial executions and multi-thread executions. For the multi-thread executions we use all the available cores of each chip: 4 threads for the Q8200, 6 for the X5650 and 8 for the PowerXCell 8i. It can be noticed that the X5650 processor offers a 2-way SMT execution but we only use one thread per core (up to 6 threads) since BLAS routines do not usually take advantage of the SMT capability. The $\mathcal{O}(P^3)$ *M2L* computation scheme with rotations is also presented in Fig. 7 for CPUs as a reference implementation. Following [1, 16, 9], we use the L_2 norm as a measure of error. The L_2 error is here computed with respect to the forces obtained by direct computation (in DP) on 10 000 randomly chosen particles in each distribution.

As far as precision is concerned, one can see for the uniform distribution on Figs. 7(a),7(b),7(e),7(f) (with P values ranging from 3 to 23) that while the DP computations on the PowerXCell 8i always match the DP computations on the CPUs, the SP computation for $P = 7$ on the PowerXCell 8i leads to an L_2 error between $P = 5$ and $P = 7$ on the CPU. This loss of precision on the PowerXCell 8i has two reasons. Firstly the truncation rounding mode of SP operations on the SPE degrades slightly the precision for both *P2P* and *M2L* computations. Secondly, our complex matrix multiplication implies a strong reordering of the computations which further degrades in SP the numerical stability of the *M2L* computations on the SPE. Nevertheless, in some cases computations in SP for $P = 7$ on the PowerXCell 8i offers lower *downward+direct* computation times when compared to $P = 5$ or $P = 3$ on the CPUs. This justifies the use of SP computations on the PowerXCell 8i for such precisions. It can also be noticed that a mixed precision implementation, where the near field part is computed in SP and the far field part in DP, gives much more numerically stable results for $P = 7$ but presents significant loss in precision for $P = 15$.

For the cylinder distribution, the DP computations on the PowerXCell 8i also match the DP computa-

[3]See: http://www.tacc.utexas.edu/tacc-projects/gotoblas2

tions on the CPUs (the computation for $P = 15$ being more precise on the PowerXCell 8i because the optimal octree height is lower than on the CPUs in this case). The SP computations for $P = 7$ are unstable on the CPUs (and thus not presented in Fig. 7), and give an L_2 error greater than $P = 5$ on the PowerXCell 8i, which makes SP computations for $P = 7$ unsuitable for such non-uniform distribution.

As far as performance is concerned, it can be first noticed that the *M2L* rotation scheme has similar performance than the *M2L* BLAS scheme. More precisely, the $\mathcal{O}(P^3)$ rotation scheme delivers slightly lower computation times for the high values of P, whereas the $\mathcal{O}(P^4)$ BLAS computation delivers slightly better performance for the low values of P.

When considering the *downward+direct* times, where *P2P* and *M2L* computations are offloaded on the PPE whereas *L2L* computations are performed on the PPE single core, the deployment on the Cell processor offer significant speedups in most cases. In DP for $P = 15$ with the uniform distribution, the PowerXCell 8i is 3.4 times faster than the Q8200, both released in 2008, and 2.0 times faster than the high-end X5650. More precisely, the offloading of the *P2P* operator in SP enables a speedup of up to 14.7 (respectively 7.8) on the PowerXCell 8i over the Q8200 (resp. X5650). In DP, the speedup becomes 5.1 (resp. 3.5), because of the 128-bit SPU vector unit and since our *P2P* kernel is a little less efficient in DP than in SP. For the downward pass, the speedups are lower because the *L2L* operator is performed on the PPE single core. The PowerXCell 8i is also generally slower than the CPUs for $P = 7$ since the matrices are too small to enable an efficient offloading on the SPE: as presented in Fig. 3 the DMA transfer times dominate for small matrices. $P = 15$ and $P = 23$ enable speedups up to 2.2 over the Q8200 for the uniform distribution. The X5650 is always faster or as fast as the PowerXCell 8i for the downward pass, and both CPUs are also faster on the cylinder distribution.

Besides, when comparing serial and multi-thread executions, one can see that the higher number of cores of the PowerXCell 8i is beneficial for the uniform distribution, but this only partially offsets the low speedups of the downward pass for the cylinder distribution (see Sect. 4.1).

It can also be noticed that while the BLAS routines take advantage of the SSE instructions for the *M2L* computations on the CPUs, the FMB code has not been written with SSE instructions for the *P2P* computations on CPU. Additional performance on CPU could thus be obtained for the direct computation part [11].

When considering the complete FMM times, the PowerXCell 8i is however as fast or slower than the CPUs. This is due to the fact that the upward pass (*P2M* and *M2M* operators) and the evaluation of the local expansions (*L2P* operator) are more than 8 times slower on the PPE than on one CPU core. As a consequence, and because of the efficient offloading of the *P2P* and *M2L* operators on the SPE, the upward pass and the local expansion evaluation become the most time consuming parts of the complete FMM even with only one SPE. When using more SPEs to speedup the *P2P* and *M2L* computations, the *P2M*, *M2M* and *L2P* operators, as well as *L2L*, are still performed on the PPE single core and the decrease in computation time for the complete FMM is thus minority.

5 Conclusion and future work

In this paper we have presented the first deployment of the FMM on the Cell processor, where the most time consuming parts are offloaded on the SPEs. This implementation in the FMB code supports both single and double precision computations, as well as both uniform and non-uniform distributions of particles. This hybrid MPI-thread code also scales efficiently on several Cell blades. We have detailed the efficient SPE offloading of the particle-to-particle operator which take advantage of the mutuality of gravity, and of the multipole-to-local (*M2L*) operator which relies on complex matrix products.

This deployment has not been as direct as initially planned since we had to write our own CGEMM and ZGEMM BLAS routines for complex matrix products on the SPEs, since we have also shown that data movements are best handled directly by the SPEs, and since other changes were necessary in the original FMB code designed for multicore CPU nodes (e.g. the move to a single thread PPE code for the near field part, and the need of 16MB memory huge pages for the far field part).

We obtain significant speedups for the offloaded computations with respect to multicore CPUs released in the same year as the PowerXCell 8i: up to 14.7 for the direct computation part, and up to 3.4 when including also the downward pass of the FMM. Some improvements are still possible: a specific kernel could be used for the computation of the

extra lines in the matrix product on the SPE. The use of 64 KB memory pages may also improve the performance.

However, the main limitation of this work is that the overall performance of our FMM on the Cell processor is bottlenecked by the upward pass and local expansion evaluation which are slowly performed on the PPE single core. Following [8, 32], which have also faced performance bottlenecks due to the PPE, we can foresee better performance for our code on an architecture similar to the *Roadrunner* supercomputer which combines PowerXCell 8i and AMD Opteron dual-core processors on the same node. The upward pass and the evaluation of local expansion could indeed be more efficiently performed on the Opteron cores. Besides, our FMM presents numerical instabilities on the Cell SPEs with medium precisions ($P = 7$) on uniform distributions. We also show that the FMM in single precision is unstable on non-uniform distributions, for both the Cell and the CPUs and for both *M2L* computation schemes with BLAS and rotations.

The main future direction of this work is to deploy efficiently and directly the Fast Multipole Method with BLAS on compute nodes with multiple GPUs and multiple CPU cores. This hybrid deployment could enable us to perform the upward pass, the local expansion evaluation and the small *M2L* matrix products on the CPU cores, whereas the direct computation and the other *M2L* matrix products could be offloaded on the GPU. Such efficient deployment may require algorithmic changes [33], as well as relevant hardware: the concurrent execution of multiple kernels on NVIDIA Fermi GPUs could here be useful to efficiently offloads such medium-grained tasks. Integrated GPU in the AMD Fusion APU (Accelerated Processing Unit) could also be interesting here to accelerate tasks too small to be offloaded on the discrete GPU. All this would also require a dynamic load balancing and task management, that could rely on specific runtimes such as HMPP[4] or StarPU [34].

Acknowledgements

This work was carried out with partial support from HPC@LR, a Competence Center in High-Performance Computing from the Languedoc-Roussillon region, funded by the Languedoc-Roussillon region, the European Union and the *Université Montpellier 2 Sciences et Techniques*. The authors would like to cordially thank the system teams at HPC@LR and at Polytech'Paris-UPMC, as well as B. Cirou at CINES, for helpful assistance during the performance tests.

[4] See: www.caps-entreprise.com/hmpp.html

References

[1] Cheng H, Greengard L, Rokhlin V (1999) A Fast Adaptive Multipole Algorithm in Three Dimensions, *J. Comput. Phys.*, 155:468-498

[2] Dongarra J, Sullivan F (2000) Guest Editors' Introduction: The Top 10 Algorithms, *Comput. Sci. Eng.*, 2(1):22-23

[3] Lashuk I, Chandramowlishwaran A, Langston H, Nguyen TA, Sampath R, Shringarpure A, Vuduc R, Ying L, Zorin D, Biros G (2009) A massively parallel adaptive fast-multipole method on heterogeneous architectures, *SC'09, Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, pp. 58:1-58:12

[4] Arora N, Shringarpure A, Vuduc R (2009) Direct N-body Kernels for Multicore Platforms, *Int. Conf. on Parallel Processing (ICPP)*, pp. 379-387

[5] Knight TJ, Park JY, Ren M, Houston M, Erez M, Fatahalian K, Aiken A, Dally WJ, Hanrahan P (2007) Compilation for Explicitly Managed Memory Hierarchies, *12th ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP'07*, pp. 226-236

[6] De Fabritiis G (2007) Performance of the Cell processor for biomolecular simulations, *Comput. Phys. Commun.*, 176:660-664

[7] Luttmann E, Ensign D, Vaidyanathan V, Houston M, Rimon N, Øland J, Jayachandran G, Friedrichs M, Pande V (2009) Accelerating molecular dynamic simulation on the cell processor and Playstation 3, *J. Comput. Chem.*, 30(2):268-274

[8] Swaminarayan S, Kadau K, Germann TC, Fossum GC (2008) 369 Tflop/s molecular dynamics simulations on the Roadrunner general-purpose heterogeneous supercomputer, *SC'08, Int. Conf. for High Performance Computing, Networking, Storage and Analysis*

[9] Gumerov NA, Duraiswami R (2008) Fast multipole methods on graphics processors, *J. Comput. Phys.*, 227:8290-8313

[10] Yokota R, Bardhan JP, Knepley MG, Barba LA, Hamada T (2011) Biomolecular electrostatics using a fast multipole BEM on up to 512 GPUs and a billion unknowns, *Comput. Phys. Commun.*, 182(6):1272-1283

[11] Chandramowlishwaran A, Williams S, Oliker L, Lashuk I, Biros G, Vuduc R (2010) Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures, *Int. Parallel and Distributed Processing Symposium (IPDPS)*

[12] Hu Q, Gumerov NA, Duraiswami R (2011) Scalable fast multipole methods on distributed heterogeneous architectures, *SC'11, Int. Conf. for High Performance Computing, Networking, Storage, and Analysis*

[13] Hu Q, Gumerov NA, Duraiswami R (2012) Scalable Distributed Fast Multipole Methods, *14th Int. Conf. on High Performance Computing and Communications (HPCC'12)*

[14] Rahimian A, Lashuk I, Veerapaneni S, Chandramowlishwaran A, Malhotra D, Moon L, Sampath R, Shringarpure A, Vetter J, Vuduc R, Zorin D, Biros G (2010) Petascale Direct Numerical Simulation of Blood Flow on 200K Cores and Heterogeneous Architectures, *SC'10, Int. Conf. for High Performance Computing, Networking, Storage and Analysis*

[15] Yokota R, Barba L (2012) Hierarchical N-body Simulations with Autotuning for Heterogeneous Systems, *Comput. Sci. Eng.*, 14(3):30-39

[16] Coulaud O, Fortin P, Roman J (2008) High performance BLAS formulation of the multipole-to-local operator in the Fast Multipole Method, *J. Comput. Phys.*, 227(3):1836-1862

[17] Coulaud O, Fortin P, Roman J (2010) High-performance BLAS formulation of the adaptive Fast Multipole Method, *Math. Comput. Modelling*, 51(3-4):177-188

[18] Takahashi T, Cecka C, Fong W, Darve E (2012) Optimizing the multipole-to-local operator in the fast multipole method for graphical processing units, *Int. J. Numer. Meth. Eng.*, 89(1):105-133

[19] Nyland L, Harris M, Prins J (2007) Fast N-Body Simulation with CUDA, *GPU Gems 3*, pp. 677-695

[20] Fortin P, Lamotte JL (2009) Fast Multipole Method on the Cell Broadband Engine: the Near Field Part, *Selected Papers from the int. Parallel Computing Conf. (ParCo'2009)*, 19:323-330

[21] IBM (2008a) Basic Linear Algebra Subprograms Library Programmer's Guide and API Reference, Software Development Kit for Multicore Acceleration Version 3.1

[22] Bourgerie Q, Fortin P, Lamotte JL (2010) Efficient Complex Matrix Multiplication on the Synergistic Processing Element of the Cell Processor, *Int. Conf. on Cluster Computing, Workshop on Parallel Programming and Applications on Accelerator Clusters (PPAAC'10)*

[23] Coulaud O, Fortin P, Roman J (2007) Hybrid MPI-thread parallelization of the Fast Multipole Method, *6th Int. Symposium on Parallel and Distributed Computing (ISPDC)*, pp. 391-398

[24] Nabors K, Korsmeyer FT, Leighton FT, White J (1994) Preconditioned, Adaptive, Multipole-Accelerated Iterative Methods for Three-Dimensional First-Kind Integral Equations of Potential Theory, *SIAM J. Sci. Comput.*, 15(3):713-735

[25] Arevalo A, Matinata RM, Pandian M, Peri E, Ruby K, Thomas F, Almond C (2008) Programming the Cell Broadband Engine Architecture, Examples and Best Practices, *IBM Redbook* SG24-7575

[26] IBM (2008b) Cell Broadband Engine Programming Handbook, Including the PowerXCell 8i Processor, Version 1.11

[27] Kurzak J, Dongarra J (2007) Implementation of mixed precision in solving systems of linear equations on the Cell processor, *Concurr. Comput.: Pract. Exper.*, 19(10):1371-1385

[28] Williams SW, Shalf J, Oliker L, Husbands P, Yelick K (2005) Dense and Sparse Matrix Operations on the Cell Processor. Lawrence Berkeley National Laboratory: LBNL Paper LBNL-58253

[29] Kurzak J, Alvaro W, Dongarra J (2009) Optimizing Matrix Multiplication for a Short-Vector SIMD Architecture - CELL Processor, *Parallel Comput.*, 35(3):138-150

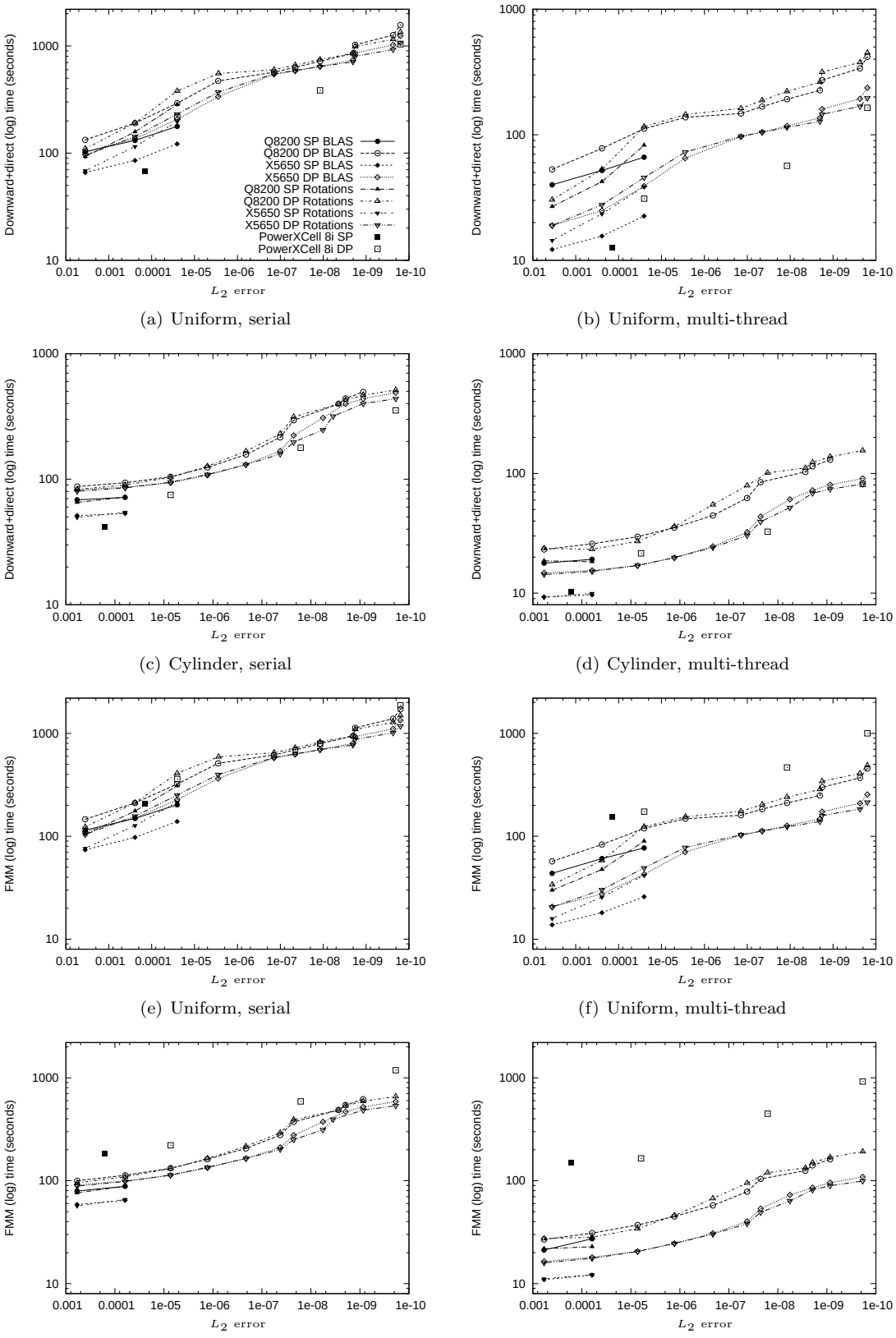
[30] Kurzak J, Buttari A, Dongarra J (2008) Solving Systems of Linear Equations on the CELL Processor Using Cholesky Factorization, *IEEE T. Parall. Distr.*, 19(9):1175-1186

[31] Kistler M, Gunnels J, Brokenshire D, Benton B (2009b) Petascale computing with accelerators, *14th ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP'09*, pp. 241-250

[32] Kistler M, Gunnels J, Brokenshire, D, Benton, B (2009a) Programming the Linpack benchmark for the IBM PowerXCell 8i processor, *Scientific Programming*, 17(1-2):43-57

[33] Hamada T, Narumi T, Yokota R, Yasuoka K, Nitadori K, Taiji M (2009) 42 TFlops hierarchical N-body simulations on GPUs with applications in both astrophysics and turbulence, *SC'09, Int. Conf. for High Performance Computing, Networking, Storage, and Analysis*, pp. 62:1-62:12

[34] Augonnet C, Thibault S, Namyst R, Wacrenier PA (2011) StarPU: a unified platform for task scheduling on heterogeneous multicore architectures, *Concurr. Comput.: Pract. Exper.*, 23(2):87-198



(a) Uniform, serial

(b) Uniform, multi-thread

(c) Cylinder, serial

(d) Cylinder, multi-thread

(e) Uniform, serial

(f) Uniform, multi-thread

(g) Cylinder, serial

(h) Cylinder, multi-thread

Figure 7: Comparison between one PowerXCell 8i, one low-end Q8200 CPU and one high-end X5650 CPU for uniform and cylinder distributions of 8Mi particles. We use $P = 7$ in SP and $P \in \{7, 15, 23\}$ on the PowerXCell 8i. On CPU we use $P \in \{3, 5, 7\}$ in SP and $P \in \{3, 5, 7, \ldots, 21, 23\}$ in DP. Results on the Q8200 in DP for the cylinder distribution and $P = 23$ with BLAS are not presented here since they imply swapping on the 8 GB of memory available with the Q8200