Portable Node-Level Performance Optimization for the Fast Multipole Method

Andreas Beckmann and Ivo Kabadshow

Abstract This article provides an in-depth analysis and high-level C++ optimization strategies for the most time-consuming kernels of a Fast Multipole Method (FMM). The two main kernels of a Coulomb FMM are formulated to support different hardware features, such as unrolling, vectorization or threading without the need to rewrite the kernels in intrinsics or even assembly. The abstract description of the algorithm automatically allows optimal node-level peak performance on a broad class of available hardware platforms. Most of the presented optimization schemes allow a generic, hence platform-independent description for other kernels as well.

1 Introduction

Classical simulations in the field of molecular dynamics or astrophysics [8] are mostly constrained by the number of long-range particle interactions. A straightforward computation is limited to small particle numbers due to the $O(N^2)$ scaling. Fast summation methods such as [1, 3, 5] are capable of reducing the complexity to $O(N \log N)$ or even $O(N)$. The linear scaling FMM allows to solve the N-body problem for any user-provided precision, and hence can replace the classical computation for larger particle systems. The development of parallel computers in the last decades accelerated this development further. Now simulation codes can handle multi-billion particle simulations [7] easily. However, every fast summation method has auxiliary parameters, data structures and memory requirements that need to be supplied. Additionally, the overall performance of such algorithms on the node-level strongly depends on the available features of the underlying hardware architecture and the ability of the implementation to make efficient use of these features.

In this paper we describe and evaluate a basic threading and vectorization approach for two exemplary FMM kernels. Additionally, we explore less known

A. Beckmann (✉) • I. Kabadshow
Forschungszentrum Jülich GmbH, Jülich Supercomputing Centre (JSC), 52425 Jülich, Germany
e-mail: a.beckmann@fz-juelich.de; i.kabadshow@fz-juelich.de

M. Mehl et al. (eds.), *Recent Trends in Computational Engineering - CE2014*,
Lecture Notes in Computational Science and Engineering 105,
DOI 10.1007/978-3-319-22997-3_2

strategies to leverage even more single-core performance. The focus of our analysis lies on a fast, but still portable implementation.

2 The Fast Multipole Method in a Very Small Nutshell

The Fast Multipole Method allows to compute the pairwise interactions of N particles with charge or mass q_i in $O(N)$ complexity. The Coulomb force $\mathbf{F}$ at position $\mathbf{r}_i$ due to a charge q_j at position $\mathbf{r}_j$ with distance $|\mathbf{r}_{ij}|$ for a system with open boundary conditions can be written as

$$\mathbf{F}(\mathbf{r}_i) = q_i \sum_{j=1}^{N} \frac{q_j}{|\mathbf{r}_{ij}|^3}\mathbf{r}_{ij} \quad (i \neq j)\,. \tag{1}$$

The inverse distance $1/|\mathbf{r}_{ij}|$ can be factorized by a bipolar expansion [9] into three individual parts: a multipole expansion $\omega(\mathbf{a})$ around a first coordinate system's center $\mathbf{O}_1$, a multipole expansion $\omega(\mathbf{b})$ around a second coordinate system's center $\mathbf{O}_2$ and an operator $B(\mathbf{R})$. The vector $\mathbf{R}$ is defined by $\mathbf{O}_2 - \mathbf{O}_1$. An M2L operation translates multipole expansions ω into local expansions μ via B. The force of the expanded coordinates reads

$$\mathbf{F}(\mathbf{a}) \approx \sum_{l=0}^{p}\sum_{j=0}^{p}\sum_{m=-l}^{l}\sum_{k=-j}^{j}(-1)^j \nabla\omega_{lm}(\mathbf{a})B_{j+l,m+k}(\mathbf{R})\omega_{jk}(\mathbf{b}) \tag{2}$$

$$\approx \sum_{j=0}^{p}\sum_{m=-l}^{l}(-1)^j \nabla\omega_{lm}(\mathbf{a})\mu_{lm}\,. \tag{3}$$

However, the convergence condition of Eq. (2) demands $|\mathbf{a}| + |\mathbf{b}| < |\mathbf{R}|$. To avoid convergence issues, the FMM tree must be subdivided until the condition is met, or interactions violating the convergence condition have to be performed directly in the so called near field. Thus, in general the computation of the Coulomb force is split up into a far field (FF) part, where the interactions take place over the representation as expansions and a near field (NF) part, where the interactions are computed classically. The total force on a particle can be summed as

$$\mathbf{F}_C(\mathbf{r}_i) = \mathbf{F}_{NF}(\mathbf{r}_i) + \mathbf{F}_{FF}(\mathbf{r}_i)\,. \tag{4}$$

The FMM does require three parameters (see Fig. 1), the depth of the octree d, the expansion order p and the separation criterion ws, dividing the workload between near field and far field.

For the sake of simplicity, we will not introduce FMM theory in more detail and refer the reader to [6, 9]. In this paper we do not require more insights into the

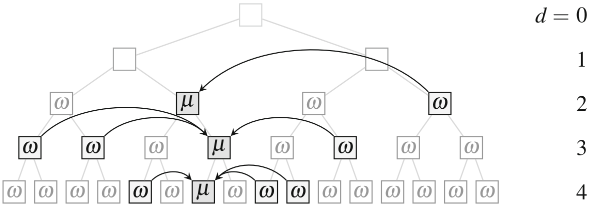


Fig. 1 Illustration of the FMM with a binary tree and depth $d = 4$. The depicted multipole expansions ω at each node on each tree level are translated via $M2L$ operations (*black arrows* = B operators) into local expansions μ. Only nodes with non-separated parent nodes are taken into account. The separation criterion $ws = 1$ requires to skip direct neighbors for such a translation

method, since we are not interested in the algorithm itself, but its kernel optimization on different hardware. Equations (1) and (2) account for 95 % of the runtime of the FMM.

In general, floating point numerics are usually not associative due to the limited precision. However, numerical stability of the FMM does not depend on using (or avoiding) a certain operation order. Thus, the summation order can be changed to allow for more optimization capabilities.

3 Hardware Features and Node-Level Performance

Recent processors (and accelerators) come with a variety of features that should and can be exploited to maximize performance in time-critical parts of applications: multi-core (and many-core) CPUs, SMT (simultaneous multithreading), SIMD vectorizations, ILP (instruction-level parallelism), out-of-order execution (or lack thereof), large register sets, cache hierarchies, NUMA (non-uniform memory access), etc. To ease application development and adaptation to new platforms, these features ideally need to be accessed through some abstraction layer. To identify the time-critical computational kernels, profiling tools are a convenient way to perform that task. Evaluating a kernel to find its current limiting factor as well as unused resources requires much manual work, as well as in-depth knowledge of the architecture and the algorithm. Performance counters that are provided by the hardware can be utilized, e.g., by using the PAPI [2] library. Additionally, an inspection at assembly level is beneficial, if the kernels consist of only a few instructions and can be surveyed easily. For our Coulomb FMM, these terms apply, since the kernels reveal about 40 instructions for their innermost loops. Once a kernel fully uses some resource (port saturation, bandwidth, etc.) it is considered as fully optimized.

Our benchmarks were performed on three systems at Jülich Supercomputing Centre. The relevant hardware parameters are listed in Table 1.

Table 1 Systems and configurations used for performance tests

	Ivy Bridge	Haswell	Blue Gene/Q
Vendor	Intel	Intel	IBM
CPU	Core i7-3770	Xeon E5-2695 v3	PowerPC A2
Cores	4	14	16
Hardware threads (SMT)	2	2	4
Out-of-order execution	Yes	Yes	No
Frequency			
Scalar (GHz)	3.4	3.3–2.8	1.6
SIMD (GHz)	3.4	3.0–2.6	1.6
Cache			
L1 Data (L1D)	32 KB	32 KB	16 KB
L1 Instruction (L1I)	32 KB	32 KB	16 KB
L2	256 KB	256 KB	32 MB
L3	8 MB	35 MB	—
SIMD			
FP registers	16	16	32
`float` ×4	SSE	SSE	QPX
`double` ×2	SSE	SSE	—
`float` ×8	AVX	AVX	—
`double` ×4	AVX	AVX	QPX
FMA	No	Yes	Yes
Compiler	GCC 4.8	GCC 4.9	Clang for BG/Q

4 The Near Field Kernel

We will start designing and optimizing an $O(N^2)$ Coulomb kernel for single-core performance. The aim is to maximize *throughput*—to finish all interactions as fast as possible—and not minimizing the *latency* for computing a single interaction. Therefore we design the following algorithm to leverage SIMD and ILP.

4.1 Designing a Generic SIMD-Capable Kernel

Since the computational work done in the outer loops of the kernel is negligible, we will start by designing and analyzing a generic kernel for the innermost loop in some loop nest. The loop structure for a straightforward implementation of, e.g., Eq. (1) could look like this (in C++-like pseudo-code, indentation defines scopes):

```
for (i = 0; i < n; ++i)
  for (j = 0; j < n; ++j)
    if (i != j)
      interact(i, j);
```

A first step towards an optimized variant would be the removal of the if-clause. The conditional expression can be eliminated by splitting the inner loop:

```
for (i = 0; i < n; ++i)
  for (j = 0; j < i; ++j)
    interact(i, j);
  for (j = i + 1; j < n; ++j)
    interact(i, j);
```

To avoid code duplication, we move the inner loop to a subroutine that will be subject to further inspection later on. For now, we give this subroutine the generic name `interaction_sequence(i, j_begin, j_end)` and use it as follows:

```
for (i = 0; i < n; ++i)
  interaction_sequence(i, 0, i);
  interaction_sequence(i, i + 1, n);
```

The physical properties that need to be computed in `interaction_sequence(i, j_begin, j_end)` are the potential and the force at position $\mathbf{x}_i$ contributed by the range of particles $[j_{\text{begin}}, \ldots, j_{\text{end}})$:

$$\Phi_i = \Phi_i + \sum_{j=j_{\text{begin}}}^{j_{\text{end}}-1} \frac{q_j}{|\mathbf{x}_i - \mathbf{x}_j|} \quad (i \neq j) \tag{5}$$

$$\mathbf{F}_i = \mathbf{F}_i + q_i \sum_{j=j_{\text{begin}}}^{j_{\text{end}}-1} \frac{q_j}{|\mathbf{x}_i - \mathbf{x}_j|^3} (\mathbf{x}_i - \mathbf{x}_j) \quad (i \neq j)\,. \tag{6}$$

Computing these two properties together is advisable since they can share many intermediate results. The force computation contains a multiplication with the local q_i that has been factored out and moved into the outer loop for efficiency. We expect to use registers for the position $\mathbf{x}_i$ as well as (temporary) accumulators for the results.

A generic templated SIMD-capable implementation of an algorithmic kernel is provided for scalar operations by committing scalar types (i.e. SIMD-width = 1) instead of actual SIMD types (SIMD-width > 1) as template parameters. We expect that the compiler optimizes away any provisions existing in the generic implementation that are not needed for operating on scalar types and therefore the performance does not differ from an equivalent purely scalar implementation. For the sake of simplicity, the generic SIMD implementation (see Algorithm 1) shall only be called for a range of particles that is a multiple of the SIMD-width. The remaining particles could be processed by a call to the same generic routine, but with scalar type parameters.

```
template <typename Scalar, typename Real>
void potential_and_force_sequence(
  // input:
  Scalar * x, Scalar * y, Scalar * z, Scalar * q,  // particles, AoS
  Real x_i_x, Real x_i_y, Real x_i_z,  // particle i
  int j_begin, int j_end,  // particle range
  // output (accumulators):
  Real &F_i_x, Real &F_i_y, Real &F_i_z, Real &Phi_i)
  {
    for (int j = j_begin; j < j_end; j += SIMD_traits<Real>::width)
    {
      Real x_j_x = load<Real>(x + j);
      Real x_j_y = load<Real>(y + j);
      Real x_j_z = load<Real>(z + j);
      Real q_j   = load<Real>(q + j);
      Real dx    = x_i_x - x_j_x;
      Real dy    = x_i_y - x_j_y;
      Real dz    = x_i_z - x_j_z;
      Real rlen      = rsqrt(dx * dx + dy * dy + dz * dz);
      Real q_j_rlen  = q_j * rlen;
      Real rlen2     = rlen * rlen;
      Real q_j_rlen3 = q_j_rlen * rlen2;
      F_i_x += dx * q_j_rlen3;
      F_i_y += dy * q_j_rlen3;
      F_i_z += dz * q_j_rlen3;
      Phi_i += q_j_rlen;
    }
  }
```

Alg. 1 Type-generic implementation of the inner loop for an $O(N^2)$ Coulomb solver. `Real` can be an arbitrary scalar or SIMD type, `Scalar` is the corresponding basic scalar type.

Assuming that operators for basic arithmetic operations (`+`, `-`, `*`, `/`, `+=`, `-=`) are overloaded and template specializations for `load<Real>()`, `rsqrt<Real>()`, and `SIMD_traits<Real>` are provided for the `Real` types to be used, this implementation can be used for a wide range of platforms and their scalar and SIMD types. Additionally, the design is not limited to `Real` types building on the scalar types `float` and `double`. The generic implementation can be used with, e.g., types from the boost multiprecision library as well.

Using SIMD-enabled algorithms, we target throughput computing and therefore apply SIMD operations over the "larger" of the available quantities, e.g., particles (N) instead of dimensions of the input (only three).

The compiler-generated assembly targeting the "Ivy Bridge" machine is listed in Algorithm 2. For other data types and SIMD variants, the structure including instruction class and count does not change. Just the data type specific parts will be adjusted, e.g., data type suffix, register class, size increment.

The assembly instructions can be easily mapped to the source code shown in Algorithm 1 and do not exhibit obvious inefficiencies.

```
loop:
       vsubpd 0x0(%r13,%rax,4),%ymm12,%ymm7
       vsubpd (%r14,%rax,4),%ymm13,%ymm6
       vsubpd (%r15,%rax,4),%ymm14,%ymm5
       vmulpd %ymm7,%ymm7,%ymm8
       vmulpd %ymm6,%ymm6,%ymm0
       vaddpd %ymm0,%ymm8,%ymm0
       vmulpd %ymm5,%ymm5,%ymm8
       vaddpd %ymm8,%ymm0,%ymm0
       vsqrtpd %ymm0,%ymm0
       vdivpd %ymm0,%ymm9,%ymm0
       vmulpd (%rdx,%rax,4),%ymm0,%ymm8
       add    $0x4,%rax
       cmp    %rax,%rbx
       vmulpd %ymm0,%ymm0,%ymm0
       vmulpd %ymm0,%ymm8,%ymm0
       vaddpd %ymm8,%ymm1,%ymm1
       vmulpd %ymm0,%ymm7,%ymm7
       vmulpd %ymm0,%ymm6,%ymm6
       vmulpd %ymm0,%ymm5,%ymm0
       vaddpd %ymm7,%ymm3,%ymm3
       vaddpd %ymm6,%ymm2,%ymm2
       vaddpd %ymm0,%ymm4,%ymm4
       ja     loop
```

Alg. 2 Compiler-generated assembly code (platform: x86_64, target CPU: Intel Ivy Bridge, SIMD: AVX, basic data type: `double`, compiler: GCC 4.8) for the loop in Algorithm 1.

4.2 Performance Analysis

For low-level performance measurements, we use the PAPI library in conjunction with the provided hardware counters. Initially, we are interested in counting instructions and cycles for a single iteration of the inner loop of the near field. However, since the overhead for reading the performance counters is not negligible, we measure a full $O(N^2)$ kernel run for $N = 114\,537$ particles and amortize over the number of inner loop iterations $O(N^2)$. The performance measured for the generic near field kernel on the "Ivy Bridge" machine is listed in Table 2.

From the disassembly listing (Algorithm 2) we can build a dependency graph (Fig. 2) and retrieve some statistics: There are 20 floating point instructions (9 ADD/SUB, 9 MUL, 1 DIV, 1 SQRT) and three instructions for maintaining the loop (increment, compare, branch). Four instructions are reading from memory (floating point arithmetic with memory source operand) and there are no instructions writing to memory. Within the loop, six integer registers out of 16 and 13 floating point registers out of 16 are used.

Table 2 Performance of the generic P2P kernel (Algorithm 1) using different scalar and SIMD types (SSE: 128-bit registers, AVX: 256-bit registers) on an Intel Core i7-3770 (Ivy Bridge) CPU. Only one core is being utilized

Basic type	`float`			`double`		
SIMD	None	SSE	AVX	None	SSE	AVX
SIMD-width	1	4	8	1	2	4
Total time [s]	68	17	14	109	55	54
Speedup	1	3.99	4.86	1	1.98	2.02
Inner loop						
Instructions	25.00	24.01	23.03	25.00	23.50	23.01
Cycles	18.58	17.78	29.27	28.68	28.61	56.21
Bytes read	16	64	128	32	64	128

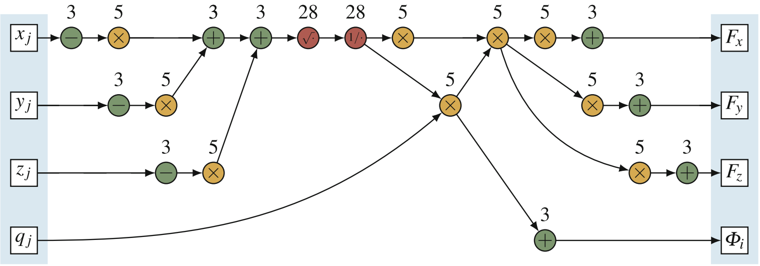


Fig. 2 Data dependencies within a single iteration of the P2P kernel, according to the assembly in Algorithm 2. One iteration computes one interaction, namely the forces $\mathbf{F}(\mathbf{x}_i)$ and the potential $\Phi_i(\mathbf{x}_i)$ from a particle at $\mathbf{x}_j$. The number stacked on top of each circle represents the latency of this operation

4.3 Critical Path Analysis

In order to reduce the runtime of our near field kernel, we need to perform a detailed critical path analysis. The goal of such an analysis is to find the operations that contribute to the exposed runtime. Our testcase consists of double precision instructions on an AVX(4) SIMD-register (Table 3). We identified the following details:

90	Cycles	Lower bound on the latency
9	Cycles	Instructions being issued on the FP adder unit
11	Cycles	Instructions being issued on the FP multiplier unit
56	Cycles	Blocking the divider unit
4–8	Cycles	Load units active
10	Instructions	Longest sequence of dependent instructions

Table 3 Lower bounds for instruction latency and reciprocal throughput for the floating point instructions in CPU clock cycles, taken from [4]. Latency (lat.) is the time (in cycles) needed until the result of an instruction is available and can be used in a subsequent operation. Reciprocal throughput (r.t.) is the time (in cycles) until the next instruction can be issued to the same functional unit. Fully pipelined instructions (reciprocal throughput 1) like ADD or MUL can be issued in each cycle since they use different functional units

Basic type	`float`						`double`					
SIMD	none		SSE		AVX		None		SSE		AVX	
SIMD-width	1		4		8		1		2		4	
	lat.	r.t.	lat.	r.t.	lat.	r.t.	lat.	r.t.	lat.	r.t.	lat.	r.t.
ADD/SUB	3	1	3	1	3	1	3	1	3	1	3	1
MUL	5	1	5	1	5	1	5	1	5	1	5	1
DIV/SQRT	7	7	7	7	14	14	14	14	14	14	28	28

For double precision computation the limiting factor is clearly the blocking of the FPU divider unit in the CPU. But when do the other instructions get executed if already two (SQRT and DIV) account for the runtime of 56 cycles?

Out-of-order execution (OoOE) as well as speculative execution of branches allow the hardware to do "loop unrolling" for a small window of instructions. Additions and multiplications from the preceding and subsequent iterations will be executed in parallel to the limiting instructions SQRT and DIV utilizing the reorder buffer (120 registers). OoOE is a costly hardware addition both in transistors numbers (die space) and runtime energy consumption, but helps to efficiently hide the overhead from execution pipelines with unbalanced latencies.

Also note that there is nearly no speedup when switching from SSE to AVX SIMD mode since the latency and reciprocal throughput for the critical SQRT and DIV instructions also grows by a factor of two for AVX.

4.4 Solving the SQRT+DIV Bottleneck

The near field computations of the inverse distance between two particles requires SQRT (square root) and DIV (divide) instructions which have long latencies and cannot be pipelined. Several hardware platforms provide a low precision bypass to alleviate the bottleneck. The original instruction can be dropped and replaced by an estimate for a combined reciprocal square root estimate (RSQRTE) which has to be followed by one (or more) Newton-Raphson (NR) iteration(s) to increase the accuracy as needed. On x86_64 the estimate is only available for single precision and provides 11 bits of accuracy which will be doubled for each NR step. The lack of this estimate for double precision can be damped by adding two additional instructions for converting to single precision and back. Each NR iteration costs four additional multiplications and one subtraction (plus some data dependencies). On CPUs with fused-multiply-add (FMA) instruction sets, most ADD/SUB instructions

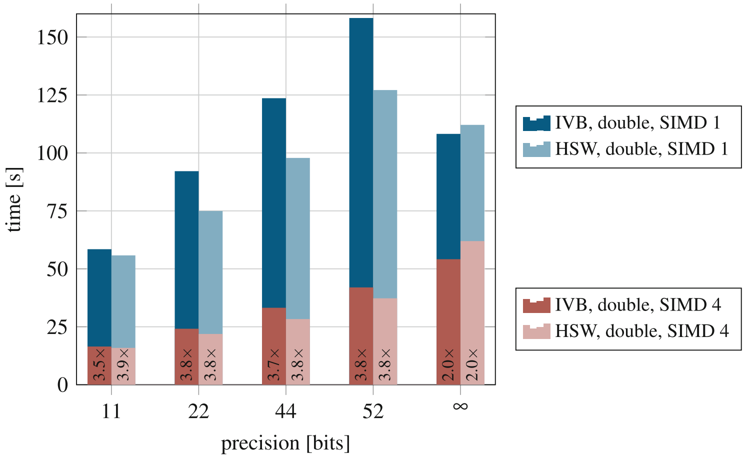


Fig. 3 Runtime for RSQRT estimates + NR in double precision on a Haswell CPU (HSW, 3.3 GHz) with FMA vs. an Ivy Bridge CPU (IVB, 3.4 GHz) without FMA. Speedup for SIMD operations (`double` ×4, *red*) over scalar operations (*blue*). Precision ∞ indicates usage of IEEE 754 conform $1/\sqrt{\cdot}$

can be merged with a preceding MUL to reduce the overall number of instructions. This benefits the NR iterations as well.

Figure 3 shows runtimes and speedups achievable when switching from exact inverse square root to estimate-based instructions with reduced precision. The precision of these estimates varies significantly between platforms, but utilizing such an estimate also adds the flexibility to do fewer (or even none) NR steps in case the required accuracy is less than machine precision.

4.5 Multi-threaded Implementation

To utilize all cores of a node we implemented and tested the following threading approaches

- pThreads,
- `std::thread`,
- `boost::thread`, and
- OpenMP.

All schemes worked well with a parallelization where the iterations of the outer loop were split evenly by the number of cores and each chunk was processed by a separate thread. No changes had to be made for the inner loop. The obtained efficiency for scaling on a single node can be seen in Fig. 4.

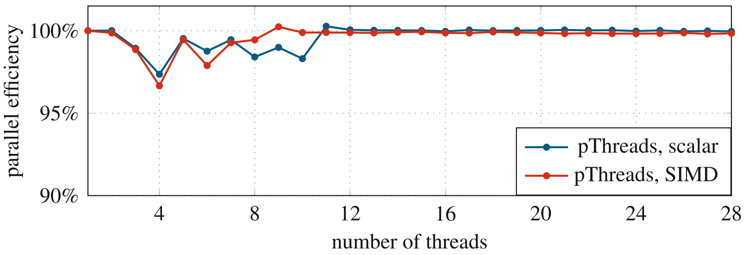


Fig. 4 Efficiency for scaling an $O(N^2)$ Coulomb kernel on a dual Xeon E5-2695 v3 CPU (2 × 14 cores). The plot has been normalized to the maximum turbo boost core frequency which varies with the number of active cores. Scheduling was left to the operating system; no explicit pinning to particular cores was employed

Since each thread is already capable of fully utilizing a CPU core with the instructions on the critical path there is no additional benefit from using hyperthreading by overcommitting the number of threads.

4.6 The Near Field Kernel on Blue Gene/Q

The SIMD capabilities on a BG/Q can be easily exploited by processing several particles concurrently, ideally using a Struct-of-Arrays (SoA) format for the particle information. The Blue Gene/Q architecture offers a (vectorized) RSQRTE estimate (with 14 bits precision), the SQRT instruction unfortunately is only available for scalar operation.

The time for computing a force and potential contribution for a pair of particles is largely dominated by the dependencies of instructions on the critical path. To increase the number of instructions available for dispatch and fill unused slots, an out-of-order (OoO) architecture will (speculatively) issue instructions from the next iteration(s) while waiting for the dependencies that block progress in the current iteration. Since the data dependencies between loop iterations are minimal (only the accumulator at the end is shared) this is easily handled by e.g. Intel CPUs.

For in-order platforms such as Blue Gene/Q, this has to be realized by the compiler (ideally) or the software engineer. Loop unrolling can be applied to interweave two (or more) subsequent iterations, but scheduling the registers and instructions for the interleaved iterations may be a nontrivial task. Unfortunately, the compilers cannot do this optimally. To fill the unused issue slots, the number of instructions available per loop iteration can be increased by aggressive loop unrolling.

Threads are used to load the individual cores with work. The execution model on the Blue Gene/Q cores demands the use of SMT to leverage ILP which can easily be achieved using thread oversubscription.

Due to compiler insufficiencies we implemented a manually interleaved and scheduled SIMD-inline-assembly kernel. Running this kernel in four-way SMT on one core issues $4 \times 21 = 84$ floating point instructions over 86.8 cycles which is close to the limit of issuing 1 FP op per cycle. This approach should only be the exception and was purely motivated by compiler insufficiencies.

5 The Far Field Kernels

We focus on a slightly simplified version of the most frequent (and time consuming) computational kernel executed during the far field computation, defined as

$$\mu_{lm} = \sum_{l=0}^{p} \sum_{m=-l}^{l} \sum_{j=0}^{p} \sum_{k=-j}^{j} B_{j+l,k+m} \cdot \omega_{jk} \qquad (7)$$

with an implementation as shown in Algorithm 3.

This kernel is also representative for several other similarly structured kernels with the same complexity of $O(p^4)$. Variables ω and μ are triangular arrays containing complex numbers at elements $(l,m) : 0 \leq l \leq p, |m| \leq l$. The operator B has identical structure, but extends up to $2 \cdot p$.

The access pattern from Eq. (7) suggests to use a column-major memory layout for the data structures. The loops can be interchanged arbitrarily and partial results can be temporarily accumulated in registers, since the summation order does not affect numeric stability. Reordering the loops allows to trade cache locality between the three data structures. For details see Table 4.

```
for (l = 0; l <= p; ++l)
  for (m = -l; m <= l; ++m)
    for (j = 0; j <= p; ++j)
      for (k = -j; k <= j; ++k)
        mu[l, m] += B[j + l, k + m] * omega[j, k]
```

Alg. 3 Slightly simplified far field kernel in $O(p^4)$ complexity. The innermost loop iteration consists of only one complex multiplication $B_{j+l,k+m} \cdot \omega_{jk}$ and a single accumulation into μ_{lm}.

Table 4 Trade-off for value reuse in subsequent iterations for read/write operations (higher is better) and minimal cache requirement (lower is better). There are $O(p^4)$ iterations per kernel call in total

Loop order	ω (read)	B (read)	μ (accumulate)	Cache requirement
`l-m-j-k`	1	1	$O(p^2)$	$O(p^2)$
`j-k-l-m`	$O(p^2)$	1	1	$O(p^2)$
`l-j-m-k`	1	1	$O(p)$	$O(p)$

Now, let's take a closer look at some possible access patterns.

The `l-m-j-k` order (as shown in Algorithm 3) can perform $O(p^2)$ subsequent accumulations for the same element μ_{lm} in registers followed by a single memory update. But reuse of elements from ω and B only happens after $O(p^2)$ steps. This is a good choice for small p where $O(p^2)$ elements fit in cache.

The `j-k-l-m` order gains optimal p^2 reuse of each element from ω at the cost of not caching any memory update (accumulate) for the results μ in registers.

The `l-j-m-k` order splits the computation of each element μ_{lm} into $O(p)$ phases with one memory update per phase, but improves the locality and reuse in ω and B to a set of only $O(p)$ hot elements. This is a better choice for larger p values.

To combine the advantages of the `l-m-j-k` and the `l-j-m-k` orderings, a hybrid scheme can be used. A splitter value l' cuts the loops over l and j into two parts. The `l-m-j-k` ordering will be carried out for $l, j \leq l'$. This results in a reduction of memory updates while only a small amount of cache is occupied. The remaining computations are performed with the more cache-efficient `l-j-m-k` ordering. The choice of l' depends on the platform (L1D cache size) and data type and aims to minimize the total runtime.

For small values of p, the control structures (loop counters, offset computation, tests, branches, ...) for the four nested loops may significantly outweigh the actual computation. Using C++-lambdas and templates, we generate compiled code that is free of conditionals and branches for several small values of p without duplicating the source code. This, however, easily blows up code size and exceeds the instruction cache size incurring significant penalties due to unavoidable cache misses for larger p.

The schemes mentioned above show best performance only for a subset of certain p. Therefore, we combined all the ideas into a composite scheme that selects the fastest algorithm depending on the number of poles p and should provide the best performance over the full range of p. Such a mapping of p to different algorithms can either be a static assignment or based on runtime benchmarking and varies between architectures.

5.1 Parallelization of M2L Operations

A typical FMM run with millions of particles N, tree depth $d = 6$, and separation criterion ws $= 1$ needs more than $45 \cdot 10^6$ M2L evaluations. Neglecting all but the lowest level in the tree (which clearly dominates the work needed on the higher levels), this kernel uses $8^d = 262{,}144$ distinct triangular array data structures for each of ω and μ as well as hundreds of distinct operator matrices for B. Such workload and data structures can easily be distributed over multiple nodes or threads. Optimizing performance on a single core, we especially need to leverage SIMD and ILP to maximize throughput for these operations.

One possibility for applying SIMD operations is the vectorization of the innermost loop. Another one would be the vectorization over several independent evaluations of the kernel. The latter approach requires reorganization of the memory layout to allow for efficient SIMD memory access. The data structures passed as parameters to previously independent kernel calls must be interleaved element-wise. Unfortunately such a layout is only reusable for one or very few vectorized kernel calls before requiring differently interleaved data structures. Hence, we will focus on a general vectorization within the kernel.

5.2 Memory Layout for Vectorized M2L Operations

The innermost loop operates on two equally sized windows into the linearized representation of ω and B. Pairwise complex multiplications are performed on the array elements and the results are accumulated into a single complex number in a register. The size and placement of these windows are defined by the three outer loops and each pair of elements from the two arrays is multiplied at most once. The window start addresses into B are moving element-wise (i.e. less than SIMD-width), preventing the exclusive use of aligned loads. With respect to cache efficiency, using replication and padding will not be efficient. Also a decision to use explicit shuffling may be more expensive than the penalties for unaligned loads.

There exist several possibilities for storing arrays of complex numbers. While this does not impact purely scalar operations, it is essential for vectorization. The classical implementation is defined as an array of `struct complex { Real re, im; };`. Loading a SIMD-register from a so called Array-of-Structs (AoS) yields alternating real and imaginary parts: R I R I. This requires some additional shuffling instructions within the CPU to separate the real and imaginary parts. An alternative representation uses two separate arrays storing only either the real or the imaginary parts, so called Struct-of-Arrays (SoA). SIMD loads from SoAs would fill a register with either only real parts: R R R R or imaginary parts: I I I I. The SoA layout is better suited for achieving higher throughput, but may require data rearrangement prior to the computation. Temporary copies could be created even within the kernel if this memory layout does not match the remaining part of the application. The rearrangement costs of $O(p^2)$ can be amortized over the subsequent $O(p^4)$ memory accesses within a kernel call.

In the linearization of the triangular arrays, some zeroes—at most one SIMD word—can be inserted as padding whenever the “outer” dimension changes. This allows to increase the number of iterations in the innermost loop, which unfortunately is odd in most cases, to be a multiple of the SIMD width. This avoids testing and special casing for the last few elements of the input. The “superfluous” operations performed are complex multiplications with one operand being zero. This instruction must be performed anyway, since at least one element in the SIMD-register is valid.

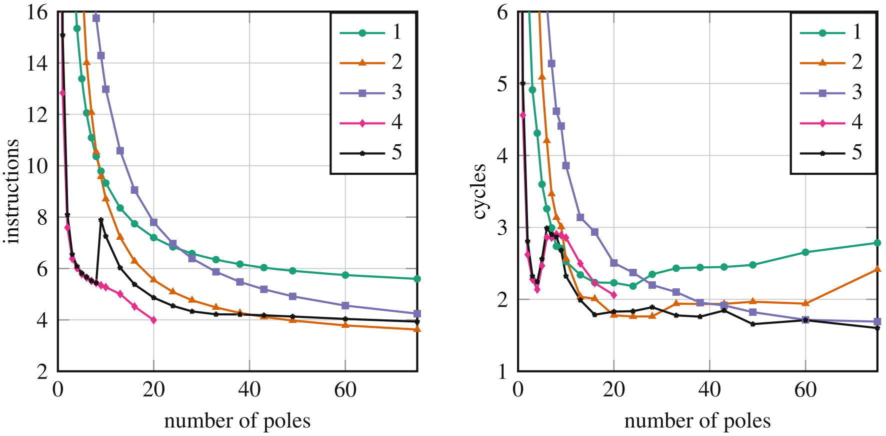
All the results will be accumulated eventually, but for intermediate results we can use SIMD-registers containing a partial accumulator per element that will be summed up ultimately.

5.3 M2L Operation Benchmark Results

In Fig. 5, we present results for several optimization approaches of the M2L kernel running on the “Haswell” machine operating in double precision and using AVX vectorization for the innermost loop.

We show instruction and cycle counts for the whole kernel amortized over the number of complex multiplications performed in the innermost loop. While there is often no linear correlation between instructions and runtime (Fig. 5 left), cycles typically correspond to the runtime (Fig. 5 right).

Plot 1 (green) shows a baseline vectorization in classical loop ordering and AoS memory layout. Switching the memory layout from AoS to SoA (as part of the kernel) in plot 2 (orange) improves performance for $p > 10$, but the high cost for



legend	algorithm	type	loop order		memory layout/cache lines	
—●—	1	fixed	l-m-j-k		AoS	→ R I R I R I R I R I
—▲—	2	fixed	l-m-j-k		SoA	→ R R R R R → I I I I I
—■—	3	fixed	l-j-m-k		SoA	→ R R R R R → I I I I I
—♦—	4	fixed	unrolled		AoS	→ R I R I R I R I R I
			unrolled	$p \leq 8$	AoS	→ R I R I R I R I R I
—★—	5	composite	l-m-j-k	$l, j \leq 30$	SoA	→ R R R R R → I I I I I
			l-j-m-k	$l, j > 30$	SoA	→ R R R R R → I I I I I

Fig. 5 Average instruction and cycle counts for the M2L operator in $O(p^4)$ complexity. The instruction and cycle counts including all loop overhead are amortized over the total iteration count of the innermost loop in Algorithm 3, i.e., over the basic operation: performing a single complex multiplication and accumulating the result

the data reorganization cannot be amortized for smaller values of p. Changing the loop ordering from `l-m-j-k` to `l-j-m-k` in plot 3 (blue) shows an improvement starting at $p = 40$ caused by better cache utilization. Eliminating the loop control instructions by fully unrolling the nested loops, but keeping the initial AoS memory layout, can be seen in plot 4 (purple). While this clearly results in the lowest instruction counts, the code size grows with $O(p^4)$ and quickly overflows the cache sizes, leaving it only viable for $p \leq 8$.

The approach in plot 5 (black) combines all mentioned techniques. For $p \leq 8$, unrolled routines are used with the initial AoS memory layout, while for larger values of p the layout is transformed to SoA. There the hybrid approach with splitting is employed using the `l-m-j-k` loop order for $l,j \leq l'$ and switching to the more cache-efficient `l-j-m-k` ordering afterwards. The splitter value $l' = 30$ has been obtained experimentally and may change on different hardware.

6 Conclusions

The main objective of our FMM is to provide a performance-portable implementation. To fully exploit the available hardware features while keeping a high-level abstraction, we decided to use C++ as our programming language of choice. This enabled us to implement our core algorithms in a readable and maintainable fashion, without losing the possibility to apply certain optimizations hidden in a hierarchy of abstraction layers. Such abstractions include, but are not limited to: memory layout, compile-time dependent loop unrolling, data and vectorization type generalization, and shared memory parallelization schemes. We have shown that high performance and high-level description of mathematical algorithms are not mutually exclusive. The provided assembly listing clearly shows optimal compilation and no overhead from the abstraction layers whatsoever. The generic code is capable of efficiently utilizing arbitrary SIMD-widths. Additionally, with the help of C++-lambdas and templates, features such as unrolling or ILP could be realized efficiently without intervening with the algorithmic description in the high-level language.

The code that was developed on an Intel Ivy Bridge (AVX) immediately utilized new AVX2 features when compiled and executed on an Intel Haswell without additional modifications. The runtime of an initial OpenMP near field kernel with 28 threads and no explicit vectorization accounted for 7.16 s. The optimized kernel, utilizing threading, SIMD and a fast reciprocal square root reduced the runtime to 1.32 s. This corresponds to a speedup of 5.42. The in-depth analysis of the former result showed that the floating point units are already saturated by instructions from the critical path. Hence, no further improvements can be expected and we regard the kernel as fully optimized.

However, we also noticed that a large fraction of the performance achieved on out-of-order hardware cannot be transferred to in-order machines directly. The compilers are still limited in the possibilities for optimizations and cannot compete against dynamic OoO features of modern CPUs. Especially instructions that break the pipelining flow like SQRT and DIV require special attention and careful tuning.

Our kernel benchmarking for the threaded reference implementation (16 threads, no SIMD) on BG/Q required a runtime of 185.9 s. By increasing the number of threads to 64, hence using the four-way SMT feature of the A2 processor, the runtime dropped to 56.8 s. This is already a speedup of 3.27 without additional code changes. The low port utilization due to a lacking OoO execution on the A2 is partly compensated by SMT. A subsequent analysis of the critical path showed that the floating point units are not fully saturated. Enabling SIMD does not change the saturation, but helps to reduce the loop count. To obtain port saturation we unrolled the inner loop. Still, the performance did not increase significantly. The final improvement was gained by interleaving independent loop iterations together with manual instruction scheduling in assembly. This measure reduced the kernel runtime to 3.24 s; a 17.53-fold improvement over the initial implementation.

We have seen that kernel performance can be limited by very different factors: e.g., full utilization of a kernel resource, cache overflows, loop overhead dominating over useful computation, compiler insufficiencies on in-order machines.

Notwithstanding the compilers capability to optimize the templated code well on the tested platforms, we still need an experienced software developer with deep insights into the hardware platforms to write this code in the first place.

The admittedly cumbersome template programming used for this project currently requires too much boilerplate code. But we are confident this will converge to a more readable form with C++14 and further language extensions.

Acknowledgement This work is supported by the German Research Foundation (DFG) under the priority programme 1648 “Software for Exascale Computing – SPPEXA”, project “GROMEX”.

References

1. Barnes, J., Hut, P.: A hierarchical O($N \log N$) force-calculation algorithm. Nature **324**, 446–449 (1986)
2. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A portable programming interface for performance evaluation on modern processors. Int. J. High Perform. Comput. Appl. **14**(3), 189–204 (2000)
3. Eastwood, J.W., Hockney, R.W., Lawrence, D.N.: P3M3DP–the three-dimensional periodic particle-particle/particle-mesh program. Comput. Phys. Commun. **19**(2), 215–261 (1980)
4. Fog, A.: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. http://www.agner.org/optimize/instruction_tables.pdf
5. Greengard, L., Rokhlin, V.: A fast algorithm for particle simulations. J. Comput. Phys. **73**(2), 325–348 (1987)
6. Kabadshow, I., Dachsel, H.: An error controlled fast multipole method for open and periodic boundary conditions. In: Sutmann, G., Gibbon, P., Lippert, T. (eds.) Fast Methods for Long-Range Interactions in Complex Systems. IAS Series, vol. 6, pp. 85–114. FZ, Jülich (2011)
7. Rahimian, A., et al.: Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures. In: SC ‘10, pp. 1–11. IEEE Computer Society (2010)

8. Springel, V.: The millennium-XXL project: simulating the galaxy population of dark energy universes. inSiDE **8**(2), 20–28 (2010)
9. White, C.A., Head-Gordon, M.: Derivation and efficient implementation of the fast multipole method. J. Chem. Phys. **101**, 6593–6605 (1994)

Springer

http://www.springer.com/978-3-319-22996-6

Recent Trends in Computational Engineering - CE2014
Optimization, Uncertainty, Parallel Algorithms, Coupled and Complex Problems
Mehl, M.; Bischoff, M.; Schäfer, M. (Eds.)
2015, X, 326 p., Hardcover
ISBN: 978-3-319-22996-6