International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

Research Paper

The International Journal of High Performance Computing Applications
1–21

DOI: 10.1177/1094342020964857
journals.sagepub.com/home/hpc


A CUDA fast multipole method with highly efficient M2L far field evaluation

Bartosz Kohnke[1], Carsten Kutzner[1], Andreas Beckmann[2], Gert Lube[3], Ivo Kabadshow[2], Holger Dachsel[2] and Helmut Grubmüller[1]

Abstract

Solving an N-body problem, electrostatic or gravitational, is a crucial task and the main computational bottleneck in many scientific applications. Its direct solution is an ubiquitous showcase example for the compute power of graphics processing units (GPUs). However, the naïve pairwise summation has $\mathcal{O}(N^2)$ computational complexity. The fast multipole method (FMM) can reduce runtime and complexity to $\mathcal{O}(N)$ for any specified precision. Here, we present a CUDA-accelerated, C++ FMM implementation for multi particle systems with r^{-1} potential that are found, e.g. in biomolecular simulations. The algorithm involves several operators to exchange information in an octree data structure. We focus on the Multipole-to-Local (M2L) operator, as its runtime is limiting for the overall performance. We propose, implement and benchmark three different M2L parallelization approaches. Approach (1) utilizes Unified Memory to minimize programming and porting efforts. It achieves decent speedups for only little implementation work. Approach (2) employs CUDA Dynamic Parallelism to significantly improve performance for high approximation accuracies. The presorted list-based approach (3) fits periodic boundary conditions particularly well. It exploits FMM operator symmetries to minimize both memory access and the number of complex multiplications. The result is a compute-bound implementation, i.e. performance is limited by arithmetic operations rather than by memory accesses. The complete CUDA parallelized FMM is incorporated within the GROMACS molecular dynamics package as an alternative Coulomb solver.

Keywords

Fast multipole method, Multipole-to-Local, molecular dynamics, electrostatics, CUDA

1 Introduction

The fast multipole method (FMM) was introduced by Greengard and Rokhlin (1987) to efficiently evaluate pairwise, Coulombic or gravitational, interactions in many body systems, which arise in many diverse fields like biomolecular simulation (Dror et al., 2012; Hansson et al., 2002), astronomy (Arnold et al., 2013; Potter et al., 2017) or plasma physics (Dawson, 1983). Moreover, the FMM can improve iterative solvers for integral equations by speeding up the underlying matrix-vector products (Engheta et al., 1992; Gumerov and Duraiswami, 2006).

The originally proposed FMM uses a spherical harmonics representation of the inverse distance r^{-1} between particles. For distant interactions (far field), which can be strictly defined, it uses multipole expansions built by clustered particle groups. The expansions are shifted and then transformed into Taylor moments by applying linear operators in a hierarchical manner to achieve linear scaling with respect to the number of particles. The complexity of the operators is $\mathcal{O}(p^4)$, where p is the order of the multipole expansion. White and Head-Gordon (1996) and Greengard and Rokhlin (1997) proposed rotational operators, that align the transformation axis to reduce the operator complexity to $\mathcal{O}(p^3)$. Cheng et al. (1999) used plain wave expansions to further reduce the complexity of the operators to $\mathcal{O}(p^2)$, however a few $\mathcal{O}(p^3)$ translations are still required. The algorithm has been developed further to support oscillatory kernels e^{ikr}/r (Ying et al., 2004). Fong and Darve (2009) parametrized the inverse distance function

[1] Theoretical and Computational Biophysics, Max Planck Institute for Biophysical Chemistry, Göttingen, Germany
[2] Jülich Supercomputing Centre, Forschungszentrum Jülich, Jülich, Germany
[3] Institute for Numerical and Applied Mathematics, Georg-August University of Göttingen, Göttingen, Germany

Corresponding author:
Helmut Grubmüller, Theoretical and Computational Biophysics, Max Planck Institute for Biophysical Chemistry, Am Fassberg 11, 37077 Göttingen, Germany.
Email: hgrubmu@gwdg.de

using Chebyshev polynomials and proposed a "black-box" FMM, which uses a minimal number of coefficients to represent the far field.

In atomistic molecular dynamics (MD) simulations, Newton's equations of motion are solved for a system of N particles (Allen and Tildesley, 1989) in a potential that accounts for all relevant interactions between the atoms. The integration time step is limited to a few femtoseconds such that the fastest atomic motions can be resolved. To reach the time scales many biomolecules operate on, millions of time steps need to be computed (Bock et al., 2013; Lane et al., 2013; Paul et al., 2017; Schwantes et al., 2014). This can easily require weeks or even months of compute time even on modern hardware (Kutzner et al., 2019). Hence, to speed up the calculation of an MD trajectory, the execution time for each individual time step has to be reduced. This can be achieved with better algorithms, with special-purpose hardware (Shaw et al., 2014), by introducing heterogeneous parallelization, e.g. harnessing SIMD, multi-core, and multi-node parallelism (Abraham et al., 2015; Hess et al., 2008; Páll et al., 2015) and by using GPUs (Páll and Hess, 2013; Salomon-Ferrer et al., 2013). Here, we utilize GPUs for our FMM implementation.

The electrostatic contribution to the inter-atomic forces is governed by Coulomb's law

$$\mathbf{F}_{ij} = \frac{1}{4\pi\varepsilon_0} \frac{q_i q_j}{||\mathbf{r}_{ij}||_2^2} \frac{\mathbf{r}_{ij}}{||\mathbf{r}_{ij}||_2} \tag{1}$$

where $\mathbf{r}_{ij} = \mathbf{x}_i - \mathbf{x}_j$ is a vector distance between atoms carrying partial charges q_i, q_j at positions $\mathbf{x}_i$, $\mathbf{x}_j$, ε_0 is the vacuum permittivity and $||\cdot||_2$ is the Euclidean distance. The calculation of nonbonded forces, i.e. Coulomb and van der Waals forces, is usually by far the most time-consuming part of an MD step. The van der Waals forces decay very quickly with distance r, so calculating them up to a cutoff distance suffices. The Coulomb forces, however, decay only quadratically with r, and the use of a finite Coulomb cutoff can therefore lead to severe simulation artifacts (Patra et al., 2003; Schreiber and Steinhauser, 1992). Direct evaluation of the electrostatic interactions in a typical biomolecular simulation system becomes prohibitive for two reasons. First, the $\mathcal{O}(N^2)$ scaling of a direct evaluation hinders its usage already at small system sizes, e.g. for $N \approx 50,000$ particles. Second, the usually employed periodic boundary conditions (PBC) make such calculation even impossible. Biomolecular simulation, therefore, requires an efficient Coulomb solver that properly accounts for the full, long-range nature of the electrostatic interactions.

To this aim, several FMM implementations have been developed. A standard FMM was included as an electrostatic solver for the NAMD package (Board et al., 1992; Nelson et al., 1996). Ding et al. (1992a) proposed the Cell Multipole Method (CMM) to simulate polymer systems of up to 1.2 million atoms. In further work, they combined CMM with the Ewald method showing a considerable speedup with respect to a pure Ewald treatment (Ding et al., 1992b). Niedermeier and Tavan (1994) introduced a structure-adapted multipole method. Eichinger et al. (1997) combined the structure-adapted multipole method with a multiple-time-step algorithm. Andoh et al. (2013) developed MODYLAS, a FMM adoption for very large MD systems and benchmarked it on the K-computer using 65,536 nodes. Very recently, it was extended to support rectangular boxes (Andoh et al., 2020). Yoshii et al. (2020) developed a FMM for MD systems with two-dimensional periodic boundary conditions. Shamshirgar et al. (2019) implemented a regularization method for improved FMM energy conservation. Gnedin (2019) combined fast Fourier transforms (FFTs) and the FMM for improved performance.

Considering efficient parallelization approaches, Gumerov and Duraiswami (2008) pioneered the GPU implementations of the spherical harmonics FMM with rotational operators. Depending on accuracy they achieved speedups of 30–70 with respect to a single CPU. Different GPU parallelization schemes for the "black-box" FMM (Fong and Darve, 2009) were implemented by Takahashi et al. (2012). Yokota et al. (2009) parallelized ExaFmm on a GPU cluster with 32 GPUs achieving a parallel efficiency of 44% and 66% for 10^6 and 10^7 particles, respectively. Even more GPUs (256) were used by Lashuk et al. (2009) for a system of 256 million particles. Rotational based Multipole-to-Local operators were efficiently parallelized with GPUs by Garcia et al. (2016). Task-based parallelization approaches to the FMM were proposed in Blanchard et al. (2015) and Agullo et al. (2016). A review of fast multipole techniques for calculation of electrostatic interactions in MD systems can be found in Kurzak and Pettitt (2006).

However, the early adoptions of FMMs in MD simulation codes were mostly superseded by particle Mesh Ewald (PME) (Essmann et al., 1995) due to its higher single-node performance. As a result, PME currently dominates the field. It is based on the FFT, which inherently provides the PBC solution. Nevertheless, PME suffers from a scaling bottleneck when parallelized over many nodes, as the underlying FFTs require all-to-all communication (Board et al., 1999; Kutzner et al., 2007, 2014). In addition, large systems with nonuniform particle distributions become memory intensive, since PME evaluates the forces on a uniform mesh across the whole computational domain.

In the era of ever-increasing parallelism and exascale computers, it is time to revisit the FMM, which does not suffer from the above mentioned limitations. To this end, we implemented and benchmarked a single-node full CUDA parallel FMM. Our implementation has been tailored for MD simulations, i.e. it targets a millisecond order runtime for one MD step by careful GPU parallelization of all FMM stages and by optimizing their flow to hide possible latencies. It was also meticulously integrated into the GROMACS package to avoid additional FMM independent performance bottlenecks. Here, we present three

different parallelization approaches. The implementation is based on the ScaFaCos FMM (Arnold et al., 2013), which utilizes spherical harmonics to describe the r^{-1} function. We use octree grouping to describe the interaction hierarchy. Such grouping is achieved by recursive subdivision of the cubic simulation box into eight equal subboxes. It has a major advantage: the far field operators can be precomputed for the whole simulation box, allowing for efficient parallelization. Additionally, the PBC computation becomes negligible as the PBC operators reduce to a single operator appliance. Moreover, a strict error control of the approximation (Dachsel, 2010) can be applied.

Here, we focus on the CUDA parallelization of the Multipole-to-Local (M2L) operator, which is most limiting to the overall FMM far field performance. An overview of the parallelized FMM, including all stages and complete runtimes, can be found in Kohnke et al. (2020b).

2 The fast multipole method

We consider a system of $N \gg 1$ particles. Following Hockney and Eastwood (1988), the challenge is to most efficiently evaluate

$$\Phi(\mathbf{x}_j) = \sum_{\substack{k=0 \\ k\neq j}}^{N-1} \frac{q_k}{||\mathbf{x}_j - \mathbf{x}_k||_2}, \quad j = 0, ..., N-1 \tag{2}$$

where $\mathbf{x}_j$ and $\mathbf{x}_k$ are positions of particles j and k, respectively and q_k is the charge of the k-th particle. For a direct solution of Eq. (2), interactions between all pairs of particles (j, k) with $j \neq k$ need to be computed. This leads to two nested loops and $\mathcal{O}(N^2)$ calculation steps.

2.1 Mathematical foundations

Expansion of the inverse distance between arbitrary particles $\mathbf{x}_j$ and $\mathbf{x}_k, j \neq k$ yields

$$\frac{1}{||\mathbf{x}_j - \mathbf{x}_k||_2} = \sum_{l=0}^{\infty}\sum_{m=-l}^{l} \frac{||\mathbf{x}_j||_2^l}{||\mathbf{x}_k||_2^{l+1}} Y_{lm}^*(\theta_j, \phi_j) Y_{lm}(\theta_k, \phi_k) \tag{3}$$

where

$$Y_{lm}(\theta, \phi) := \sqrt{\frac{2l+1}{4\pi}}\sqrt{\frac{(l-m)!}{(l+m)!}} P_{lm}(\cos\theta) e^{im\phi} \tag{4}$$

are spherical harmonics and Y^* their complex-conjugate, θ and ϕ are polar and azimuthal angle, respectively, and P_{lm} are the associated Legendre polynomials

$$P_{lm}(y) := (-1)^m (1-y^2)^{m/2} \frac{\mathrm{d}^m}{\mathrm{d}y^m} P_l(y) \tag{5}$$

where P_l are ordinary Legendre polynomials

$$P_l(y) := \frac{1}{2^l l!} \frac{\mathrm{d}^l}{\mathrm{d}y^l} (y^2 - 1)^l \tag{6}$$

The normalized associated Legendre polynomials form an orthonormal set of basis functions on the surface of a sphere. The j-th and k-th dependent parts of the right hand side of Eq. (3) are chargeless multipole moments

$$\mathring{\omega}_{lm}^j := \mathring{\omega}_{lm}^j(\mathbf{x}_j) := \frac{||\mathbf{x}_j||_2^l}{(l+m)!} P_{lm}(\cos\theta_j) e^{im\phi_j} \tag{7}$$

and chargeless local moments

$$\mathring{\mu}_{lm}^k := \mathring{\mu}_{lm}^k(\mathbf{x}_k) := \frac{(l-m)!}{||\mathbf{x}_k||_2^{l+1}} P_{lm}(\cos\theta_k) e^{im\phi_k} \tag{8}$$

respectively. The moments weighted with corresponding charges q_j and q_k, respectively, can be summed yielding charged multipole moments

$$\omega_{lm} := \sum_{j=0}^{J-1} q_j \mathring{\omega}_{lm}^j \tag{9}$$

and charged local moments

$$\mu_{lm} := \sum_{k=0}^{K-1} q_k \mathring{\mu}_{lm}^k \tag{10}$$

This allows to evaluate the potential at arbitrary particles at $\mathbf{x}_j, j = 0, ..., J-1$ due to a distant discrete charge distribution of K particles with positions $\mathbf{x}_k$, $k = 0, ..., K-1$ in terms of charged local moments and chargeless multipole moments with

$$\Phi(\mathbf{x}_j) = \sum_{l=0}^{\infty}\sum_{m=-l}^{l} \mu_{lm} \mathring{\omega}_{lm}^j \tag{11}$$

This calculation is referred to as far field. To achieve convergence in Eq. (3),

$$||\mathbf{x}_j||_2 < ||\mathbf{x}_k||_2 \tag{12}$$

has to be fulfilled for all distinct index pairs j and k. Application of addition theorems for regular and irregular solid harmonics (Tough and Stone, 1977) yields translation and transformation operators for the expansions. The moments ω_{lm} of a multipole expansion about a common origin $\mathbf{a}$

$$\omega(\mathbf{a}) := \sum_{l=0}^{\infty}\sum_{m=-l}^{l} \omega_{lm} \tag{13}$$

of particles $\mathbf{x}_j, j = 0, ..., J-1$ can be translated to a new origin $\mathbf{a}'$ with

$$\omega_{lm}(\mathbf{a}') = \sum_{j=0}^{l}\sum_{k=-j}^{j} \omega_{jk}(\mathbf{a}) \mathbf{A}_{l-j,m-k}(\mathbf{a} - \mathbf{a}') \tag{14}$$

where $\mathbf{A} \equiv \mathring{\omega}$ is the Multipole-to-Multipole operator. Further, the multipole expansion $\omega(\mathbf{a})$ can be transformed into a local expansion $\mu(\mathbf{r})$ at $\mathbf{r}$ with $||\mathbf{r}||_2 > ||\mathbf{x}_j||_2$, $j = 0, ..., J-1$ with

$$\mu_{lm}(\mathbf{r}) = \sum_{j=0}^{\infty}\sum_{k=-j}^{j} \omega_{jk}(\mathbf{a}) \mathbf{M}_{l+j,m+k}(\mathbf{a} - \mathbf{r}) \tag{15}$$

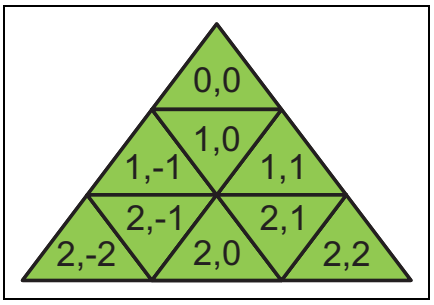

Figure 1. Indexing of triangular shaped matrices. The used indexing scheme is based on standard matrix index notation: The first subscript is a row number, the second one is a column number, which can be negative. In case of (l, m) notation, $l \geq |m|$ and $l \leq p$.

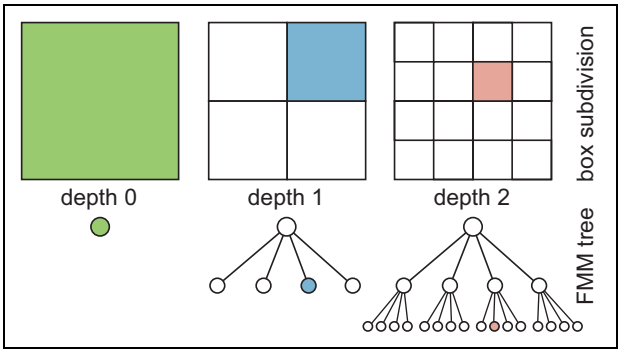

Figure 2. 2D example of FMM tree depth and resulting box subdivision for depths $\mathcal{D} = 0$, 1, and 2. At $\mathcal{D} = 1$, the whole simulation box (green) is split into four subboxes (blue). At $\mathcal{D} = 2$, each of the subboxes is split again into four smaller boxes (red).

where $\mathbf{M} \equiv \mathring{\mu}$ is the Multipole-to-Local operator. Finally, the local expansions $\mu(\mathbf{r})$ can be translated to any point $\mathbf{r}'$ with

$$\mu_{lm}(\mathbf{r}') = \sum_{j=l}^{\infty} \sum_{k=-j}^{j} \mu_{jk}(\mathbf{r}) \mathbf{C}_{j-l,k-m}(\mathbf{r} - \mathbf{r}') \tag{16}$$

where $\mathbf{C} \equiv \mathring{\omega}$ is the Local-to-Local operator.

2.2 Algorithm

Applying the operators defined in the previous section requires truncation of Eq. (3) to a finite multipole order p, which controls the accuracy of the solution approximation. Such expansions have a triangular shape with indexing shown in Figure 1. The truncation yields

$$\omega, \mu, \mathbf{A}, \mathbf{C} \in \mathbb{K}^{p \times p} := \left\{ (a_{lm})_{l=0,\ldots,p,\, m=-l,\ldots,l} \mid a_{lm} \in \mathbb{C} \right\} \tag{17}$$

and $\mathbf{M} \in \mathbb{K}^{2p \times 2p}$. The translations and transformations defined in the Mathematical foundations section are performed on moments expanded for clusters of particles. The clustering is based on a hierarchical partition of the computational domain $\Omega := [0, \ell]^3 \in \mathbb{R}^3$ into 8^d boxes for $d = 0, \ldots, \mathcal{D}$, where $\mathcal{D}$ is a predefined parameter. It leads to an octree of depth $\mathcal{D}$ shown in Figure 2. Figure 3 illustrates the six main stages of the FMM and their execution order. In the Particle-to-Multipole (P2M) stage, the particles occupying boxes on the deepest level $\mathcal{D}$ of the octree are expanded to multipoles ω with respect to the center of the boxes. In the Multipole-to-Multipole (M2M) stage, the expanded moments

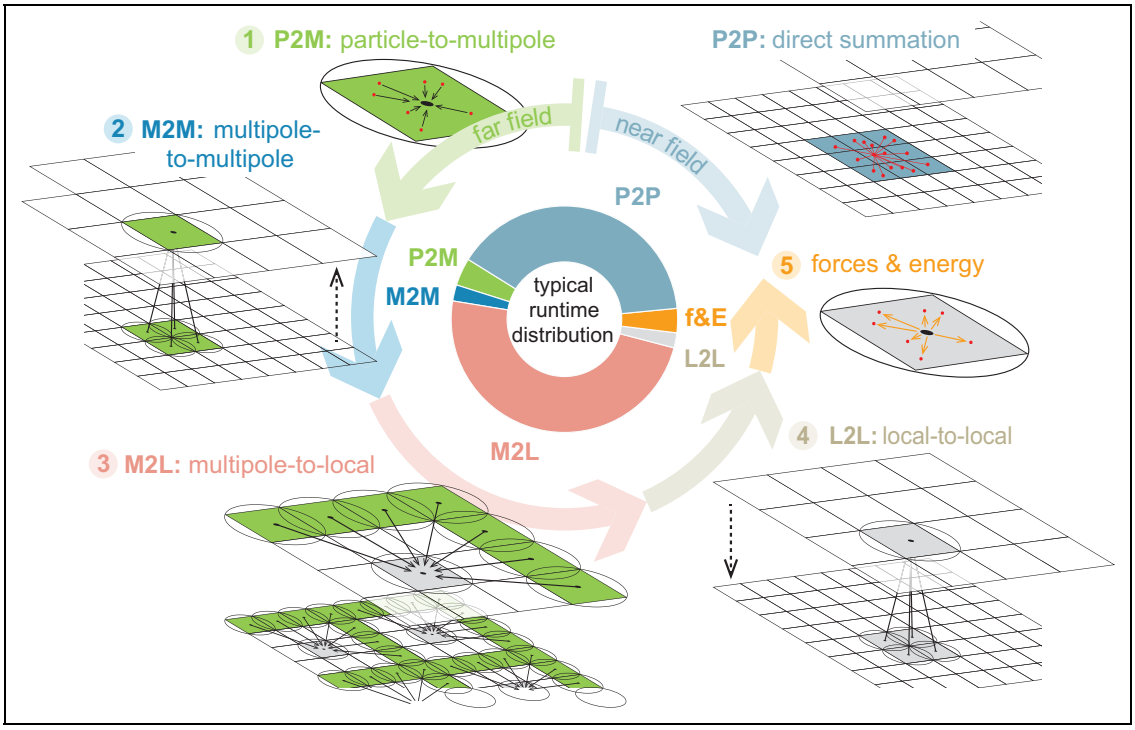

Figure 3. The six different stages of the FMM with an exemplary execution time distribution at the center. The near field part (P2P, top right corner) can be executed concurrently with the far field (stages 1–5) in a parallel implementation. Green squares indicate the representation by multipoles, light brown squares a representation by local moments, blue squares indicate direct summation.

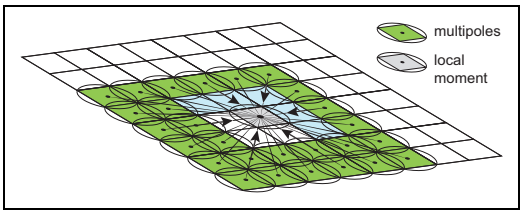

Figure 4. 2D interaction set $\mathcal{L}^d$ (green) of an arbitrary box with a local moment μ^d (light brown). The white boxes do not belong to interaction set $\mathcal{L}^d$. The interactions with the light blue boxes need to be skipped as well because they are nearest neighbors.

ω are distributed to all boxes of the octree by translating them level-wise from d to $d-1, d=\mathcal{D},...,1$ with the **A** operator. Subsequently, during the Multipole-to-Local (M2L) stage, the multipole moments ω are transformed into local moments μ by applying the operator **M**. A detailed description follows in the next section. After M2L, in the Local-to-Local (L2L) stage, the local moments μ are shifted from the octree root to the leaves with the **C** operator. The interactions between particles occupying the same lowest level boxes and between neighboring boxes (near field) are evaluated directly (P2P), Eq. (2). Finally, the far field forces and potentials are evaluated at particle positions in the tree leaves. Additionally, the number of directly interacting boxes can be defined with a `well separation criterion`, which controls how many layers of adjacent boxes interact directly on the lowest tree depth. In the following we will only discuss the case with one well separated layer of boxes.

2.3 The Multipole-to-Local (M2L) transformation

We will now explain the M2L transformation and its execution hierarchy. Let $\mathcal{D}$ be a fixed depth of an octree and p a multipole order. The multipole or local expansions in box i at depth $d=0,...,\mathcal{D}$ will be denoted by ω_i^d and μ_i^d, respectively. For each μ_i^d there exists an interaction set $\mathcal{L}_i^d$, $|\mathcal{L}_i^d|=208$. Figure 4 shows the interaction set $\mathcal{L}_i^d$, which contains the indices of multipole expansions ω_j^d in all children boxes of the direct neighbors of μ_i^d's parent box. The direct neighbors share at least one common vertex, edge or face with each other. A particular μ_i^d is calculated from all $\omega_j^d, j\in\mathcal{L}_i^d$, omitting μ_i^d's direct neighbor boxes in order to satisfy Eq. (12). This results in 189 $\mathcal{O}(p^4)$ M2L transformations.

Let $\mathcal{M}:=\{\mathcal{M}^d\}_{d=1}^{\mathcal{D}}$ be the set of level-wise operator sets $\mathcal{M}^d:=\{\mathbf{M}_{j\to i}^d|\mathbf{M}$ transforms j-th multipole moment to i-th local moment at level $d\}$. All M2L operations performed in the FMM octree yield

$$\mu_i^d=\sum_{\substack{j\in\mathcal{L}_i^d\\ \mathbf{M}_j^d\in\mathcal{M}^d}}\mathbf{M}_{j\to i}^d\omega_j^d, i=1,...,8^d, d=1,...,\mathcal{D}. \qquad (18)$$

Figure 5 shows one $\mathcal{O}(p^4)$ M2L transformation. It contains $\mathcal{O}(p^2)$ dot products between an ω and a part of the corresponding operator **M**.

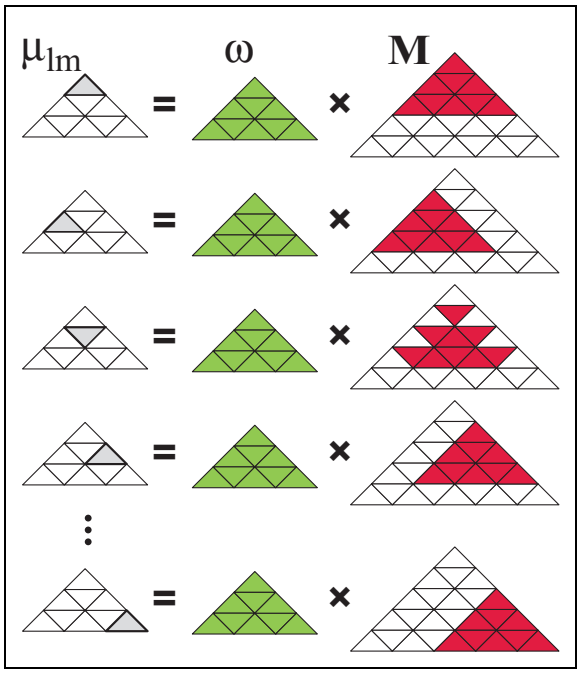

Figure 5. One M2L transformation. The matrix-vector like multiplication requires a part of the **M** operator (red) to calculate one element μ_{lm} (light brown) of a target expansion.

3 Implementation

We focus our GPU parallelization efforts on the M2L operator, as it is the most time-consuming FMM far field operator (see Figure 3 above and Figure 12 in Kohnke et al., 2020b). In PBC, it requires 189 transformations per box, whereas both M2M and L2L, which translate the moments between different tree levels, require only a single transformation per octree box (except for the root box). Since these transformations are of the same complexity, M2L involves 94.5× the number of operations as M2M and L2L combined. The second most time-consuming part is the P2P near field computation, which will not be discussed in this paper, was optimized as laid out in Páll and Hess (2013). Proper choice of $\mathcal{D}$ and p allows to balance the near and far field contribution, which minimizes the overall runtime. In case of parallel implementation these stages can run concurrently.

3.1 CUDA implementation considerations

We will now briefly outline the CUDA programming model, see Nickolls et al. (2008) for details. A typical GPU consists of a few thousand cores that are grouped into larger units called multiprocessors. CUDA threads are organized in `blocks`. Threads within a `block` are grouped into subunits called `warps`, each consisting of 32 threads. For optimal performance, threads within the same `warp` should execute the same instruction, otherwise the execution is serialized. This type of parallelization is called Single Instruction Multiple Threads (SIMT). Once a `block` of threads is spawned, it occupies the multiprocessor until the respective computations are completed. Dynamic scheduling is performed

warp-wise, thus thread blocks should consist of at least several warps to hide memory and arithmetic latencies within a multiprocessor. Blocks are organized in grids. Each block of a grid and thread of a block is identified with its unique 1D, 2D or 3D index. The dimensions of the grid and the blocks can be chosen independently. To identify threads, CUDA provides the 3D structures gridDim, blockDim, blockIdx and threadIdx.

We will use the following abbreviations: $B_\alpha := \texttt{blockDim}.\alpha$, $G_\alpha := \texttt{gridDim}.\alpha$, $Bid_\alpha := \texttt{blockIdx}.\alpha$ and $tid_\alpha := \texttt{threadIdx}.\alpha$, $\alpha \in \{x, y, z\}$. The hierarchy of threads described above affects the memory access and communication between threads. Whereas all threads can access global memory, this access should be minimized as it has a latency of a few hundred cycles. The memory within a block can be shared via shared memory. If no bank conflicts occur fetching shared memory is only slightly slower than register access (20–40 cycles). Synchronization of threads is possible only within a block. Since CUDA-6.0, threads within the same warp are able to share their content via the shuffle instruction by directly sharing their registers.

3.2 Sequential FMM and data structures

Our CUDA implementation is based on a C++11 version of the sequential ScaFaCos FMM (Arnold et al., 2013). It provides class templates with a possibility to use diverse memory allocators. With CUDA Unified Memory (Knap and Czarnul, 2019) the usage of original data structures became feasible by harnessing the C++ memory allocators.

To allow for an efficient manipulation of triangular shaped data (see Figures 1 and 5), we have implemented a dedicated triangular_matrix class that stores the moments and operators. It provides the indexing logic and utilizes a 1D vector of complex values (std:: vector<complex>) for this purpose. For symmetry reasons, it suffices to store one half of the triangular matrix for the moments, as the entries on the left ($m < l$) and right side ($m > l$) are identical except for the signs. The signs are computed on the fly at negligible costs from the parity of indices. The overall size of the matrices depends on the multipole order p. Exploiting symmetry, $(p^2 + p)/2$ complex values are stored for the expansions and $((2p)^2 + 2p)/2$ for the $\mathbf{M}$ operator.

Let $\mathcal{D}$ be a fixed depth of the tree. Thus, there are $d = 0, ..., \mathcal{D}$ levels in an octree. For Multipole-to-Local operations, as described in The Multipole-to-Local (M2L) transformation section, an underlying tree implementation is needed. Listing 1 shows a very basic approach for traversing an octree of depth $\mathcal{D}$. The function $\texttt{index}(x, y, z, d)$ applies the lexicographic approach to compute a unique 1D box index in the octree:

$$z\texttt{dim}(d)\texttt{dim}(d) + y\texttt{dim}(d) + x + \texttt{nb}(d-1) \qquad (19)$$

where $\texttt{dim}(d) := 2^d$ is the number of boxes in each orthogonal direction and $\texttt{nb}(d) := \sum_{d=0}^{\mathcal{D}} 8^d = \lfloor (8^{d+1})/7 \rfloor$ is the number of all boxes in an octree of depth d. The parent box index is easily obtained as $\texttt{index}(x/2,\ y/2,\ z/2,\ d-1)$.

Listing 1. Loops for traversing an octree in 3D space (pseudocode).

```
1 //chosen tree depth D
2 int d,z,y,x,i;
3 for (d = 0; d <= D; ++d)
4 {
5  for (z = 0; z < std::pow(2,d); ++z)
6  {
7   for (y = 0; y < std::pow(2,d); ++y)
8   {
9    for (x = 0; x < std::pow(2,d); ++x)
10   {
11    //unique one dimensional index
12    i = index(x,y,z,d);
```

Listing 2. The loops that start the M2L operators in the octree traverse the whole tree, compute the M2L interaction set and launch one M2L translation for each interaction in the computed set (pseudocode).

```
1 //chosen tree depth D
2 //for each relevant depth
3 int d,muZ,muY,muX,muXp,muYp,muZp
4 for(int d = 1; d <= D; ++d)
5 {
6  for(muZ=0; muZ<std::pow(2, d); ++muZ)
7  {
8   for(muY=0; muY<std::pow(2, d); ++muY)
9   {
10   for(muX=0; muX<std::pow(2, d); ++muX)
11   {
12    int muI = index(muX, muY, muZ, d);
13    //parent box of mu
14    muXp = muX/2;
15    muYp = muY/2;
16    muZp = muZ/2;
17    int omZ, omY, omX;
18    //operator index
19    int opI;
20    //computation of interaction lists
21    //based on the parent box information
22    for(omZ=(muZp-1)*2; omZ<(muZp+2)*2; ++omZ)
23    {
24     for(omY=(muYp-1)*2; omY<(muYp+2)*2; ++omY)
25     {
26      for(omX=(muXp-1)*2; omX<(muXp+2)*2; ++omX)
27      {
28       opI = opindex(omX-muX, omY-muY, omZ-muZ);
29       //remap out of bounds indices
30       periodic_remapping(omX, omY, omZ);
31       omegaI    = index(omX, omY, omZ, d);
32       M2L(mu[muI], omega[omegaI], M[opI]);
```

Listing 2 shows the sequential form of the M2L transformation. The first four for-loops (lines 3–9) traverse the octree as shown in Listing 1. omega and mu store the pointers to triangular_matrix objects for the multipole and local moments, respectively. The next three for-loops determine all multipole expansions $\omega_j^d \in \mathcal{L}_i^d$ that are needed for the calculation of μ_i^d, $i = 1, ..., 8^d$. Figure 6 shows the complete 2D operator set $\mathcal{M}$ for the WELL separation criterion $w = 1$. For each $d = 1, ..., \mathcal{D}$ the set $\mathcal{M}^d$ requires storing 343 pointers. A unique

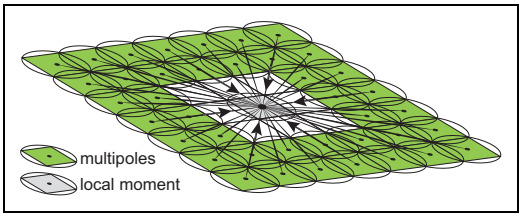

Figure 6. 2D representation of the operator set $\mathcal{M}$. In 3D, there are 342 possible positions (green) relative to the central box (light brown). Since the nearest neighbors (white) and self-interactions are excluded, the number of active operators reduces to 316.

Listing 3. Basic implementation of the M2L operato (pseudocode).

```
typedef triangular_matrix Tm;
void M2L(Tm* mu, Tm* omega, Tm* M)
{
 for (int l = 0; l <= p; ++l)
 {
  for (int m = 0; m <= l; ++m)
  {
   for (int j = 0; j <= p; ++j)
   {
    for (int k = -j; k < j; ++k)
    {
     mu(l,m) += M(j+l,k+m) * omega(j,k);
```

mapping function $\texttt{opindex}(x,y,z)$ returns a 1D index for each $\mathbf{M}_j^d \in \mathcal{M}^d, j = 0,...,343-1, d = 1,...,\mathcal{D}$ with

$$x + 3 + (y+3)\delta + (z+3)\delta^2 \tag{20}$$

where $x = x_\omega - x_\mu, y = y_\omega - y_\mu, z = z_\omega - z_\mu$ are the relative positions of ω and μ in 3D and $\delta = 3 + 4\sigma$. Here, σ denotes the number of directly interacting box layers according to the `well separation criterion`. In PBC, any $\omega_j^d \in \mathcal{L}_i^d$ with an out-of-box position is remapped from the corresponding periodic position with the `periodic_remapping(·)` function. Listing 3 shows a basic implementation of the `M2L` $(\cdot)$ function, which computes Eq. (15) up to order p in four nested for-loops.

3.3 Three CUDA parallelization approaches

The previous section described the basic sequential FMM. We will now present three different parallelization approaches. Approach (i) is conceptually straightforward, nevertheless it achieves decent speedups compared to a sequential CPU implementation with only minor parallelization work. It directly maps for-loops to CUDA threads, leaving the sequential program structure nearly unmodified. Approach (ii) performs well for high accuracy demands (high multipole orders $p \gtrsim 12$, double precision), however it scales poorly for smaller p. Approach (iii) minimizes the number of arithmetic operations by exploiting the symmetry of the $\mathbf{M}$ operator. It scales well in the broad range $0 \leq p \leq 20$, however it requires additional data structures to minimize bookkeeping and to utilize symmetries.

Listing 4. Direct mapping of FMM octree and M2L loops (top part, lines 1–20) to CUDA threads (bottom part, lines 22–43) (pseudocode).

```
1 *** M2L loops structure ***
2 //loop over depths
3 int d, muZ, muY, muX, omZ, omY,omX
4 for (int d = 1; d <= D; ++d)
5  //three loops over tree boxes
6  for(muZ = 0; muZ < std::pow(2, d); ++muZ)
7   for(muY = 0; muY < std::pow(2, d); ++muY)
8    for(muX = 0; muX < std::pow(2, d); ++muX)
9     //computation of the M2L interaction lists
10     for(omZ=(muZ/2-1)*2;omZ<(muZ/2+2)*2;++omZ)
11      for(omY=(muY/2-1)*2;omY<(muY/2+2)*2;++omY)
12       for(omX=(muX/2-1)*2;omX<(muX/2+2)*2;++omX)
13        //M2L operation
14        int l,m,j,k;
15        for(l = 0; l <= p; ++l)
16         for(m = 0; m <= l; ++m)
17          for(j = 0; j <= p; ++j)
18           for(k = -j; k < j; ++k)
19            //one complex multiplication
20            //and addition
21
22 *** CUDA M2L structure ***
23 //loop over tree levels d = 1,..,D on CPU
24 int dim = std::pow(2,d)
25 int p1 = p+1
26 //computation of the one-dimensional index i
27 int i = blockIdx.x * blockDim.x + threadIdx.x
28 //three loops over tree boxes on level d
29 int muZ = (i/(p1*p1*p1*216*dim*dim))%dim
30 int muY = (i/(p1*p1*p1*216*dim))%dim
31 int muX = (i/(p1*p1*p1*216))%dim
32 //computation of the M2L interaction lists
33 int om_z = (i/(p1*p1*p1*6*6))%6 - (muZ/2-1)*2
34 int om_y = (i/(p1*p1*p1*6))%6 - (muY/2-1)*2
35 int om_x = (i/(p1*p1*p1))%6 - (muX/2-1)*2
36 //M2L operation
37 int l = (i/(p1*p1))%p1
38 int m = (i/p1)%p1
39 if(m > l)
40   return;
41 int j = i%p1
42 for (int k = -j; k < j; ++k)
43 //one complex multiplication and addition
```

3.3.1 Naïve parallelization approach (I). The complete M2L operation in 3D requires 11 loops as shown in Listing 2. Listing 4 shows the comparison of the FMM loop structure and its naïve CUDA parallelization counterpart. Since CUDA provides a 3-component vector `threadIdx` to control the parallel execution of the threads, the main idea is to map the loops directly to the CUDA structures. To this end, we use a transformation between 1D and 3D indices. Any sequence of n indices $i = 0,..,n-1$ can be transformed into n m-dimensional tuples of indices $(x_0,...,x_{m-1})$ with $x_j = (i/m^j) \bmod m, j = 0,...,m-1$. As our FMM operates on cubic domains, the number of boxes is $\texttt{dim}(d)$ in each orthogonal direction for depths $d = 1,..,\mathcal{D}$. The loop over the M2L interaction set $\mathcal{L}$ is of a fixed size $(6 \times 6 \times 6)$ on each depth d. The M2L operation contains four for-loops of size $\leq p$. The iteration over the tree levels is performed by

the CPU. Since the M2L operations are level independent, the kernels are spawned asymmetrically for each level of the octree enabling overlapped execution. The last for-loop in Listing 3 is performed sequentially by each thread. It accumulates partial sums

$$\mu_{lm}(j) = \sum_{k=-j}^{j} \mathbf{M}_{l+j,m+k}\omega_{jk} \tag{21}$$

of the complete dot product

$$\mu_{lm} = \sum_{j=0}^{p}\sum_{k=-j}^{j} \mathbf{M}_{l+j,m+k}\omega_{jk} = \sum_{j=0}^{p}\mu_{lm}(j) \tag{22}$$

This reduces the number of atomic writes by a factor of $\mathcal{O}(p)$.

The naïve strategy allows a rapid FMM parallelization. Replacing the existing serial FMM loops with the corresponding CUDA index calculations leads to speedups that make the FMM algorithm applicable for moderate problem sizes. No additional data structures and code modification are required. However, the achieved bandwidth and parallelization efficiency is still far from optimal on the tested hardware.

3.3.2 CUDA dynamic parallelism approach (2). A substantial performance issue of the naïve approach is integer calculation, which introduces a significant overhead even for large p. For $d = 1, ..., \mathcal{D}$, $p^3 \times 216 \times 8^d$ threads are started, where each computes a valid pair of 3D source and target box indices to perform $\mathcal{O}(p)$ complex multiplications and additions $\mu_{lm} = \mu_{lm} + \mathbf{M}_{l+j,m+k}\omega_{jk}$. This leads to $\mathcal{O}(p^3)$ redundant source and target box index computations. A possible mitigation of the expensive index computations is Dynamic Parallelism (Jones, 2012). It allows to spawn kernels recursively, what simplifies hierarchical calculations. The dynamic approach exploits Dynamic Parallelism to avoid the expensive bookkeeping calculations of the naïve approach.

The determination of μ_i^d for $i = 0, ..., 8^d$ and $d = 1, ..., \mathcal{D}$ is done on the host as given in Listing 1. To this aim, the octree is traversed in 3D to precompute the coordinates (x_μ, y_μ, z_μ) of μ_i^d and the origin coordinates $(x_\mathcal{L}, y_\mathcal{L}, z_\mathcal{L})$ of $\mathcal{L}_i^d$. Together with 1D index i of μ_i^d, they are passed as arguments to a parent kernel spawned for each μ_i^d. Let $\mathcal{P}_J^d := \{j_0, ..., j_7\}$ be the set of indices of all boxes at depth d contained in the parent box of ω_j^d. Thus, for an arbitrary μ_i^d it holds: $\mathcal{L}_i^d = \cup_{J=0}^{25}\mathcal{P}_J^d$, $\mathcal{P}_J^d \cap \mathcal{P}_{J'}^d = \varnothing$, for any distinct pair $J \neq J'$. Listing 5 shows the parent kernel, that is engaged only in octree operations. To better utilize concurrency, it is started with $B_x = B_y = B_z = 3$ for $6 \times 6 \times 6$ interaction sets $\mathcal{L}_i^d$. The parent kernel consists of threads that can be uniquely identified with (tid_x, tid_y, tid_z) tuples. Each thread precomputes one 3D source positions $(x_\omega, y_\omega, z_\omega)$ of $\omega_{j_0}^d$, $j_0 \in \mathcal{P}_J^d$, $J = 0, ..., 25$ (lines 4–6). The index j_0 of the proper operator $\mathbf{M}_{j_0 \to i}^d$ is calculated from the relative 3D coordinates of μ_i^d and $\omega_{j_0}^d$ (line 12). Since the parent box index I of μ_i^d contains only its direct neighbors ($\mathcal{P}_I^d \not\subseteq \mathcal{L}_i^d$), the direct neighbors in $\mathcal{P}_I^d$ are omitted. Figure 7 illustrates the dynamic kernel. Each parent kernel spawns 26 child kernels with $G_x = G_y = G_z = 2$ and 2D `blocks` $B_x = p+1, B_y = p+2, B_z = 1$. One child kernel computes eight $\mathcal{O}(p^4)$ M2L transformations between one target μ_i^d and all $\omega_j^d, j \in \mathcal{P}_J^d$.

Listing 5. Parent kernel in the dynamic approach. It determinates valid ω coordinates and spawns child kernels performing M2L computations (pseudocode).

```
1 parent_kernel(int muI, int muX, int muY,
2               int muZ, int xL, int yL, int zL
                     ,...)
3 {
4   int omZ = threadIdx.z * 2 + zL;
5   int omY = threadIdx.y * 2 + yL;
6   int omX = threadIdx.x * 2 + xL;
7
8   if(muZ/2 != omZ/2 ||
9      muY/2 != omY/2 ||
10     muX/2 != omX/2 )
11  {
12    //operator index
13    int opI;
14    opI = opindex(omX-muX, omY-muY, omZ-muZ);
15    periodic_remapping(omX, omY, omZ);
16    int omegaI = index(omX, omY, omZ);
17
18    dim3 block(p+1,p+2,1);
19    dim3 grid(2,2,2);
20    child_kernel(muI, omegaI, opI,...);
21  }
22 }
```

Listing 6 shows child kernel computations, which can be divided in two parts. In the first part, $\omega_j^d, j \in \mathcal{P}_J^d$ and the operator $\mathbf{M}_{j \to i}^d$ are determined. Since indices of ω and $\mathbf{M}$ are provided by the parent kernel, the $(2 \times 2 \times 2)$ grid facilitates a straightforward way to determine eight different $\omega_j^d, j \in \mathcal{P}_J^d$ with $j = j' + Bid_x + Bid_y * \mathtt{dim}(d) + Bid_z * \mathtt{dim}(d)^2$, where j' is the index passed by the parent kernel. The operators $\mathbf{M}_j^d$ are obtained correspondingly, by replacing $\mathtt{dim}(d)$ with 7 and j' with the operator index passed by the parent kernel.

To decrease the number of global memory accesses, `shared memory` is used to cache ω and $\mathbf{M}$. This is advantageous, since one M2L operation executes $\mathcal{O}(p^4)$ steps on $\mathcal{O}(p^2)$ data structures. The triangular shaped matrices are converted to 1D arrays in `shared memory`, allowing consecutive addressing in the for-loops performing the reduction step. The `shared memory` storage index s_i of each moment $\omega_{lm} \in \omega_j^d$ is calculated with $s_i = l^2 + l + m$, where $l := tid_y$ and $m := tid_x$. A similar approach holds for the operator $\mathbf{M}_j^d$, however, since $\mathbf{M} \in \mathcal{O}(2p^2)$, threads need to be reused to write the elements into `shared memory` .

In our implementation, the direct neighbor operator is given the size $p = 0$. This allows to skip the remaining nearest neighbor interactions of μ_i^d for $\mathcal{P}_J^d \subsetneq \mathcal{L}_i^d$ by checking for the condition $p = 0$.

In the second part, the `j_k_reduction()` function computes the M2L operation. To mitigate the waste of

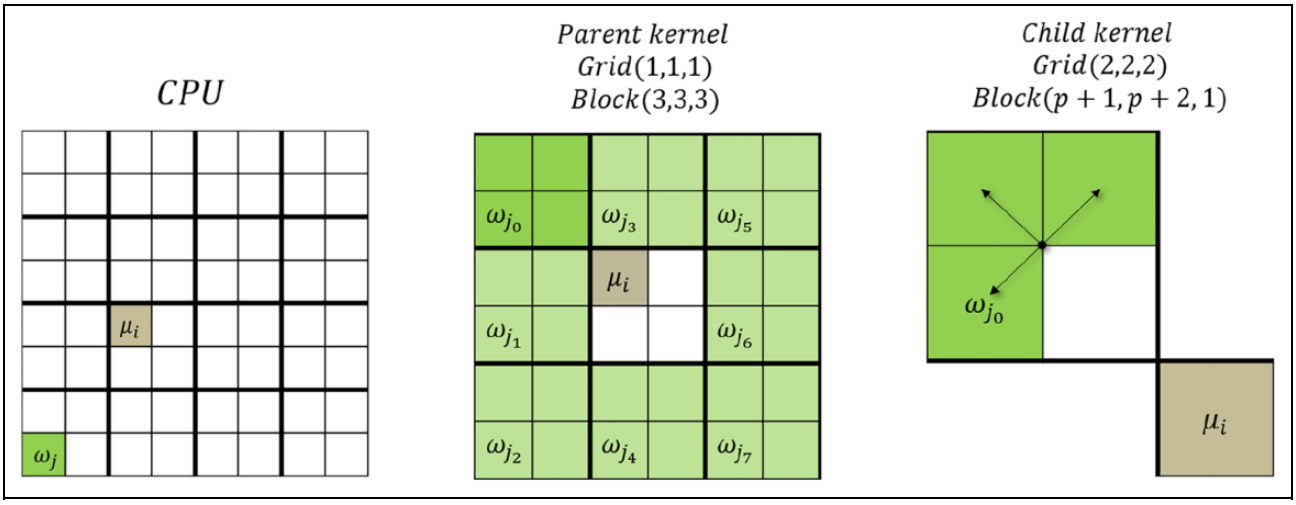

Figure 7. Dynamic M2L scheme. The CPU computes a μ_i^d and the corresponding $\mathcal{L}_i^d$. The parent kernel determines all valid $\mathcal{P}_j^d \subsetneq \mathcal{L}_i^d$ sets, whereas the child kernel performs the M2L operations for $\omega_{j_k}^d$, $j_k \in \mathcal{P}_j^d$, $k = 0, ..., 7$ and the target μ_i^d.

Listing 6. Child kernel in the dynamic approach (pseudocode).

```
template<typename M2L,typename OMEGA,
         typename MU>
void child_kernel(int muI, int omIstart,
                  int opIstart,
                  int opSize,
                  M2L** M_Operator,
                  OMEGA** Omega,
                  MU** Mu)
{
  int boxX = blockIdx.x;
  int boxY = blockIdx.y;
  int boxZ = blockIdx.z;

  extern __shared__ complex_type cache[];
  complex_type* shared_M=cache;
  complex_type* shared_O=&cache[opSize];
  //M2L operator
  int opI = opIstart + boxZ*7*7 + boxY*7 + boxX;
  M2L* M = M_Operator[opI];
  //skipping nearest neighbor operations
  if(M->p() == 0)
    return;
  //omega index
  int omI;
  omI = omIstart + boxZ * dim(d) * dim(d)
      + boxY * dim(d) + boxX;
  OMEGA* O = Omega[omI];

  int m = threadIdx.x;
  int l = threadIdx.y;
  int tx = l*(p+1) + m;

  if(tx < (p+1)*(p+1))
  {
    //writing of the moments and operators
    //intto shared memory
    shared_O[l*l+l+m] = O(l,m);
    shared_M[l*l+l+m] = M(l,m);
  }
  __syncthreads();

  MU* mu = Mu[muI];
  j_k_reduce(shared_O, shared_M, l, m, mu);
}
```

threads due to the triangular shape of the μ, ω and $\mathbf{M}$ and to minimize the number of atomic global memory writes, each thread executes the two innermost loops, Eq. (22), sequentially. However, a straightforward approach leads to `warp divergence`, since threads that correspond to $m > l$ indices of the target moments need to be skipped. Splitting the innermost loop, such that it is partly performed by threads $m > l$ circumvents this issue. Figure 8 and Listing 7 show a possible splitting scheme of the M2L operation. It uses $(p+1) * (p+2)$ threads, where some unique thread pairs are mapped to the same target index tuple (l, m) of a target element $\mu_{lm} \in \mu_i^d$. These compute a distinct part of the reduction. The described dynamic approach allows for further optimization, as it splits the computation in two independent parts. The parent kernels handle the octree position evaluation, whereas the child kernels implement the M2L computation. The efficiency of the this approach is satisfactory for high multipole orders. The necessity of an efficient parallelization also for small p leads to the next approach.

3.3.3 Presorted list-based approach with symmetric operators (3). In this approach, the FMM interaction pattern is precomputed for higher efficiency. Additionally, operator symmetries are exploited to reduce both the number of complex multiplications as well as global memory access.

3.3.4 Octree interactions precomputation. The pattern of interactions between the octree boxes is static, hence it can be precomputed and stored. This step does not need to be performance-optimal, as it is done only once at the start of a simulation that typically spans millions of time steps. In a PBC octree configuration, for each ω_i^d, $d = 1, ..., \mathcal{D}$, $i = 0, ..., 8^d - 1$ there exists an interaction set $\mathcal{R}_i^d$, $|\mathcal{R}_i^d| = 208$. It consists of all the indices j of local moments μ_j^d that a multipole ω_i^d is contributing to. Note that the index sets $\mathcal{R}_i^d$ and $\mathcal{L}_i^d$ (defined in section The Multipole-

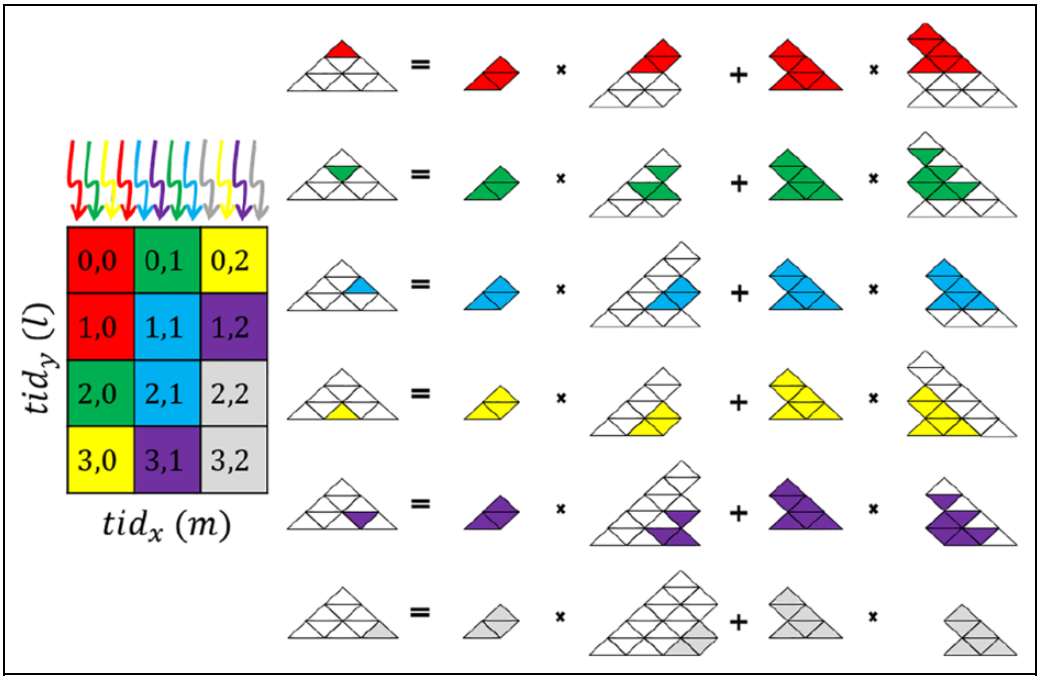

Figure 8. Thread splitting scheme to minimize `warp divergence`. Example of 12 threads performing six reductions. Threads are allocated in a 2D block. Each target (l, m) is shared by two threads (same color) performing a different part of the dot product.

Listing 7. Reduction function. It computes new target indices *ll*, *mm* from arguments *l*, *m* in a way that innermost loop is split to be performed by all threads in the block (pseudocode).

```
1 void j_k_reduce(complex_type& shared_O,
2                 complex_type& shared_M,
3                 int l, int m, MU* mu)
4 {
5   int l1, t1, ll, mm, f, f1, k_start, k_end;
6   t1 = tx - l;
7   l1 = t1/p_out;
8   //f = 0,1
9   f  = 1 - l - l1;
10  f1 = 1 - f;
11  ll = f1 * l1 + f * m;
12  mm = f1 * m  + f * l1;
13
14  complex_type mu_l_m = 0.0;
15  complex_type mu_l_m_j, op, om;
16  for (int j = 0; j <= p; j++)
17  {
18    mu_l_m_j = 0.0;
19
20    int jl       = j + l1;
21    int jl2_jl_m = jl * jl + jl + mm;
22    int jj2_j    = j * j + j;
23
24    for (int k = f * j; k <= f * j + j + f; k++)
25    {
26        int lk = k - j;
27        op = shared_M[jl2_jl_m + lk];
28        om = shared_O[jj2_j    + lk];
29        mu_l_m_j += op * om;
30    }
31    //changing sign in the odd j-th element
32    mu_l_m += change_sign_if_odd_j(j, mu_l_m_j);
33  }
34  //atomic add on global memory
35  *mu(ll, mm) += mu_l_m;
36 }
```

to-Local (M2L) transformation) are identical. For higher efficiency, the sets $\mathcal{R}_i^d$ are precalculated and stored as lists $\hat{\mathcal{R}}_i^d = (j_0, ..., j_{188})$ (with nearest neighbors skipped) with an arbitrary but fixed order. In addition, the corresponding operators are determined and stored as lists $\hat{\mathcal{M}}_i^d$ ordered in a way that

$$\mu_{j_k} = \mathbf{M}_{j_k} \omega_i, \quad k = 0, ..., 188 \tag{23}$$

describes all valid M2L transformations of the i-th multipole moment. The precalculation of $\hat{\mathcal{R}}_i^d$ and $\hat{\mathcal{M}}_i^d$ is achieved by sequentially traversing the octree as shown in Listings 1–2. Within the `omega` class, each ω_i stores 189 pointers to its targets μ_{j_k} and the corresponding pointers to M2L operators. We make sure that the internal list orders preserve the validity of Eq. (23), so that it suffices to store direct pointers to the target moments and operators instead of their indices. We will use the index list notation $\hat{\mathcal{R}}_i^d$ and $\hat{\mathcal{M}}_i^d$, keeping in mind that the lists actually store pointers.

The precomputed interaction lists $\hat{\mathcal{R}}_i^d$ and $\hat{\mathcal{M}}_i^d$ enable the following kernel configuration. The number of distinct M2L transformations for each ω_i^d is set with $G_z = 189$. $G_x = G_y = p$, thus $\mathcal{O}(p^2)$ CUDA `blocks` are spawned to handle $\mathcal{O}(p^4)$ interactions. The remaining $\mathcal{O}(p^2)$ operations are executed sequentially by each thread. $B_x = 8^{d-1}$ is the number of boxes on octree level $d-1$. This value fits CUDA architecture requirements for the `blocksize` particularly well, as it is always an even multiple of `warpsize`. For $d > 4$, B_x exceeds the largest allowed `blocksize` of 1024, so we replicate kernel launches for consecutive ω in strides of size 1024.

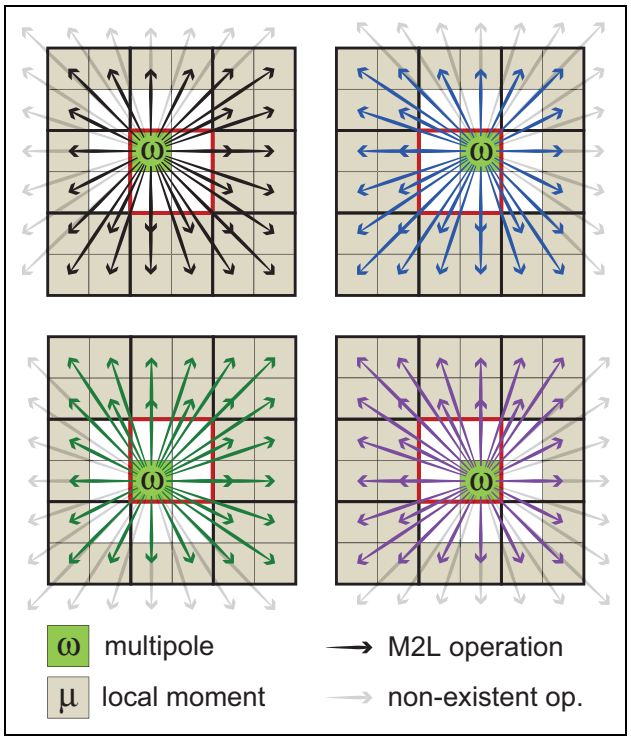

Figure 9. Different operator groups. The groups are represented by arrows of distinct color, depend on the position of ω within its parent (red squares). Four possible 2D operator groups $\mathcal{G}_s$ are shown. In 3D, there are eight different operator groups.

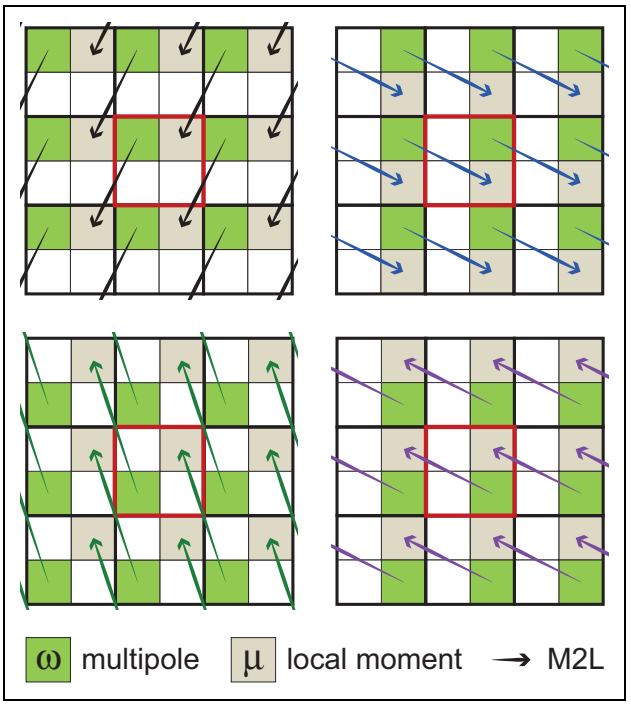

Figure 10. Parallelization of M2L operations for the operator groups shown in Figure 9. Each single operator is processed in parallel for all boxes on a level by starting one CUDA block for each $\omega_i \in \mathcal{G}_s$ (arrows of same color show one example for each operator group). The kernels are replicated for $s = 0, ..., 7$.

Figure 9 illustrates that for each position of ω_i^d within its parent box a specific interaction set $\mathcal{R}_i^d$ results. Therefore, the precomputed lists $\hat{\mathcal{R}}_i^d$ and $\hat{\mathcal{M}}_i^d$ require reshuffling to facilitate the straightforward indexing within the kernel. Let $\mathcal{G}_s$, $s = 0, ..., 7$ denote the eight possible groups governed by the position of ω within the parent box. The 8^d pointers to ω are reshuffled such that eight consecutive sequences of 8^{d-1} pointers in memory belong to the same group $\mathcal{G}_s$. Rigorously, for ω_i^d where $d = 1, ..., \mathcal{D}$ and $i = 0, ..., 8^d$ it holds $\omega_{i_k}^d \in \mathcal{G}_s, k = s8^{d-1}, ..., (s+1)8^{d-1} - 1, s = 0, ..., 7$. With reshuffled ω pointers the CUDA parallelization proceeds as shown in Figure 10. One kernel is started for each $\mathcal{G}_s$, $s = 0, ..., 7$. Each thread tid_x within a `block` evaluates one dot product (Eq. (22)). The pointers to the targets $\mu_{j_k}^d$ and to $\mathbf{M}_{j_k}^d$ are accessed with precomputed and presorted lists $\hat{\mathcal{R}}_i^d$ and $\hat{\mathcal{M}}_i^d$ without any additional integer operation, hence $Bid_z \equiv j_k$. The particular moments $(\mu_{lm})_{j_k}^d$, $j_k \in \mathcal{G}_s$ are evaluated in a parallel CUDA `block` with $l = Bid_x$ and $m = Bid_y$. As these are block variables, skipping of the $m > l$ part does not lead to `warp divergence`. Additionally, only the relevant part of the triangular operator matrix (Figure 5) needs to be loaded into `shared memory` to be accessed by all threads tid_x within the `block`.

A further improvement of the kernel is gained by rearranging the moments in memory. Threads tid_x of an l, m `block` access the same moments ω_{lm} of consecutive ω_i, with $i = tid_x$. For `warpwise` coalesced memory access, the arrangement of the moments in memory is switched from Array of Structures (AoS) to Structure of Arrays (SoA). The moments $(\omega_{lm})_i^d$, $i = 0, ..., 8^d - 1$ are stored in SoA triangular matrices such that for fixed l, m, the i indexed elements are contiguous in memory.

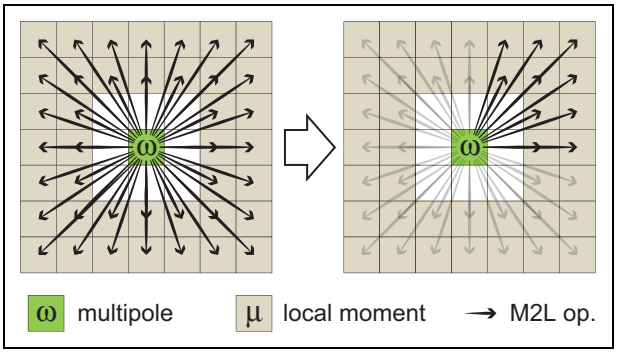

Figure 11. Reduction of the M2L operator set. The complete operator set $\mathcal{M}^d$ (left) and a reduced operator set $\tilde{\mathcal{M}}^d \subseteq \mathcal{M}^d$ (right) in 2D. Each black arrow symbolizes one M2L operator.

3.3.5 Operator symmetry. The symmetry of associated Legendre polynomials

$$P_{lm}(-x) = (-1)^{l+m} P_{lm}(x) \tag{24}$$

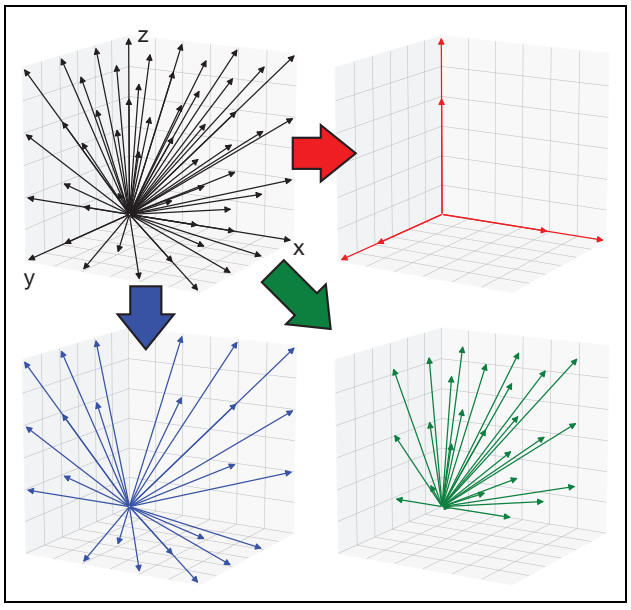

Figure 12. Grouping of the M2L operators according to their symmetry properties. The reduced operator set as shown in black in the upper left panel (a 2D version is shown in the right panel of Figure 11) is sorted into three groups (red, blue, green) depending on whether the operator is aligned with an axis (red), or within one of the *xy*, *xz*, or *yz* planes (blue), or none of that (green). Each of the red operators (in *x*, *y*, and *z* direction) has one symmetrical counterpart (in $-x$, $-y$, and $-z$ direction, respectively). Each of the blue operators has four symmetrical counterparts each (one in each quadrant of the plane). Each of the remaining operators has eight symmetrical counterparts each (one in each octant of the cube).

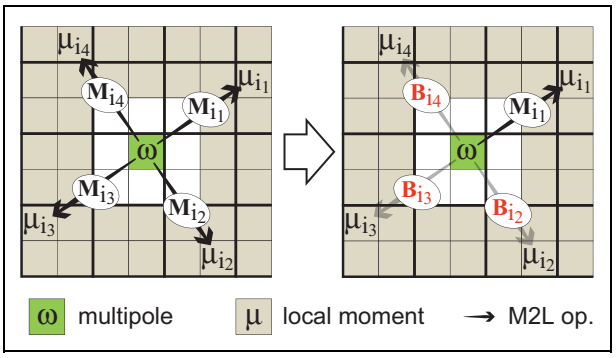

Figure 13. M2L operator symmetry exploitation. Left: Computation of four M2L operations with four orthogonal operators **M** (black arrows). Right: The use of bitsets **B** (orange) minimizes the redundant memory accesses and reduces the number of complex multiplications.

emerges directly from their definition (Eqs.5–6). It allows to reduce the size of the operator set $\mathcal{M}^d$, as shown in Figure 11.

In 3D, the complete operator set spans a cube with the operators originating from its center to all $7^3 - 3^3$ subcubes. The reduced operator $\tilde{\mathcal{M}}^d$ contains 56 M2L operators $\omega_i \rightarrow \mu_{j_x}$ $(x = 0, ..., 55)$ of one of the octants. Let the octant of the cube with parameters $\theta, \phi \in [0, \frac{1}{2}\pi]$ in spherical coordinates be the reference octant. The generation of particular operator moments with symmetrical functions

$$\mathbf{M}_{lm} = \frac{(l-m)!}{||x||_2^{l+1}} P_{lm}(\cos\theta) e^{im\phi} \tag{25}$$

where

$$e^{im\phi} = \cos(m\phi) + i\sin(m\phi) \tag{26}$$

yields three operator symmetry groups containing orthogonal operators that differ only by their sign. Figure 12 shows the symmetry groups in $\tilde{\mathcal{M}}^d$.

Depending on the relative position of ω_i in its parent box, the interaction set $\mathcal{R}_i$ requires a different subset of operators in $\mathcal{M}$, see Figure 9. Hence, for each $\mathcal{G}_s$, $s = 0, ..., 7$ on each depth $d = 0, ..., \mathcal{D}$, the operator set $\mathcal{M}^d$ and the corresponding index set $I = \{0, ...188\}$ can be split in disjunct subsets $\mathcal{T}$ such that

$$\begin{aligned}
\mathcal{T}_{\alpha_1} = \{ \quad & \mathbf{M}_j | \nexists\, \mathbf{M}_i \in \mathcal{M} : \text{abs}(\mathbf{M}_i) \\
& = \text{abs}(\mathbf{M}_j)\ \forall j \in I \,\} \\
\mathcal{T}_{\alpha_2} = \{ \quad & \mathbf{M}_{i_0}, \mathbf{M}_{i_1} | \text{abs}(\mathbf{M}_{i_x}) \\
& = \text{abs}(\mathbf{M}_{i_y})\ \forall x, y \in \{0, 1\}\} \\
\mathcal{T}_{\alpha_3} = \{ \quad & \mathbf{M}_{i_0}, ..., \mathbf{M}_{i_3} | \text{abs}(\mathbf{M}_{i_x}) \\
& = \text{abs}(\mathbf{M}_{i_y})\ \forall x, y \in \{0, 1, 2, 3\}\} \\
\mathcal{T}_{\alpha_4} = \{ \quad & \mathbf{M}_{i_0}, ..., \mathbf{M}_{i_7} | \text{abs}(\mathbf{M}_{i_x}) \\
& = \text{abs}(\mathbf{M}_{i_y})\ \forall x, y \in \{0, 1, 2, 3, 4, 5, 6, 7\}\}
\end{aligned} \tag{27}$$

where $\text{abs}(X) := \text{abs}(X_{l,m}), l = 0, ..., p, m = -l, ..., l$ and $\alpha_1 = (0, ..., 6)$, $\alpha_2 = (7, ..., 27)$, $\alpha_3 = (28, ..., 48)$, $\alpha_4 = (49, ..., 55)$. This property allows to reduce the $G_z = 189$ to $G_z = 56$, however further kernel modifications are required.

To make efficient usage of the operator symmetry, the lists $\hat{\mathcal{M}}_i^d$ for each ω_i are again reordered such that

$$\hat{\mathcal{M}}_i^d = (\mathcal{T}_{\alpha_1}, \mathcal{T}_{\alpha_2}, \mathcal{T}_{\alpha_3}, \mathcal{T}_{\alpha_4}) \tag{28}$$

The corresponding lists $\hat{\mathcal{R}}_i$ need to be resorted as well, to preserve Eq. (23). A bitset $\mathcal{B}$ is added to the **M** operator class to store signs of its elements. As these are complex values, it takes two bits to store the signs. The bitset is indexed in an array like manner with the most significant bit as the zero-th element. With $u = 2(l^2 + l) + 2m$, $\mathcal{B}(u)$ and $\mathcal{B}(u+1)$ represents the sign of the real and complex part of $\mathbf{M}_{lm}$, respectively. Since bitsets are precomputed during the operator initialization phase, they do not introduce any performance degradation whereas their additional memory footprint is negligible. Figure 13 shows an example of an M2L computation with bitsets. A single operator access from global memory computes 1, 2, 4, or 8 target moments μ depending on the operator type $\mathcal{T}$ as given in Eq. (27). The target moments μ_{t_γ}, with $\gamma = 0, ..., \beta$, $\beta = 1, 2, 4, 8$ for any source ω are

computed as follows. The intermediate products $\mu_{lm,jk} := \mathbf{M}_{l+j,m+k}\,\omega_{jk}$ of the complete dot product Eq. (22) are split in

$$\begin{aligned} ac &= \Re(\mathbf{M}_{l+j,m+k})\Re(\omega_{jk}) \\ bd &= \Im(\mathbf{M}_{l+j,m+k})\Im(\omega_{jk}) \\ ad &= \Re(\mathbf{M}_{l+j,m+k})\Im(\omega_{jk}) \\ bc &= \Im(\mathbf{M}_{l+j,m+k})\Re(\omega_{jk}) \end{aligned} \tag{29}$$

where $\mathbf{M}_{l+j,m+k}$ are the elements of the reference operator $\mathbf{M}_{t_0}$. The split products change their signs for $(\mu_{lm,jk})_{t_\gamma}$, $\gamma \neq 0$ according to $\hat{\mathcal{B}}_{t_\gamma} = \mathcal{B}_{t_\gamma} \oplus \mathcal{B}_{t_0}$, where $\mathcal{B}_{t_0}$ is the bitset of the reference operator, $\oplus$ is the binary XOR operator and $\gamma = 0, ..., \beta$, $\beta = 0, 2, 4, 8$ depending on the operator symmetry group $\mathcal{T}_{\alpha_x}$, $x = 1, 2, 3, 4$. For $x = 1$ there is no symmetric counterpart of the operator $\mathbf{M}$. For $x > 1$ the intermediate moments calculation is

$$\begin{aligned} (\mu_{lm,jk})_{t_\gamma} = \quad & \mathrm{sign}(ac, \hat{\mathcal{B}}_{t_\gamma}, u) - \mathrm{sign}(bd, \hat{\mathcal{B}}_{t_\gamma}, u+1) \\ & + i(\mathrm{sign}(ad, \hat{\mathcal{B}}_{t_\gamma}, u) + \mathrm{sign}(bc, \hat{\mathcal{B}}_{t_\gamma}, u+1)) \end{aligned} \tag{30}$$

with

$$\mathrm{sign}(x, \hat{\mathcal{B}}, u) = \begin{cases} x, \text{ if } \hat{\mathcal{B}}(u) = 0 \\ -x, \text{ if } \hat{\mathcal{B}}(u) = 1 \end{cases} \tag{31}$$

The `sign` function changes the sign of x by shifting the u-th bit of the bitset $\hat{\mathcal{B}}$ to the left most bit position and by evaluating the $x \oplus \hat{\mathcal{B}}_{shifted}$ subsequently. This creates no `warp divergence` since the sign change is a result of the arithmetic and logical operations.

The constant size of the lists, see Eq. (28), allows to implement the M2L kernel for the symmetry groups $\mathcal{T}_{\alpha_x}$, $x = 1, 2, 3, 4$ as a function template, resulting in a single kernel that efficiently treats different groups $\mathcal{T}_{\alpha_x}$. Listing 8 shows the kernel configuration for different symmetry groups. For different operator groups $\mathcal{G}_s$, $s = 0, ..., 7$, the kernels are replicated. The computation of the index i of ω_i is straightforward as pointers to ω_i are contiguous for any $\mathcal{G}_s$. For each symmetry group $\mathcal{T}_{\alpha_x}$, $x = 1, 2, 3, 4$ one kernel with distinct template parameters is started, where the first parameter describes the cumulative offset of a particular $\mathcal{T}_{\alpha_x}$ in $\hat{\mathcal{M}}_i^d$ and the second one is the number of symmetrical operators within the current $\mathcal{T}_{\alpha_x}$. The size of α_x, $x = 1, 2, 3, 4$ is set to G_z. The kernels are launched for all configurations $\mathcal{T}_{\alpha_x} \times \mathcal{G}_s$ asymmetrically to utilize concurrency. Listing 9 shows the implementation of the symmetric kernel. At the beginning, the reference operator $\mathbf{M}_{t_0}$ and the bitsets of all orthogonal operators $\mathcal{B}_{t_\gamma}$ are loaded into `shared memory` . Depending on the group $\mathcal{T}_{\alpha_x}$, $x = 1, 2, 3, 4$ different number of bitsets is loaded. The if statement, that tests the value of the template

Listing 8. Configuration and launches of the symmetric M2L kernel (pseudocode).

```
#define STREAMS 32
dim3 b(boxes_on_this_depth/8,1,1);
//multipoleorder p
dim3 g1(p,p,7);
dim3 g2(p,p,21);
dim3 g3(p,p,21);
dim3 g4(p,p,7);
int k = 0;
for (int s = 0; s < 8; ++s)
{
  M2L_symmetric<0,1>
  <<<g1,b,sm_size,stream[++k%STREAMS]>>>(s,...);
  M2L_symmetric<7,2>
  <<<g2,b,sm_size,stream[++k%STREAMS]>>>(s,...);
  M2L_symmetric<49,4>
  <<<g3,b,sm_size,stream[++k%STREAMS]>>>(s,...);
  M2L_symmetric<133,8>
  <<<g4,b,sm_size,stream[++k%STREAMS]>>>(s,...);
}
```

Listing 9. The symmetrical M2L kernel (pseudocode).

```
template<typename offset, typename group_type,
         typename Real>
M2L_symmetric(int s, int p, int d, Operators* B,
              Omegas* Omega){

int l    = blockIdx.x;
int m    = blockIdx.y;
int B_id = offset + blockIdx.z * group_index;
int omega_offset = s*blockDim.x + nb(d-1);
//shared memory alloc
unsigned int* bits0,bits1,bits2,bits3,
              bits4,bitsb5,bits6,bits7;
complex* B0;
//global memory accesses and shared memory
    writes
if(threadIdx.x == 0)
 bits0 = B[B_id]->bitset;
 for(int j = 0; j <= p; ++j)
  for(int k = -j; k <= j; ++k)
   B0[(j*(j+1) + k] = B[B_id](l+j,m+k);
 if(group_type > 1)
  bits1 = B_list[B_id+1]->bitset^bits0;
 if(group_type > 2)
  bits2 = B_list[B_id+2]->bitset^bits0;
  bits3 = B_list[B_id+3]->bitset^bits0;
 if(group_type > 4)
  bits2 = B_list[B_id+4]->bitset^bits0;
  bits3 = B_list[B_id+5]->bitset^bits0;
  bits2 = B_list[B_id+6]->bitset^bits0;
  bits3 = B_list[B_id+7]->bitset^bits0;
Real ac,bd,ad,bc;
complex mu_lm0, mu_lm1, mu_lm2,...
for(int j = 0; j <= p; ++j)
 for(int k = -j; k <= j; ++k)
  int u = 2*j*(j+1) + 2*k;
  O_jk=Omega(j,k,omega_offset+threadIdx.x);
  B_jk=B0[(j*(j+1) + k];
  ac = O_jk.real*B_jk.real;
  bd = O_jk.imag*B_jk.imag;
  ad = O_jk.real*B_jk.imag;
  bc = O_jk.imag*B_jk.real;
  mu_lm0 += complex(ac-bd,ad+bc);
  if(group_type > 1)
   mu_lm1
   +=
   complex(sign(ac,bits1,u)-sign(bd,bits1,u+1 )
           ,
           sign(ad,bits1,u)+sign(bc,bits1,u+1 )
          );
  if(group_type > 2)
   mu_lm2
   +=
   complex(sign(ac,bits2,u)-sign(bd,bits2,u+1 )
           ,
           sign(ad,bits2,u)+sign(bc,bits2,u+1 )
          );
   mu_lm3 += ...
  if(group_type > 4)
   mu_lm4 += ...
   mu_lm5 += ...
   mu_lm6 += ...
   mu_lm7 += ...
}
```

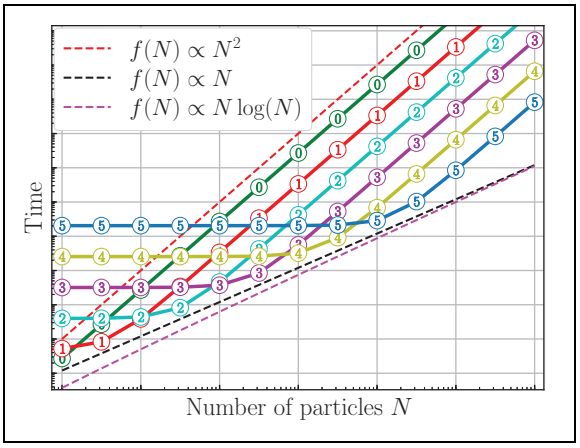

Figure 14. Qualitative sketch of the FMM scaling behavior. The optimal linear scaling (black dashes) with particle number N is achieved if and only if the tree depth $\mathcal{D}$ (as indicated by the colored numbers) is properly chosen for each N. For a constant $\mathcal{D}$, for small N, FMM run time is dominated by the far field computations, whereas for growing N, ultimately $\mathcal{O}(N^2)$ scaling results (red dashes).

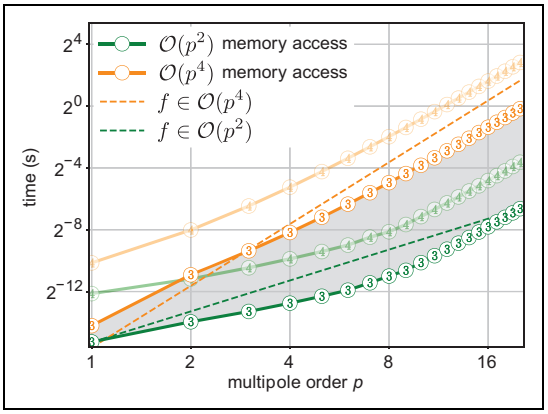

Figure 15. Runtime of the memory-bound microkernel (orange) and of the compute-bound microkernel (green) for two tree depths $\mathcal{D} = 3$ and $\mathcal{D} = 4$, as indicated by the encircled numbers. The run times of the implemented M2L kernels are expected in the shaded area between memory-bound and compute-bound microkernels.

parameter `group_type`, is resolved at compile time. The second part of Listing 9 shows the split complex multiplication implementation. The double nested for-loop computes $1, 2, 4$ or 8 $(\mu_{lm})_{t_\gamma}$ depending on the symmetry group $\mathcal{T}_{\alpha_x}$. The if statement within the innermost loop is resolved at compile time as well.

4 Benchmarks and discussion

We will now benchmark the performance and analyze the scaling behavior of the three different parallelization approaches described above.

4.1 General FMM scaling behavior

Figure 14 sketches the FMM scaling behavior with respect to the number of particles N, which is $\mathcal{O}(N)$ when the tree depth $\mathcal{D}$ is chosen properly. However, locally the FMM scales like $\mathcal{O}(n^2)$, with n being the average number of particles in the boxes at the lowest level $\mathcal{D}$. For a fixed multipole order p, at constant depth, a fixed number of $\mathcal{O}(p^4)$ far field operations are performed. In the regime of small N, the $\mathcal{O}(p^4)$ far field part completely dominates the FMM runtime, which is therefore essentially independent of N. At some critical N, the scaling curve switches to a quadratic behavior, because the P2P computations start to dominate the overall runtime. To benefit from the optimal linear scaling for growing N, the depth needs to be chosen properly. Varying p affects the slope of the overall linear scaling.

4.2 Benchmarking procedure

All performance tests were executed on a workstation with an Intel Xeon CPU E5-1620@3.60 GHz with 16 GB physical memory and a Pascal NVIDIA GeForce GTX 1080 Ti with 3584 CUDA Cores. This GPU has a theoretical single precision peak performance of 11.6 TFLOPS and maximal bandwidth of 484 GB/s. The device code was compiled with NVCC 9.1. All kernel timings were measured with the help of `cudaEvent` s and represent the average runtime of 100 runs.

In our performance comparisons we focus on $\mathcal{D} = 3$, as it provides sufficient parallelism to get proper performance metrics, which are also valid for $\mathcal{D} = 4$. For higher depths, the computation requires more kernel spawns due to limitations of `blocksize`, which leads to performance decrease. On the tested GTX 1080 Ti GPU a depth of $\mathcal{D} = 3$ is suitable for particle counts of 4×10^4–3×10^5, whereas higher N requires $\mathcal{D} = 4$ and $\mathcal{D} = 5$ for optimum performance. The current implementation of the symmetric parallelization approach allows for a maximum depth of $\mathcal{D} = 5$ at which up to $N \approx 1.2 \times 10^6$ particles can be handled efficiently. The limitation is caused by memory optimization, in which redundant pointers are stored to minimize the costs of scattered global memory writes. It can be switched off allowing for $\mathcal{D} = 6$ and system sizes up to $N \approx 10^8$. On the tested Pascal GPU this optimization increases performance by about 10%, while on a Turing GPU the effect of the optimization is negligible (Kohnke et al., 2020a).

4.3 Microbenchmarking

To evaluate the different parallelization approaches in context of the underlying hardware, we estimated the GPU performance bounds for the M2L transformation operation. To this aim, we implemented two benchmarking microkernels, which execute exactly the number of arithmetic operations and memory accesses as the M2L operation does. However, additional possible performance bottlenecks (Nickolls et al., 2008) like `warp divergence`,

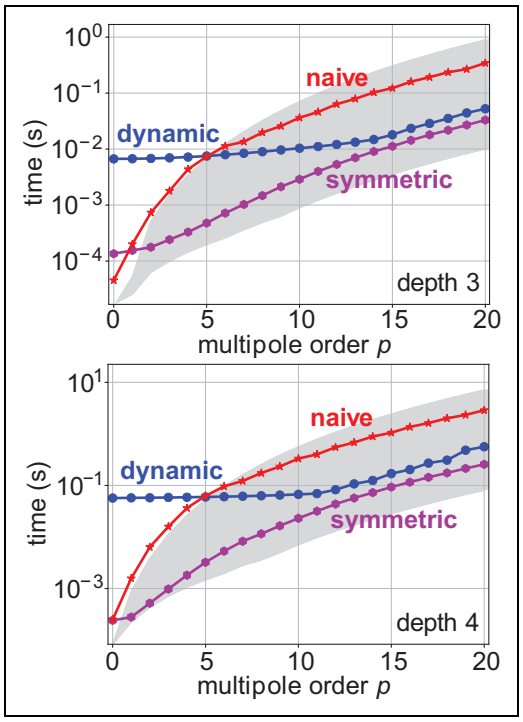

Figure 16. Runtime comparison of the three different M2L implementations. For each multipole order, $8^{\mathcal{D}} \times 189$ single M2L operations are computed for $\mathcal{D} = 3$ (top) and $\mathcal{D} = 4$ (bottom). The gray area marks the gap between the memory bound and the compute-bound reference microkernel shown in Figure 15.

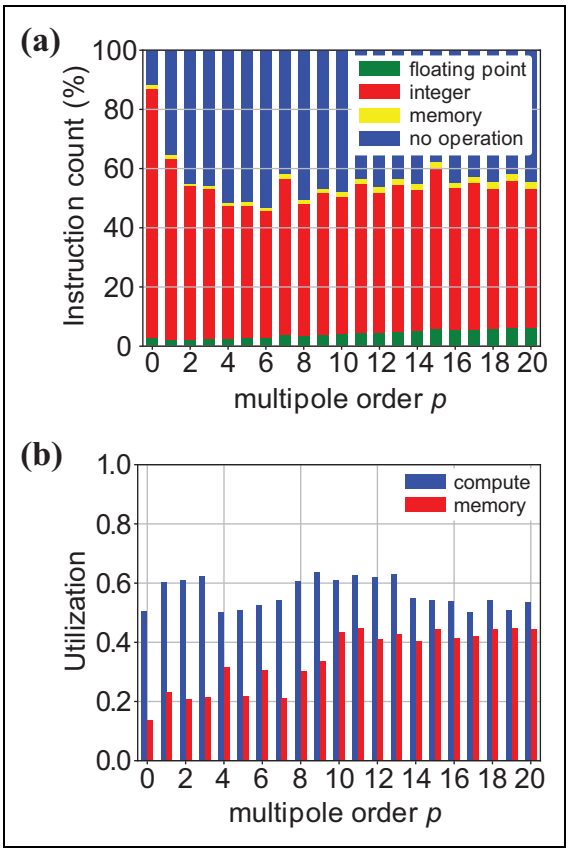

Figure 17. Performance analysis of the naïve M2L kernel. (a) Instructions distribution. (b) Memory and compute utilization.

noncoalesced memory accesses, `shared memory` bank conflicts and atomic writes are eliminated. The microkernels were then used to determine the effective runtime bounds for our three different parallelization approaches.

Figure 15 shows the absolute runtimes of the microkernels. To get the maximal theoretical throughput of the M2L kernel, we assumed the execution of three global memory accesses, eight bytes each, to perform one complex multiplication and one global addition, i.e. $\mathcal{O}(p^4)$ memory accesses for $\mathcal{O}(p^4)$ arithmetic operations. This results in a clearly memory-bound kernel with $\mathcal{O}(p^4)$ scaling in the examined range of p. For the lower bound we assumed an idealized scenario: for each box in the octree, $\mathcal{O}(p^2)$ memory accesses are performed for the moments and operators, whereas the data needed for $\mathcal{O}(p^4)$ operations is assumed to be available in registers. The full $\mathcal{O}(p^4)$ memory access approach utilizes nearly the full bandwidth of the GPU, achieving 370 Gb/s. However, the computation performance is only about 89 GFLOPS, which is $\approx 0.8\%$ of the GPU's peak performance. The second, $\mathcal{O}(p^2)$ memory access approach, varies depending on p. For $p < 10$ we observe subquadratic scaling of the execution time, indicating that the $\mathcal{O}(p^4)$ arithmetic operations are fully hidden. For $p > 10$ the curve shifts to the $\mathcal{O}(p^4)$ regime, achieving up to 8 TFLOPS, i.e. 70% of the GPU's peak performance. Compute and memory utilization are balanced at $p = 10$, which is where the curve switches from $\mathcal{O}(p^2)$ to the $\mathcal{O}(p^4)$ slope.

4.4 Performance comparison

We will now discuss the efficiency of the three proposed parallelization approaches.

4.4.1 Naïve kernel. Figure 16 shows the absolute executions times of the different kernels for $\mathcal{D} = 3$ and $\mathcal{D} = 4$. In the whole p range, the naïve kernel's theoretical arithmetic intensity (see Roofline model, Williams et al., 2009), is a lot smaller than the ratio $R = 23.95$ FLOPS/byte obtained from the GPU's FLOP rate (11.6 TFLOPS) divided by its memory transfer rate (484 GB/s). This indicates that the kernel is bandwidth limited. However, additional integer computation is required for the calculation of 3D octree indices of the interaction sets $\mathcal{L}$. Figure 17a shows that for the naïve kernel, much less than 10% of the issued instructions are useful floating point operations. A large computational overhead emerges from a high number of integer operations in the innermost for-loop. Here, each of $\mathcal{O}(p)$ complex multiplications requires 31 integer additions, 16 integer multiplications, 9 modulo operations and 11 integer divisions. In addition, performance is significantly reduced by `warp divergence`, since different threads in a `block`

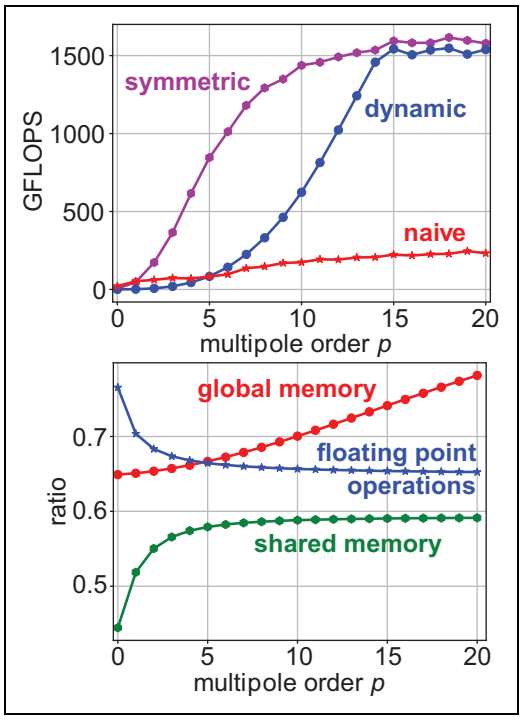

Figure 18. M2L kernel performance comparison by two different metrics. Top: FLOPS achieved by different kernels. Bottom: Ratio of global and shared memory accesses and floating point operations for the symmetric kernel with respect to the non-symmetric approach.

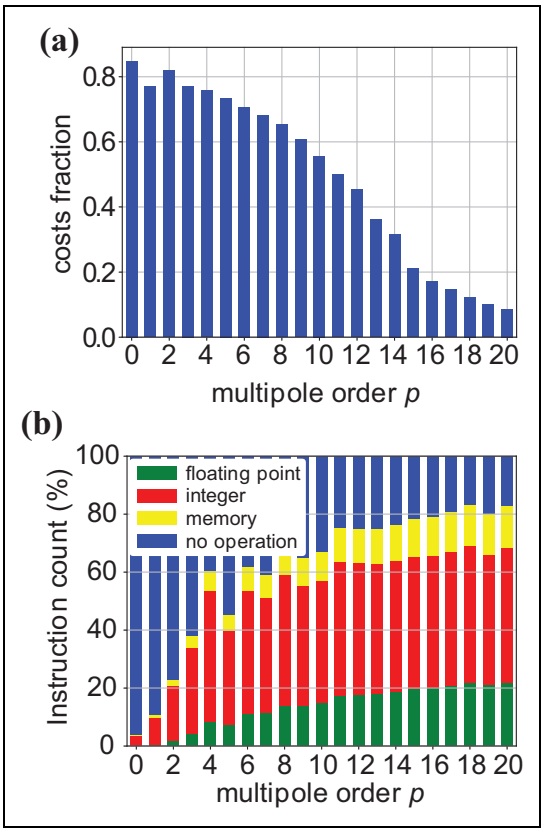

Figure 19. Performance analysis of the dynamic M2L kernel. (a) Kernel launching latencies. (b) Instructions distribution.

resolve the condition $m > l$ (line 36 of Listing 4). This effect is labeled as `no operation` in Figure 17a. Avoiding `warp divergence` would require different kernels for each $0 \leq p \leq 20$, since mapping of indices to threads differs at each p. Figure 17b shows, how well the GPU is utilized by the naïve kernel. As both memory and compute achieve roughly 50% of maximal possible utilization, the performance is likely limited by the latency of arithmetic or memory operations.

The maximum number of achieved FLOPS ($p = 19$) is at 2% of the GPU capability, see Figure 18 (top). The effective bandwidth reaches nearly 500 GB/s, which is more than the maximum memory throughput of the GPU. As seen in Figure 16, the naïve kernel achieves runtimes similar to the memory-bound microkernel. For values $p > 6$, the kernel is slightly faster than the memory-bound reference kernel, and that is for the following reason. The fact that the innermost for-loop of the naïve kernel is executed sequentially allows cache reuse. Each element ω_{lm} can be reused 189 times for a different M2L transformation and each element of the operator $\mathbf{M}$ is reused 8^d times at tree depth d. This leads to local cache throughput of roughly 3,500 GB/s, which approaches the maximal theoretically possible cache bandwidth of the GPU.

Additionally, the achieved occupancy per each Streaming Multiprocessor (SM) is at 46% of a possible maximum of 50% at this kernel configuration, a limit caused by the number of registers (64) used in the kernel.

4.4.2 Dynamic kernel. This kernel utilizes Dynamic Parallelism to minimize the index overhead computation introduced by the naïve kernel. The sizes of the child kernels $(2, 2, 2)$ allow for utilization of concurrency on the GPU, since the work in the child kernels is fully independent. On the underlying hardware the maximum number of resident grids per device is limited to 32.

Figure 19a shows the relative costs emerging from launching the child kernels, which become irrelevant only for large p. At small p, the latencies dominate the computation time, leading to an almost constant runtime for $p \leq 10$ that can be seen in Figure 16 for the dynamic kernel. Figure 19b shows the instruction distribution for the dynamic kernel. From $p \approx 3$ on, the fraction of floating point operations is significantly larger than for the naïve kernel. However, the large number of integer operations and `warp divergence` still limits the performance. Another issue is the small `block` size of the child kernels, which limits the SM occupancy for different p values. For $p < 5$, e.g. each `block` consists of only one `warp` . This limits the SM utilization, as 32 blocks but 64 `warps` can be executed simultaneously. For $p \geq 5$, the occupancy is only limited by the register usage, achieving nearly 100% of the theoretical possible occupancy. Limiting the register usage, however, increases the local memory traffic and does not further enhance the performance.

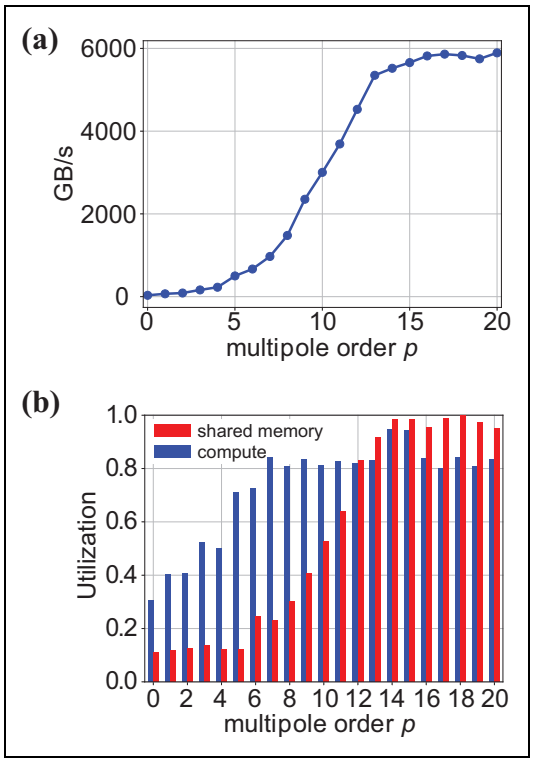

Figure 20. Performance analysis of the dynamic M2L kernel. (a) Shared memory throughput. (b) Shared memory and compute utilization.

As `shared memory` usage is an essential part of the dynamic kernel, we tested how its utilization affects the overall performance. Figure 20a and Figure 20b show the `shared memory` throughput and the GPU utilization of the dynamic kernel, respectively. For $p < 12$ the kernel is clearly compute-bound, hence the `shared memory` operations are fully hidden. At $p = 12$–15 we can observe a balance between memory and compute operations. `Shared memory` is limiting only for $p > 15$, however the achieved throughput of 6000 GB/s is at the limit of the underlying GPU.

The overall performance of the kernel gets considerably higher compared to the naïve kernel for $p > 6$, achieving a maximum of 1600 GFLOPS for $p = 20$, see Figure 18 (top). Nevertheless, memory and FLOPS peaks are achieved only for $p > 15$. At $p < 5$, the dynamic kernel performs worse than the $\mathcal{O}(p^4)$ benchmarking microkernel mainly due to kernel launch latencies mentioned above.

4.4.3 Symmetric kernel. The symmetric M2L kernel is composed of four subkernels started asynchronously for each symmetry group $\mathcal{T}_{\alpha_x}, x = 1, 2, 3, 4$ (see Eq. (27)). Figure 21 shows the achieved speedup compared to the standard implementation. Additionally, it also shows the speedups of each symmetric part $\mathcal{T}_{\alpha_x}$. As expected, the eight-way symmetric kernel performs best, followed by the four-way and then the two-way symmetric kernel. The overall speedup of the symmetric kernel combines the speedups of the subkernels. However, the achieved overall speedup

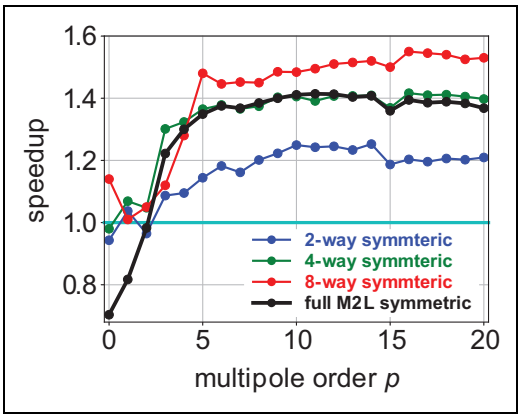

Figure 21. Speedups due to symmetry properties for different symmetry groups $\mathcal{T}_{\alpha_x}$, $x = 2, 3, 4$ (colored) and for the whole symmetric M2L operator kernel (black) compared to a non-symmetric implementation (cyan).

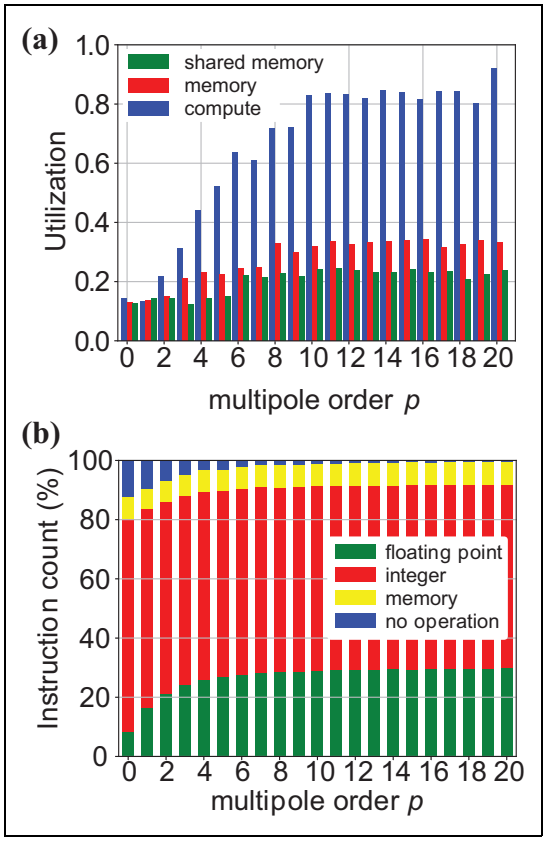

Figure 22. Performance analysis of the symmetric M2L kernel. (a) Hardware utilization. (b) Instructions distribution.

is not directly proportional to the number of the symmetrical counterparts, because the additional bit shifting and sign changing operations introduce a growing overhead for larger symmetry. In addition, register utilization is larger for the kernels with higher symmetry, harming the achieved SM occupancy. From here on, we will combine the metrics for all subkernels, referring to them as one symmetric kernel. These metrics take into account the overlapping

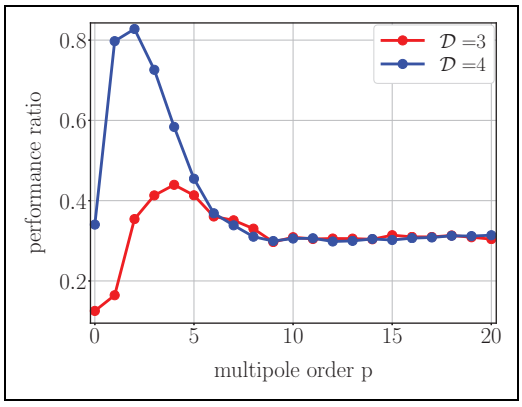

Figure 23. Absolute performance ratio of the symmetric M2L kernel with respect to the compute-bound reference microkernel with $\mathcal{O}(p^2)$ memory loads for $\mathcal{D} = 3$ and $\mathcal{D} = 4$.

execution of the symmetric subkernels. Figure 22a shows how the hardware is utilized by the symmetric kernel. This kernel is compute-bound over nearly the complete p range. As seen in Figure 22b, the fraction of floating point operations is significantly larger than in both previous approaches. `Warp divergence` is eliminated completely. The `no operation` part, resulting from pipeline latencies becomes negligible for $p > 6$. In this range, the `gridsizes` $((p+1)*(p+2)/2, 7, 1)$ and $((p+1)*(p+2)/2, 21, 1)$ of particular symmetric subkernels are too small to provide enough `blocks` to utilize the complete device, so that pipeline latencies become an issue. However, with larger p hardware utilization increases.

The register usage of the symmetric subkernels varies between 48% and 71% (not shown). Based on the kernel configuration, the theoretical maximum possible average SM occupancy for all subkernels is 57%. The kernels achieve 50% and are mainly limited by register usage. Further occupancy optimization is unlikely to further increase performance markedly, as kernels with a large register usage do not require optimal occupancy (Volkov, 2010).

As Figure 18 (top) shows, the FLOP rate of the symmetric kernel is much higher in the range $1 < p < 16$ compared to the other two kernels. However, the FLOP rates achieved by the symmetric and dynamical kernel for $p \geq 15$ are similar. Nevertheless, the symmetric scheme clearly outperforms the dynamic one when comparing the absolute execution times, see Figure 16. Figure 18 (bottom) demonstrates that, compared to the non-symmetric kernel, the symmetric kernel needs fewer floating point operations for the M2L stage. Figure 23 shows the absolute performance ratio of the symmetric kernel compared to the compute-bound reference microkernel. It achieves roughly 30% of the reference microkernel performance in $2 < p < 21$. Additionally, for both depths, the kernel shows an ideal scaling as the showed ratio remains nearly constant for $p > 5$. Hence, the scaling of the symmetric kernel follows the compute-bound reference microkernel scaling. It achieves a subquadratic scaling for $p < 10$ and switches slowly to $\mathcal{O}(p^4)$ regime, that is limited only by arithmetic throughput, compare Figure 15 and Figure 16.

5 Conclusions

Here, we have presented three different CUDA parallelization approaches for the Multipole-to-Local operator, which is performance limiting for the overall FMM performance. The first approach preserves the sequential loop structure and does not require any special data structures. It makes use of CUDA Unified Memory to achieve decent speedups compared to a sequential CPU implementation. It is useful, e.g. for rapid prototyping or for simulation systems with small to moderate numbers of particles. However, it comes with a large computational overhead due to additional integer operations.

The second approach, which exploits CUDA Dynamic Parallelism, avoids this drawback and achieves very good performance at high accuracy demands, i.e. for large multipole orders. Its main drawback is a lack of performance at low multipole orders, for which the first scheme performs better.

The third approach uses abstractions of the underlying octree and interaction patterns to allow for enhanced, efficient GPU utilization and it exploits the symmetries of the Multipole-to-Local operator. As a result, it scales perfectly with growing multipole order p maintaining a very good performance in the whole benchmarked multipole range $(0 < p < 20)$

Our FMM implementation has been optimized for biomolecular simulations and has been incorporated into GROMACS as an alternative to the established PME Coulomb solver (Kohnke et al., 2020a). We anticipate that, thanks to the inherently parallel structure of the FMM, future multi-node multi-GPU implementations will eventually overcome the PME scaling bottlenecks (Board et al., 1999; Kutzner et al., 2007, 2014).

Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This project was supported by the DFG priority programme *Software for Exascale Computing* (SPP 1648). A special thanks goes to Jiri Kraus from NVIDIA who supported this project in the early stage of its development and to R. Thomas Ullmann who took part in writing the FMM-GROMACS interface and unit tests.

ORCID iD

Bartosz Kohnke https://orcid.org/0000-0002-6000-5490
Carsten Kutzner https://orcid.org/0000-0002-8719-0307

References

Abraham MJ, Murtola T, Schulz R, et al. (2015) GROMACS: high performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX* 1–2: 19–25.

Agullo E, Bramas B, Coulaud O, et al. (2016) Task-based FMM for heterogeneous architectures. *Concurrency and Computation: Practice and Experience* 280(9): 2608–2629.

Allen M and Tildesley D (1989) *Computer Simulation of Liquids, 1987*. Oxford*:* Clarendon Press, p. 385.

Andoh Y, Yoshii N, Fujimoto K, et al. (2013) MODYLAS: a highly parallelized general-purpose molecular dynamics simulation program for large-scale systems with long-range forces calculated by fast multipole method (FMM) and highly scalable fine-grained new parallel processing algorithms. *Journal of Chemical Theory and Computation* 90(7): 3201–3209.

Andoh Y, Yoshii N and Okazaki S (2020) Extension of the fast multipole method for the rectangular cells with an anisotropic partition tree structure. *Journal of Computational Chemistry* 41: 1–15.

Arnold A, Fahrenberger F, Holm C, et al. (2013) Comparison of scalable fast methods for long-range interactions. *Physical Review E* 88: 063308.

Blanchard P, Bramas B, Coulaud O, et al. (2015) ScalFMM: a generic parallel fast multipole library. In: *SIAM Conference on Computational Science and Engineering (SIAM CSE 2015)*, Salt Lake City, USA, 18 March 2015

Board JA, Causey JW, Leathrum JF, et al. (1992) Accelerated molecular dynamics simulation with the parallel fast multipole algorithm. *Chemical Physics Letters* 1980(1): 89–94.

Board JA, Humphres CW, Lambert CG, et al. (1999) Ewald and multipole methods for periodic N-body problems. In: Deuflhard P, Hermans J, and Leimkuhler B, et al (eds) *Computational Molecular Dynamics: Challenges, Methods, Ideas*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 459–471.

Bock L, Blau C, Schröder G, et al. (2013) Energy barriers and driving forces in tRNA translocation through the ribosome. *Nature Structural & Molecular Biology* 20: 1390–1396.

Cheng H, Greengard L and Rokhlin V (1999) A fast adaptive multipole algorithm in three dimensions. *Journal of Computational Physics* 1550(2): 468–498.

Dachsel H (2010) An error-controlled fast multipole method. *The Journal of Chemical Physics* 132: 119901.

Dawson JM (1983) Particle simulation of plasmas. *Reviews of Modern Physics* 55: 403–447.

Ding H, Karasawa N and Goddard WA (1992a) Atomic level simulations on a million particles: the cell multipole method for Coulomb and London nonbond interactions. *The Journal of Chemical Physics* 970(6): 4309–4315.

Ding H-Q, Karasawa N and Goddard WA (1992b) The reduced cell multipole method for Coulomb interactions in periodic systems with million-atom unit cells. *Chemical Physics Letters* 1960(1): 6–10.

Dror RO, Dirks RM, Grossman J, et al. (2012) Biomolecular simulation: a computational microscope for molecular biology. *Annual Review of Biophysics* 41: 429–452.

Eichinger M, Grubmüller H, Heller H, et al. (1997) FAMUSAMM: an algorithm for rapid evaluation of electrostatic interactions in molecular dynamics simulations. *Journal of Computational Chemistry* 180(14): 1729–1749.

Engheta N, Murphy WD, Rokhlin V, et al. (1992) The fast multipole method (FMM) for electromagnetic scattering problems. *IEEE Transactions on Antennas and Propagation* 400(6): 634–641.

Essmann U, Perera L, Berkowitz M, et al. (1995) A smooth particle mesh Ewald method. *The Journal of Chemical Physics* 103: 8577.

Fong W and Darve E (2009) The black-box fast multipole method. *Journal of Computational Physics* 2280(23): 8712–8725.

Garcia AG, Beckmann A and Kabadshow I (2016) *Accelerating an FMM-Based Coulomb Solver with GPUs*. Berlin, Heidelberg: Springer International Publishing, pp. 485–504.

Gnedin NY (2019) Hierarchical particle mesh: an FFT-accelerated fast multipole method. *Astrophysical Journal Supplement Series* 2430(2): 19.

Greengard L and Rokhlin V (1987) A fast algorithm for particle simulations. *Journal of Computational Physics* 730(2): 325–348.

Greengard L and Rokhlin V (1997) A new version of the fast multipole method for the Laplace equation in three dimensions. *Acta Numerica* 6: 229–269.

Gumerov NA and Duraiswami R (2006) FMM accelerated BEM for 3D Laplace & Helmholtz equations. In: *Presentation for the International Conference on Boundary Element Technique BETEQ-7*, Paris, France, September 2006.

Gumerov NA and Duraiswami R (2008) Fast multipole methods on graphics processors. *Journal of Computational Physics* 2270(18): 8290–8313.

Hansson T, Oostenbrink C and van Gunsteren W (2002) Molecular dynamics simulations. *Current Opinion in Structural Biology* 120(2): 190–196.

Hess B, Kutzner C, van der Spoel D, et al. (2008) GROMACS 4: algorithms for highly efficient, load-balanced, and scalable molecular simulation. *Journal of Chemical Theory and Computation* 4(3): 435–447.

Hockney RW and Eastwood JW (1988) *Computer Simulation Using Particles*. Abingdon: Taylor and Francis, Inc.

Jones S (2012) Introduction to dynamic parallelism. In: *GPU Technology Conference Presentation*, Vol.338, March.

Knap M and Czarnul P (2019) Performance evaluation of Unified Memory with prefetching and oversubscription for selected parallel CUDA applications on NVIDIA Pascal and Volta GPUs. *The Journal of Supercomputing* 750(11): 7625–7645.

Kohnke B, Kutzner C and Grubmüller H (2020a) A GPU-accelerated Fast Multipole Method for GROMACS: performance and accuracy. *Journal of Chemical Theory and Computation*. Under review.

Kohnke B, Ullmann TR, Beckmann A, et al. (2020b) GROMEX—a scalable and versatile fast multipole method for biomolecular simulation. In: Bungartz H-J, Reiz S, Uekermann B, Neumann P, and Nagel WE, (eds) *Software for Exascale*

Computing—SPPEXA 2016–2019. Berlin, Heidelberg: Springer International Publishing, pp. 517–543.

Kurzak J and Pettitt BM (2006) Fast multipole methods for particle dynamics. *Molecular Simulation* 320(10–11): 775–790.

Kutzner C, van der Spoel D, Fechner M, et al. (2007) Speeding up parallel GROMACS on high-latency networks. *Journal of Computational Chemistry* 28(12): 2075–2084.

Kutzner C, Apostolov R, Hess B, et al. (2014) Scaling of the GROMACS 4.6 molecular dynamics code on SuperMUC. In: Bader M, Bode A, and Bungartz HJ (eds) *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*. Amsterdam: IOS Press, pp 722–730.

Kutzner C, Páll S, Fechner M, et al. (2019) More bang for your buck: improved use of GPU nodes for GROMACS 2018. *Journal of Computational Chemistry* 400(27): 2418–2431.

Lane TJ, Shukla D, Beauchamp KA, et al. (2013) To milliseconds and beyond: challenges in the simulation of protein folding. *Current Opinion in Structural Biology* 230(1): 58–65.

Lashuk I, Chandramowlishwaran A, Langston H, et al. (2009) A massively parallel adaptive fast-multipole method on heterogeneous architectures. In: *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, Portland, OR, USA, 14–20 November 2009. New York, NY: Association for Computing Machinery.

Nelson MT, Humphrey W, Gursoy A, et al. (1996) NAMD: a parallel, object-oriented molecular dynamics program. *The International Journal of Supercomputer Applications and High Performance Computing* 100(4): 251–268.

Nickolls J, Buck I, Garland M, et al. (2008) Scalable parallel programming with CUDA. *Queue* 60(2): 40–53.

Niedermeier C and Tavan P (1994) A structure adapted multipole method for electrostatic interactions in protein dynamics. *The Journal of Chemical Physics* 1010(1): 734–748.

Páll S and Hess B (2013) A flexible algorithm for calculating pair interactions on SIMD architectures. *Computer Physics Communications* 184: 2641–2650.

Páll S, Abraham MJ, Kutzner C, et al. (2015) Tackling exascale software challenges in molecular dynamics simulations with GROMACS. In: Markidis S, and Laure E, (eds) *Lecture Notes in Computer Science. EASC 2014*, Vol. 8759. Switzerland: Springer International Publishing Switzerland, pp. 1–25.

Patra M, Karttunen M, Hyvönen M, et al. (2003) Molecular dynamics simulations of lipid bilayers: major artifacts due to truncating electrostatic interactions. *Biophysical Journal* 840(6): 3636–3645.

Paul F, Wehmeyer C, Abualrous ET, et al. (2017) Protein-peptide association kinetics beyond the seconds timescale from atomistic simulations. *Nature Communications* 80(1): 1–10.

Potter D, Stadel J and Teyssier R (2017) PKDGRAV3: beyond trillion particle cosmological simulations for the next era of galaxy surveys. *Computational Astrophysics and Cosmology* 40(1): 2.

Salomon-Ferrer R, Götz AW, Poole D, et al. (2013) Routine microsecond molecular dynamics simulations with AMBER on GPUs. 2. Explicit solvent particle mesh Ewald. *Journal of Chemical Theory and Computation* 90(9): 3878–3888.

Schreiber H and Steinhauser O (1992) Molecular dynamics studies of solvated polypeptides: why the cut-off scheme does not work. *Chemical Physics* 1680(1): 75–89.

Schwantes CR, McGibbon RT and Pande VS (2014) Perspective: Markov models for long-timescale biomolecular dynamics. *The Journal of Chemical Physics* 1410(9): 090901–090907.

Shamshirgar DS, Yokota R, Tornberg A-K, et al. (2019) Regularizing the fast multipole method for use in molecular simulation. *The Journal of Chemical Physics* 1510(23): 234113.

Shaw DE, Grossman J, Bank JA, et al (2014) Anton 2: raising the bar for performance and programmability in a special-purpose molecular dynamics supercomputer. In: *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, New Orleans, LA, USA, 16–21 November 2014, pp. 41–53. New York, NY: IEEE.

Takahashi T, Cecka C, Fong W, et al. (2012) Optimizing the Multipole-to-Local operator in the fast multipole method for graphical processing units. *International Journal for Numerical Methods in Engineering* 89: 105–133.

Tough RJA and Stone AJ (1977) Properties of the regular and irregular solid harmonics. *Journal of Physics A: Mathematical and General* 100(8): 1261.

Volkov V (2010) Better performance at lower occupancy. In: *Proceedings of the GPU Technology Conference, GTC*, Vol. 10, San Jose, CA, p. 16.

White CA and Head-Gordon M (1996) Rotating around the quartic angular momentum barrier in fast multipole method calculations. *The Journal of Chemical Physics* 1050(12): 5061–5067.

Williams S, Waterman A and Patterson D (2009) Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM* 520(4): 65–76.

Ying L, Biros G and Zorin D (2004) A kernel-independent adaptive fast multipole algorithm in two and three dimensions. *Journal of Computational Physics* 1960(2): 591–626.

Yokota R, Narumi T, Sakamaki R, et al. (2009) Fast multipole methods on a cluster of GPUs for the meshless simulation of turbulence. *Computer Physics Communications* 1800(11): 2066–2078.

Yoshii N, Andoh Y and Okazaki S (2020) Fast multipole method for three-dimensional systems with periodic boundary condition in two directions. *Journal of Computational Physics* 410(9): 940–948.

Author biographies

Bartosz Kohnke is a PhD student at the Max Planck Institute (MPI) for Biophysical Chemistry in Göttingen in the department of Theoretical and Computational Biophysics. He has been working on efficient parallelization of the fast multipole method for molecular dynamics simulations. He received a master's degree in Applied Computer Science with a specialization in Scientific Computing from the University of Göttingen, Germany in 2012.

Carsten Kutzner received his PhD in physics from the University of Göttingen in 2003. Since 2004 he is a member of the Theoretical and Computational Biophysics department at the MPI for Biophysical Chemistry. He develops efficient methods for atomistic biomolecular simulations in HPC environments.

Andreas Beckmann studied computer science at Martin Luther University of Halle-Wittenberg. Aiming for performance portability, he specialized on the design and development of software abstractions for low-level hardware characteristics. Currently he works at the Jülich Supercomputing Centre in Germany.

Gert Lube received his PhD in 1980 and his Habilitation in 1989 from Otto von Guericke University Magdeburg. From 1993 until his retirement in 2019, he worked as a professor for Applied Mathematics at the Institute for Numerical and Applied Mathematics of the Georg-August University Göttingen. His main interests are the numerical solutions of partial differential equations, in particular in fluid mechanics.

Ivo Kabadshow studied electrical engineering at Chemnitz University of Technology. During his PhD in applied mathematics at University of Wuppertal he specialized on fast multipole methods for HPC applications. Currently he works at Jülich Supercomputing Centre on FMM-based libraries for massively parallel HPC systems.

Holger Dachsel studied Chemistry at the University of Leipzig. After receiving his PhD, he worked at the University of Vienna, at PNNL in Richland, WA, at the Free University Amsterdam, and at Q-Chem. Inc. in Pittsburgh, PA. His area of expertise at Jülich Supercomputing Centre is the fast multipole method, especially its mathematical aspects.

Helmut Grubmüller is scientific member of the Max Planck Society and director at the MPI for Biophysical Chemistry where he leads the department of Theoretical and Computational Biophysics. He is also honorary professor for physics at the University of Göttingen. His research aims at an understanding of the physics and function of proteins, protein complexes and other biomolecular structures at the atomic level.