A New Spin on the Fast Multipole Method for GPUs: Rethinking the Far-Field Operators

Arijus Lengvenis
Jülich Supercomputing Centre
Jülich, Germany
a.lengvenis@fz-juelich.de

Holger Dachsel
Jülich Supercomputing Centre
Jülich, Germany
h.dachsel@fz-juelich.de

Laura Morgenstern
Durham University
Durham, UK
l.morgenstern@durham.ac.uk

Ivo Kabadshow
Jülich Supercomputing Centre
Jülich, Germany
i.kabadshow@fz-juelich.de

2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 979-8-3315-3237-6/25/$31.00 ©2025 IEEE | DOI: 10.1109/IPDPS64566.2025.00075

***Abstract*—The Fast Multipole Method (FMM) is an optimally efficient algorithm for solving N-body problems: a fundamental challenge in fields like astrophysics, plasma physics and molecular dynamics. It is particularly suited for computing 1/r potentials present in Coulomb and gravitational particle systems. Despite the near-field phase being trivially parallelisable, the far-field phase of the 1/r FMM currently lacks an efficient, massively parallel GPU algorithm fitting for the era of Exascale computing. Current state-of-the-art approaches either favor highly parallel but inefficient expansion shift operators or asymptotically efficient but poorly parallelisable rotation-based ones. Recently, a breakthrough was made with the re-evaluation of a rotation operator variant called fast rotation, which dramatically increases caching effectiveness and marries the advantages of both methods. Thus, this paper incorporates this approach to create fast rotation-based operators that facilitate an efficient far-field algorithm for the FMM on GPUs. Additionally, a warp-centric data access scheme is co-developed alongside a matching octree design, which yields coalesced memory access patterns for the bottleneck operators of the far-field phase. The fast rotation algorithm is enhanced with a cache-tiling mechanism, maximising GPU cache utilisation. Compared to the state-of-the-art GPU FMM far-field implementation, our algorithm achieves lower running times across the board and a 2.47x speedup for an increased precision simulation, with the performance improvement growing as precision increases, providing concrete proof of efficacy for dense particle systems.**

***Index Terms*—Fast Multipole Method, GPU Programming, Parallel Algorithms**

I. Introduction

A. Motivation

THE Fast Multipole Method (FMM), hailed as one of the top ten algorithms of the 20^{th} century [1], [2], has revolutionised the field of N-body simulations, which aim to compute forces and potential fields within particle systems. Since its conception in the late 1980s, FMM has become a staple in computational physics and engineering, used in a vast array of topics from molecular dynamics [3], [4] to astrophysics [5], [6]. It achieves the optimal $\mathcal{O}(N)$ time complexity coupled with a memory-efficient octree data structure and error-control parameters set a priori with support for open or periodic-boundary conditions for flexible and accurate simulations [7], [8]. Finally, this algorithm only requires managing localised spatial dependencies unlike other methods, which have to perform expensive all-to-all operations every timestep [9]. These properties make it an incredible candidate for modern-day high performance computing (HPC) systems for running a large variety of inputs. One of the most ubiquitous use cases for this algorithm is simulating $1/r^k$ potentials between particles generalised for any k, such as the Coulomb or gravitational interactions [10]. For this set of problems, the FMM can be optimised further, achieving higher performance thresholds than for the general case [11]–[13]. Thus, many major molecular dynamics (MD) libraries utilise the 1/r FMM for this purpose [3], [14]. Nevertheless, one of the most prominent bottlenecks of current designs of the 1/r FMM for MD or plasma physics simulations, especially for GPU computing, is the subpar strong scaling performance of the far-field phase compared to Fast Fourier Transform (FFT)-based methods like PME [4], [15]–[20]. Thus, this paper aims to mitigate this issue.

Current research reveals two popular variants of the 1/r FMM far-field phase: the *P4 shift* which has a $\mathcal{O}(p^4)$ prefactor time complexity (the complete time complexity is $\mathcal{O}(Np^4)$) but is massively parallelisable [22], and the *rotation-based*, which possesses a more efficient prefactor time complexity of $\mathcal{O}(p^3)$ [23], but does not lend itself to parallelisation effectively. Here p is the multipole order parameter which directly affects the accuracy as well as energy conservation properties of the simulations, which is important for many use cases [3]. Unfortunately, rotation-based FMM does not scale well in a parallel setting for dense systems due to the need to compute, store and load many expensive and large Wigner matrices, cannot be effectively stored in cache and thus introduce a significant overhead [25]. This problem is further exacerbated when considering GPU implementations, leading to severe memory bottlenecks during runtime, and leaving hardware underutilised. Thus, GPU implementations of this approach ended up with a high crossover point with optimised versions of P4 shift FMM [4], [21], [24].

Hence, the *fast rotation-based* operators are a crucial development in addressing this problem [13], [37]. This approach adapts the classical rotation algorithm to use only a single Wigner matrix without affecting the prefactor complexity which can be pre-computed before runtime. Consequently, this solves the major bottleneck of rotation-based FMM, as fitting and reusing a single universally applicable Wigner matrix in cache frees a large fraction of it to be used elsewhere, which paves the way for an efficient and strong scaling algorithmic

1530-2075/25/$31.00 ©2025 IEEE
DOI 10.1109/IPDPS64566.2025.00075

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on August 26,2025 at 20:40:27 UTC from IEEE Xplore. Restrictions apply.

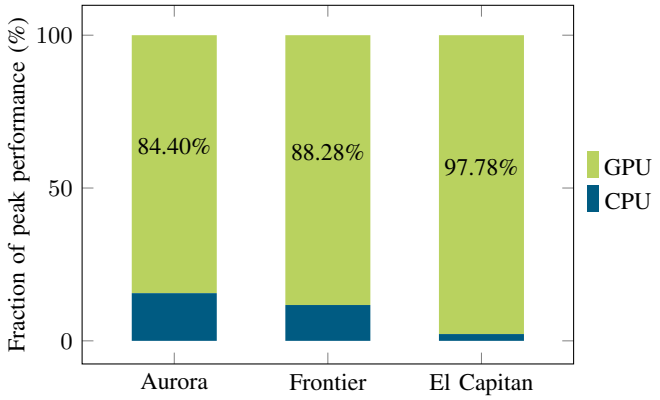


Fig. 1: The compute distribution of three supercomputing clusters with GPU nodes selected from the top five of the TOP500 list from November 2024 [35]. The vast majority of floating-point performance comes from the GPU hardware.

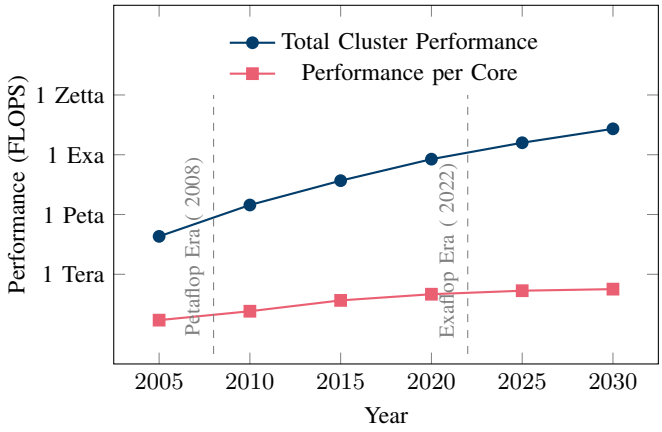


Fig. 2: The trend of Moore's law compared to the total performance in HPC clusters. Performance per core (or *vertical scaling*) fails to grow fast enough to meet demand, thus, HPC clusters employ many more cores (or *horrizontal scaling*) to keep up.

implementation for the GPU.

Given the substantial floating-point (FP) performance of modern GPUs, optimising the FMM for these platforms could unlock a generational leap in computational capability. This is particularly important as modern HPC systems are beginning to adopt many more GPU nodes. In some cases, nearly 98% of their double precision FP performance capabilities exist in the form of GPU resources, which can be seen in Fig. 1, which increases to 99% when considering single precision. The need for increasingly powerful supercomputing systems continues to grow, especially in light of modern AI research, however, due to the decline of Moore's law, this increase is maintained through horizontal scaling rather than vertical, as seen in Fig. 2. This leads to a large demand for powerful clusters with many GPU nodes, which necessitates massively parallel and efficient GPU algorithm research, a fact that is further confirmed by emerging massive collaborative initiatives like ExCALIBUR [36] and Gromex [26].

B. Research Objective & Contributions

This project addresses a few key problems within state-of-the-art research in far-field FMM algorithms, with the main research objective being *enabling an efficient and massively parallel* $1/r^k$ *potential far-field algorithm using fast rotation-based operators on GPUs by following intra-SM optimisation*. *Intra-SM*, or intra-streaming multiprocessor, optimisation entails improving the strong scaling performance within the parallelisation hierarchy of a single SM. Unlike *inter-SM* optimisation, which would focus on multi-GPU and communication framework designs for the algorithm, this paper focuses on optimising hardware utilisation for a single GPU without introducing any inter-SM dependencies so that, in the future, it could be independently parallelised to as many GPUs as necessary. Therefore, the goal of this paper will be to optimise the memory access patterns and layout to minimise memory and latency bottlenecks while maintaining the optimal prefactor time complexity.

The efficiency of the solution would be quantitatively evaluated and compared against the state-of-the-art GPU FMM implementation [4] via execution time. Finally, CUDA was chosen as the programming language due to its maturity and explicit control over low-level concepts like warps [29]. This provides tighter control over the underlying hardware when implementing the algorithm, potentially leading to greater performance. Finally, in some cases, resources exist which can transpile the code into different architectures like HIP, making the code more portable [40], [41].

The novel contributions introduced by this paper are as follows:

- A GPU-optimised 1/r FMM far-field implementation for dense systems with periodic-boundary conditions written in CUDA.
- A "hybrid-centric" memory access algorithm for the fast-rotation operators, which minimises memory fetches while enabling branchless load-balanced thread execution.
- A new dense octree design complementing the hybrid-centric access model, affording grid-agnostic contiguous memory access.
- A cache-tiled fast rotation algorithm fully utilising the CUDA cache memory hierarchy.

II. Background & Related Work

A. Fast Multipole Method Background

The FMM is structured into two parts: a near-field phase and a far-field phase. The near-field phase, often referred to as the P2P (particle-to-particle) operator, computes interactions via direct particle-particle summation, handling the spatially close pairs of particles where far-field approximations would not converge, leading to undefined behaviour. The far-field

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on August 26,2025 at 20:40:27 UTC from IEEE Xplore. Restrictions apply.

phase, on the other hand, relies on hierarchical multipole and local expansions encoded in an octree data structure, ensuring that interactions between well-separated regions of space can be used to approximate all other interactions efficiently.

Formally, the $1/|\mathbf{a}-\mathbf{b}|$ potential can be expanded into spherical harmonic representations of multipole (ω_{lm}) and local (μ_{lm}) expansions [32]:

$$\frac{1}{|\mathbf{a}-\mathbf{b}|} \approx \sum_{l=0}^{p} \sum_{m=-l}^{l} \omega_{lm}(\mathbf{a})\, \mu_{lm}(\mathbf{b})\,, \tag{1}$$

where p is the order of the expansion and ω_{lm}, μ_{lm} are basis functions of multipole and local expansions respectively, dependent on position vectors $\mathbf{a}, \mathbf{b}$ and expansion indices l, m. These expansions can be factorised as:

$$\omega_{lm}(\mathbf{a}) = \frac{a^l}{(l+m)!} P_{lm}(\cos\alpha) e^{-im\beta}\,, \tag{2}$$

$$\mu_{lm}(\mathbf{b}) = \frac{(l-m)!}{b^{l+1}} P_{lm}(\cos\theta) e^{im\phi}\,, \tag{3}$$

where P_{lm} are associated Legendre polynomials corresponding to rotations around the y-axis, and the exponential terms encode rotations in the xy-plane. The angles (α, β) and (θ, ϕ) correspond to coordinate rotations that align the expansions between source and target boxes.

Due to the relative algorithm optimisation complexity, a significant body of research currently exists contributing to efficient direct summation algorithms present in P2P on GPUs [33], [34]. The proportion of work that P2P is allocated is controlled by varying the octree depth or decreasing the well-separateness parameter `ws`, which defines the minimum distance where octree boxes are considered to be part of the local neighbourhood. Given that the far-field phase lacks a strong-scaling design, most users shift the majority of the work towards P2P to get the best wall-clock performance. The optimal parameter set for any given input for a certain precision can be computed beforehand by giving the number of FLOPs [32], which cannot be utilised due to the imbalance in hardware utilisation between the phases. Therefore, having a highly efficient far-field algorithm could improve the efficiency of the overall FMM algorithm by balancing the workload between both phases.

For the scope of this paper, the far-field design will focus on $1/r^k$ FMM, and the simulation space will be represented as a dense octree with periodic-boundary conditions, where all expansions part of the octree are stored in memory. The latter assumption significantly simplifies any parallel algorithm by making the octree resemble a hierarchical grid while maintaining the usefulness of the solution for homogeneous inputs, popular in MD and plasma physics [27]. With a dense octree, the overall time and space complexity of the far-field stage is $\Theta(N+8^d)$, where 8^d represents the number of boxes on the lowest level of the octree. Since each box is considered at most a constant number of times by each operator, it is more intuitive to analyse the far-field operators while considering the prefactor time complexity per box defined in terms of the multipole order p. Considering practical use cases, we claim that $6 \leq p \leq 18$ is enough for accurate simulations.

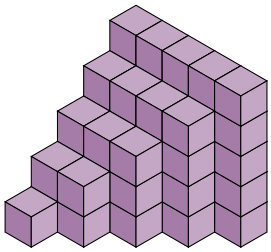


Fig. 3: The Wigner matrix data structure is used for performing rotations, where each cube represents a real number.

B. Rotation-based FMM

The FMM operators can be split into two groups – particle-based operators (i.e. P2M and L2P, P for particle, and M and L for multipole and local expansions, respectively) as well as the E2E (expansion-to-expansion) operators (i.e. M2M, M2L and L2L). The particle-based operators have a relatively low $\mathcal{O}(p^2)$ prefactor time complexity, while the E2E operators have either a $\mathcal{O}(p^4)$ or $\mathcal{O}(p^3)$ prefactor time complexity, depending on the underlying algorithm. Each E2E operator aims to achieve the same goal – re-expand a source expansion at the centre of the target box. This process is described by the shifting (or translation) algorithm, which defines this procedure for a multipole or local moment. The P4 shift algorithm considers every term in the source expansion and performs a convolution operation with a $\mathcal{O}(p^2)$ sized operator. This implies a direct multiplication of terms, leading to quadratic complexity in the number of terms in an expansion, i.e. $\mathcal{O}(p^4)$ prefactor complexity. However, there exists a special case where the size of the operator collapses to a $\mathcal{O}(p)$ size, and the algorithm improves to a $\mathcal{O}(p^3)$ prefactor time complexity. This occurs when the z-axis of the source expansion aligns with the target's centre. Thus, expansion rotation generalises this idea to align the z-axis of the source expansion towards any point in space to afford significantly cheaper translation operations [23].

This rotation requires a z-axis rotation and a y-axis rotation in spherical coordinates. The y-axis rotation is represented as large Wigner matrix data structures, which can be seen in Fig. 3. To perform a single translation operation, two Wigner matrices have to be computed, corresponding to the forward and backward angle rotation. Since these data structures have $\mathcal{O}(p^3)$ memory complexity, storing many of them in cache becomes practically infeasible. This motivates expansion binning, as some expansions will undergo the same rotation relative to their target [25], amortising the cost of computing and storing the large data structures in cache. However, M2L requires a large number of shifts to be performed for each source expansion, as each source can be part of many different target expansions, which results in many redundant reads and writes with this process to maintain an active cache. Therefore, even by optimising the order of operations and cache-tiling, fitting enough expansions and their associated Wigner matrices

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on August 26,2025 at 20:40:27 UTC from IEEE Xplore. Restrictions apply.

in cache results in overflow or thread divergence, which causes significant memory congestion and warp stalling, respectively, leading to a poorly scaling algorithm. Consequently, the strong scaling for the GPU implementations using this approach plateaus quickly as the kernels become severely memory-bound.

The advent of fast rotation aims to utilise a single Wigner matrix called the *flip operator* [37] to perform any arbitrary rotation with two 1-dimensional rotations instead of a single 2-dimensional one. The flip operator's angle is set to $\frac{\pi}{2}$, which results in roughly half of the values becoming zero, effectively halving the size of the data structure, thus the number of FLOPs required to rotate an expansion. Hence, despite this requiring two z- and y-axis rotations to achieve any arbitrary angle, due to the low complexity and overhead of z-rotations, this equates to roughly the same number of FLOPs. Furthermore, this data structure can be pre-computed at compile-time and used for rotating all expansions universally simultaneously for both forwards as well as backwards rotations. This new spin on rotation has already inspired research into how it could be used for designing massively parallel GPU rotation-based FMM operators [13]. The resulting M2M operator was highly scalable and performant; however, it was a toy example without utilising an octree or managing the implicit dependencies defined by the operators, especially M2L.

C. Current GPU Implementations

As previously mentioned, the latest research for GPU FMM algorithms revolves around P4 shift [4], [21] and rotation-based FMM [25]. There are also kernel-independent solutions which sacrifice some performance optimisation for generalisability of potentials [9], [38], [39]. Some of these have $\mathcal{O}(p^3 \log p)$ prefactor time complexity kernels [22], which have a higher complexity than fast rotation and introduce additional errors which the error control cannot account for [32]. For 1/r FMM, there exists a $\mathcal{O}(p^2 \log p)$ time complexity FFT-based kernel [11], but it introduces additional errors, which removes the ability to compute a strict error bound on the computation. Finally, modern research is focusing on AI-based simulation models, which offer orders of magnitude speedup for inputs during inference [30], [31]. Unfortunately, no error bounds exist for these solutions, and this undermines the accuracy of long-horizon simulations, which makes it less useful for scientific applications.

Garcia et al. have attempted an implementation using rotation-based FMM in CUDA [25]. However, the crossover point in terms of p for the performance improvement compared to the P4 shift FMM is relatively high. Thus, the current state-of-the-art implementation, which is also available through GROMACS [3], [28], is by Kohnke et al. [4]. It heavily optimises the P4 shift algorithm, providing intra-expansion and inter-expansion parallelism via dynamic parallelism, which scales very well. However, given the high time complexity of this algorithm, the implementation still slows down significantly as p increases. Regarding a fast rotation-based FMM, there are very few implementations of this, even on CPU. This paper has chosen a library called FMsolvr that has designed a 1/r FMM with fast rotation operators [12]. This solution utilises the new flip operator with SIMD vectorisation, which is achieved by reorganising expansions on the fly. This design has become the foundation for an efficient CUDA-based fast rotation algorithm because as far as this paper is aware – there are currently no fast rotation-based GPU implementations of the far-field phase of the FMM.

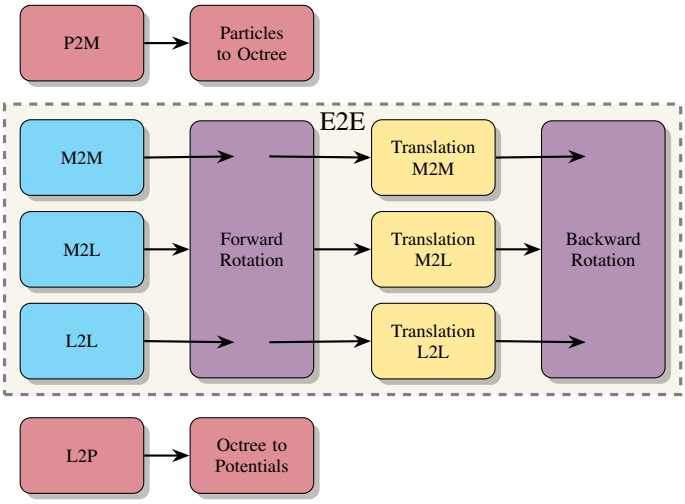


Fig. 4: Architecture diagram describing the rotation-based far-field phase of the FMM. The fast rotation algorithm affords using the same Wigner matrix for all E2E operators, enabling optimisations which will be introduced and exploited throughout the proposed methodology.

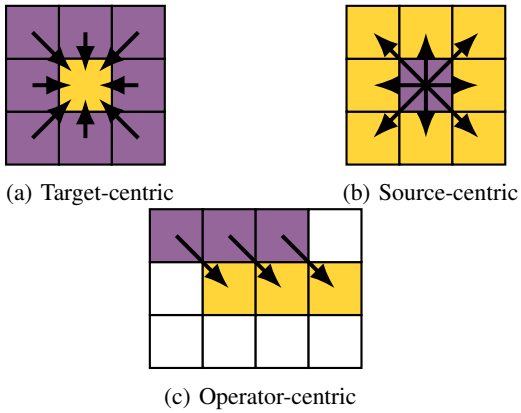
III. Methodology

This section will discuss the design process of the optimised memory layout and access model for the E2E operators enabled by fast rotation. A high-level architecture diagram of the algorithm can be seen in Fig. 4, which highlights how data is processed by each operator and showcases how fast rotation improves the generalisability of the E2E operators.

A. Memory Access Models

The first and most crucial aspect to consider is the memory access pattern for all threads within the context of the E2E operators. Parallelising the fast rotation algorithm with respect to a single expansion can lead to work imbalance between threads as well as additional communication required to synchronise data dependencies for different parts of the routine. As such, this paper has chosen thread-independent expansion processing, where any single thread is responsible for its unique set of expansions. The downside of this parallelisation pattern is that for low depths or shallow octrees where there are few expansions to work with, hardware could become underutilised. However, given that the number of elements per level increases exponentially, this downside becomes marginal when the octree has $d > 3$ by utilising operator overlapping.

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on August 26,2025 at 20:40:27 UTC from IEEE Xplore. Restrictions apply.



(a) Target-centric (b) Source-centric

(c) Operator-centric

Fig. 5: Illustration of the different data access parallelisation approaches. Each one optimises a different aspect of the operator - minimising writes (a), minimising reads (b) and maximising contiguous memory accesses (c).

This optimisation increases the amount of independent tasks concurrently available to the GPU due to M2L having at least 189 source-target pairs to compute per box for $\texttt{ws} = 1$.

The fast rotation algorithm takes a source expansion and a target expansion as input. For the M2M operator, the source expansions are multipole expansions of boxes (denoted ω) on level i that are re-expanded at the center of their direct ancestor ω on level $i-1$ within the octree. Thus, the input boxes can be referred to as child boxes, which are shifted towards the center of the target parent box residing on the level above within the octree and accumulated together. Conversely, L2L performs the opposite operation of distributing the parent local expansions (denoted μ) on a level i to their children boxes μ on level $i+1$. Finally, for M2L the source expansions are the well-separated local neighbourhood of ω on level i that need to be translated to a target box local expansion.

Given that each E2E operator can be abstracted to performing fast rotation operations from a set of sources to a set of targets, three unique parallelisation paradigms that optimise the access patterns for each one emerge. Let us call an access pattern *target-centric* when the sources belonging to the same target expansion are processed concurrently, which can be seen in Fig. 5a. This places the first parallelisation loop over the target expansions and then the inner one over the source boxes from which the associated target expansions are taken. Let us call an access pattern *source-centric* when the targets to which a source belongs are processed concurrently, as can be seen in Fig. 5b. This is the analogous opposite of target-centric by swapping the parallelisation loops. Finally, let us call an access pattern *operator-centic* where the same relative source expansion per target expansion (or vice-versa) is processed concurrently, which can be seen in Fig. 5c. This is done by iterating over all unique relative source-target transformations for each source or target box. This access pattern exploits the symmetrical property that all input boxes undergo the same fast rotation operations within their spatial neighbourhood, which is ensured by assuming a dense octree with periodic-boundary conditions.

Intuitively, the target-centric approach is ideal for the M2M operator, which would allow for all child ω part of the same parent ω to be processed concurrently, accumulated in cache and written in one global memory transaction, thus minimising memory congestion. An analogous argument can be made for the L2L operator with the source-centric model, as this would explicitly cache the parent μ, allowing for efficient reading of the data. Lastly, the M2L operator favours both target- and source-centric memory access models, as both could be applied concurrently for minimising the total number of global memory accesses. However, it would become infeasible to fit all partially completed expansions at the same time in cache, as this would scale proportionally with $\mathcal{O}(4^d)$. Nevertheless, the performance of a purely source-centric or target-centric M2L would have to be assessed empirically.

B. Hybrid-centric Access Algorithm

The downside of using either a target-centric and source-centric approach for M2L (and to a lesser extent M2M and L2L) is that it could potentially cause non-coalesced memory accesses, as there are at least 189 source expansions that undergo a translation with respect to any target, when $\texttt{ws} = 1$. Each translation follows the same fast rotation algorithm with a different set of z-rotation angles as well as translation operator coefficients stored in auxiliary arrays. Given the small memory footprint of these data structures, these can easily fit into shared memory, however, they can still contribute towards memory congestion due to bank conflicts. However, it is nearly impossible to access all input sources for all target expansions in a coalesced manner in global memory for a given space-filling curve (SFC). Therefore, this will lead to a severe memory bottleneck if not addressed.

The solution to this is to use the operator-centric model, which, if combined with an appropriate octree memory layout, would provide coalesced memory accesses. However, this approach implemented naïvely can come at the cost of vastly increased memory congestion, as every source and target expansion would be read and accumulated to 189 times, respectively, without benefiting from caching. This raises the question: *does there exist a model which combines the memory access efficiency of either target-centric or source-centric with the data locality of operator-centric for all E2E operators?*

To answer this question, we introduce the *hybrid-centric* model, displayed in high-level pseudocode in Fig. 6. This model aims to combine the benefits of two of the previously mentioned approaches, namely operator-centric as well as either target-centric or source-centric. The algorithm at line 1 computes the relative source (or target) neighbourhood stencil for the operator, i.e. a relative set of points describing the box centers of all boxes part of the operator-specific interaction set. Since all E2E operators follow this pattern, it affords a way to generalise and describe all source-target box interactions for any operator. The thread block is subdivided into warps by explicitly storing the lane and warp IDs for each thread

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on August 26,2025 at 20:40:27 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1 Hybrid-Centric Memory Access Algorithm for M2L (Target-Centric Variant)

```
1:  sourceKernel ← compute_neighbourhood_kernel(id)
2:  numSources ← len(sourceKernel)
3:  laneId ← threadId % warpSize
4:  warpId ← threadId / warpSize
5:  numWarps ← blockDim / warpSize
6:  for i ← laneId to n step warpSize do
7:      targetExpansion ← initialise_empty_mu()
8:      for j ← warpId to numSources step numWarps do
9:          sourceExpansion ← fetch_omega(i, sourceKernel[j])
10:         fast_rotation(targetExpansion, sourceExpansion)
11:         sharedMem[laneId] ← accumulate(targetExpansion)
12:     end for
13:     write_mu(i, sharedMem[laneId])
14: end for
```

Fig. 6: High-level pseudo-code describing the architecture of the hybrid-centric (target-centric variant) E2E operator design. Lanes within a warp process the same target μ expansion, but each warp processes a unique source ω expansion relative to the target. This causes threads within a single warp to behave in a *operator-centric* way, as every thread part of the same warp will be working on the same relative translation. On the other hand, all warps combined behave in a *target-centric* manner, concurrently processing many source expansions with respect to the same target.

(lines $3-5$). This is then used to iterate over `warpSize` batches of boxes, where each lane would be working on the same relative source translation to their target (line 6), while each warp would process a unique set of source expansions from the neighbourhood stencil (line 8). The batch of target expansions can thus be cached in shared memory to enable highly efficient accumulation operations before flushing the completed μ to global memory.

This results in all threads within a warp working on expansions that undergo the same relative transformation, as in the operator-centric model, and all threads within warps corresponding to the same lane working on the same target (or source) expansion, as in target-centric (or source-centric). This design is ideal, as locally operator-centric accesses ensure coalesced memory operations for the input expansions, while globally target/source-centric approach reduces the number of redundant memory accesses by caching and reusing relevant data as well as ensures that all threads in a warp will access the same rotation and translation data leading to no bank conflicts. This access pattern retains independent warp-level thread execution for the fast rotation algorithm and affords efficient reduction routines to aggregate data. For operators where the neighbourhood stencil is small, like for M2M and L2L, the definition of `warpSize` can be adapted to allow multiple warps to span a single batch, increasing the number of source-target pairs to be processed per iteration. This makes the hybrid-centric algorithm flexible for any E2E operator and grid combination. Thus, this framework brings together a best-of-both-worlds design while not overburdening the shared memory resources, as well as minimising stalling overhead by naturally vectorising warp-wise execution.

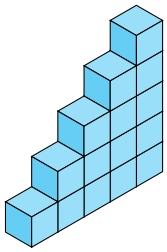


Fig. 7: The expansion data structure for $p = 4$. Each block represents a complex number.

C. Octree Design

Current implementations of the FMM use complex-valued triangular matrix data structures to represent both types of expansions in a uniform way, showcased by Fig. 7. This allows a general form of fast rotation to be employed, which can take either multipole or local expansions as input and processes them indiscriminately, excluding the special translation algorithm associated with each operator.

The expansion data structure is comprised of complex coefficient values, each needing to be processed in a specific way. However, elements corresponding to the same index within any expansion undergo the same transformations. Therefore, the data access pattern would benefit from transposing the containers to have elements with the index adjacent in memory. Some existing libraries implement a stacked transposed expansion memory layout [13], where a batch of expansions the size of the SIMD width of the system is reordered on the fly for efficient cache line utilisation. However, the necessity to actively perform these reorderings to facilitate vectorised operations incurs overhead, which can be avoided.

Instead of batching the expansions during runtime, our solution would initialise all elements in the transposed format on all levels of the octree. Therefore, any contiguous subset of expansions would be accessible in a coalesced manner without doing any additional work. Another advantage of this is that this memory layout is grid size agnostic, which enables the flexibility to adapt the grid between each level of the octree for each operator independently.

The next design aspect concerns the ordering of the expansions within any level of the octree dictated by the SFC. This is crucial as this affects the efficiency of cache line reads from global memory. The goal is to align the memory layout with the access patterns of the operator-centric access scheme to maintain all of the benefits that come with it.

Thus, this paper introduces the child-centric slabs SFC ordering, illustrated in Fig. 8. This expansion ordering utilises the important property of slab-wise ordering: given a slab-wise array A, a contiguous sub-array of elements $a \in A$ of size n, any other contiguous sub-array $b \in A$ of size n will be element-wise equidistant. That is:

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on August 26,2025 at 20:40:27 UTC from IEEE Xplore. Restrictions apply.

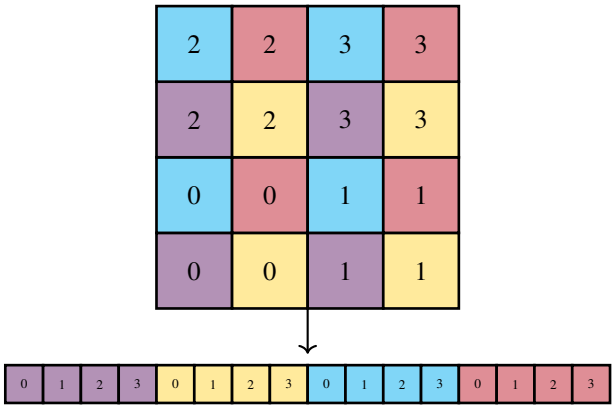


Fig. 8: Memory layout of the octree within memory compared to the spatial layout. Each expansion corresponding to a unique child relative to its parent (colour) is placed contiguously in memory sorted slab-wise by its parent box index.

$$d(a_0, b_0) = d(a_1, b_1) = \cdots = d(a_n, b_n) \tag{4}$$

where d is the distance between any two elements in memory. This property enables coalesced memory access for operator-centric access patterns, as due to this property any contiguous set of target (or source) expansions, the relative source (or target) expansions will thus also be adjacent in memory. To the best of our knowledge, there is no other SFC which exhibits this property for all boxes other than slabs or a variation of slabs.

Furthermore, the reason for splitting each child box into separate sub-arrays is related to the asymmetry of the M2L operator. The neighbourhood stencil for any box in M2L depends on the relative positioning of the child box with respect to the centre of the parent box. As such, applying operator-centric on a slab-wise ordering can yield uncoalesced memory access due to boxes corresponding to the same transformation being in vastly different locations in memory. The solution to this would be to split the eight children for all parent boxes and process them as independent operators. This child-centric split of the octree maintains the aforementioned slabs property while also ensuring efficient M2L execution when considering non-periodic-boundary accesses. Therefore, the sum of all the benefits mentioned above is maintained at little to no additional computational, memory or synchronisation overhead.

D. Cache Hierarchy Optimisation

The fast rotation algorithm is composed of three distinct parts: z-rotation, y-rotation and a translation, described in a high-level in Fig. 9. For each iteration, a source expansion undergoes two z-rotations and two y-rotations in an alternating order. Afterwards, translation is performed, after which the reverse set of z and y rotations is done to restore the alignment. The auxiliary data structures that facilitate the rotations and translation are pre-computed and stored in constant memory

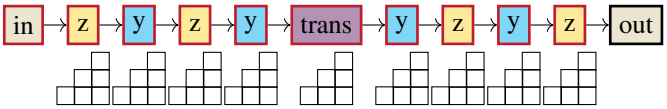


Fig. 9: High-level depiction of the fast rotation algorithm. Each box represents a step in the fast rotation algorithm. All steps are performed by reading the entire expansion (represented by triangles below each step) directly from global memory and storing the result there. A red outline denotes that the data is read from global memory.

or cached in shared memory. For the z-rotations, two angles ϕ and θ are computed from the relative distance vector R, pointing from the centre of the source box to the centre of the target box, and stored in constant memory. These angles are used to compute $e^{ik\phi}$ and $e^{ik\theta}$ on the fly for $0 \leq k \leq p$. Importantly, the same angles can then be used in complex conjugate form to rotate the expansion backward. The translation operators are vectors of size $p+1$ and $2p+1$ for M2M/L2L and M2L respectively, that utilises the relative distance vector R and is computed in the recursive form $t_n = \frac{Rt_{n-1}}{n}$ and $\frac{nt_{n-1}}{R}$ respectively. These vectors are stored in constant memory for M2M/L2L and in shared memory for M2L. Lastly, the flip operator is fetched from a pre-computed constant memory table at compile-time, which is loaded into the constant cache at run-time that is optimised for broadcast access patterns. However, it should be noted that modern GPUs have a constant cache size of 64KB, so careful consideration of data size requirements must be made.

For the hybrid-centric algorithm, we can also store the neighbourhood stencil maps in constant memory. These denote the relative coordinates of each neighbour with respect to the target box. For M2M and L2L, this neighbourhood can be simplified without the need to store the neighbourhood stencil, as every child box belonging to the same parent box will be evenly spaced out within the child-centric SFC. For M2L, storing all 316 relative 3D coordinates is necessary corresponding to the union of all M2L neighbourhood stencils. Furthermore, to be able to effectively access the angles and distance vectors, pointer maps for each index within the neighbourhood stencil must also be initialised. Overall, combining everything together results in about 62KB of constant memory for running the far-field phase up to $p = 17$ for $\texttt{ws} = 1$. This could be further optimised in the future by reducing the flip operator, computing it on the fly or moving it to texture memory instead.

E. Cache-Tiling Mechanism

The effective utilisation of L1 cache via shared memory is crucial for reducing latency bottlenecks within any kernel. The fast rotation algorithm involves reading each expansion coefficient $\mathcal{O}(p)$ times, which, if cached effectively, could significantly increase compute throughput. Therefore, research was made into efficient cache utilisation for storing expansion data for every thread.

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on August 26,2025 at 20:40:27 UTC from IEEE Xplore. Restrictions apply.

Storing a single expansion data structure of order p requires

$$2r\frac{(p+1)(p+2)}{2} - r(p+1) = r(p+1)^2 \tag{5}$$

bytes of memory, where r is the number of bytes per floating point variable. There are $p+1$ rows and columns of complex values in the triangular matrix, except for the first row, which is populated only with real values. Given that each thread works with expansions independently, the memory required to store everything simultaneously is given by $2tr(p+1)^2$ where t is the number of threads per SM. Additionally, the shared memory buffer used to cache a batch of expansions in the hybrid-centric algorithm requires an additional $wr(p+1)^2$, where w is the warp size. Hence, the total memory requirement becomes:

$$r(2t+w)(p+1)^2. \tag{6}$$

Assuming a block size of 256 threads, a warp size of 32 and a multipole order of $p = 16$, the total bytes required per block would be $628,864$ bytes, significantly exceeding the L1 cache capacity of any currently existing GPU. This limitation implies that storing full expansion data structures in shared memory concurrently for all threads is impossible. However, this limit could be further improved by processing the expansions piece-wise.

It is important to notice that the y-rotation requires only column-wise dependencies for rotating an expansion, implying that only a single column's values can be loaded in cache. Additionally, since the z-rotation operates element-wise, it introduces no extra dependencies, and in cases where a y-rotation is immediately followed by a z-rotation, these can be combined into one to save on shared memory bandwidth. Finally, the translation step for all operators requires row-wise dependencies, which necessitates having additional auxiliary global memory arrays to store the forward-rotated and translated expansions to transition from a column-wise to row-wise and back to column-wise cache-tiling approaches, respectively. Consequently, we only require a column vector the size of the largest expansion column/row for running the entire fast rotation routine, which reduces the shared memory cost from quadratic in p to linear:

$$r(2t+w)(2p+1). \tag{7}$$

This optimisation reduces the shared memory requirements to $71,808$ bytes for the previous example, which is manageable by most modern GPUs.

The two auxiliary global memory buffers for storing intermediate expansions incur a small enough memory footprint that they can at least fit into L2 cache during execution, reducing latency further. The inner loops of each step can now be handled entirely in shared memory utilising column vectors, which can be switched to global memory when the amount of shared memory required exceeds the available L1 cache resources. This fast rotation memory model redesign, visualised in Fig. 10, not only brings a sizeable speedup when enough shared memory is present but also improves cache utilisation in the general case.

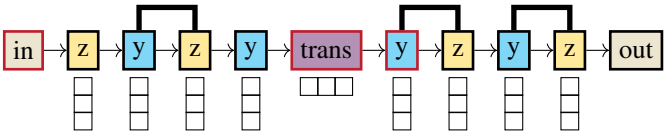


Fig. 10: Optimised fast rotation algorithm by exploiting the CUDA cache memory hierarchy. The connected boxes with a thick black line imply that they can be condensed into one, reducing the number of shared memory operations.

TABLE I: Default parameters for the benchmarking process.

Parameter		**Value**
p	Multipole order	11
d	Depth	7
ws	Well-separatedness	1
n	# of Particles / box	4
Threads per Block		256
Number of Blocks		528
Warp Size		32
Source-centric M2L		True
FP32		True
GPU		GH200

IV. Results & Evaluation

A. Benchmarking Platform

The benchmarking process was done on a nvidia GH200 Grace Hopper Superchip system and was tested on systems with V100 and A100 GPUs. All input data was generated randomly using a uniform particle distribution. The accuracy and precision of the results were compared against the verified CPU FMSolvr solution, ensuring that the values were always within the same order of magnitude.

The far-field FMM benchmark involved measuring the time taken to run the complete GPU routine, including data structure initialisation and particle sorting, as the far-field kernel execution time. Memory allocation, copying and de-allocation times were excluded since, in a simulation environment, these would only need to be run once, and thus their overhead would become inconsequential given sufficient iterations. Once the particle data is loaded onto the GPU, the entire timestep routine can be run without leaving the device. Finally, the evaluation figures show the average run times over ten timesteps with a default set of parameters unless mentioned otherwise, described in Tab. I.

B. Bottleneck Analysis

The primary bottleneck of the far-field algorithm, as expected, is the M2L operator, which contributes to 95.7% of the total run-time performance for the parameters shown in Tab. I. An examination of the profiling data reveals that

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on August 26,2025 at 20:40:27 UTC from IEEE Xplore. Restrictions apply.

TABLE II: Achieved compute and memory throughput compared to theoretical peak given by the Nsight Compute (NCU) profiler on the GH200 for parameters in Tab. I with a fully occupied grid of 528 blocks and 256 threads per block.

Operator	Compute (%)	Memory (%)
P2M	54.94%	78.85%
M2M	70.90%	50.22%
M2L target	70.06%	36.33%
M2L source	75.74%	29.72%
L2L	66.58%	26.74%
L2P	64.56%	17.46%

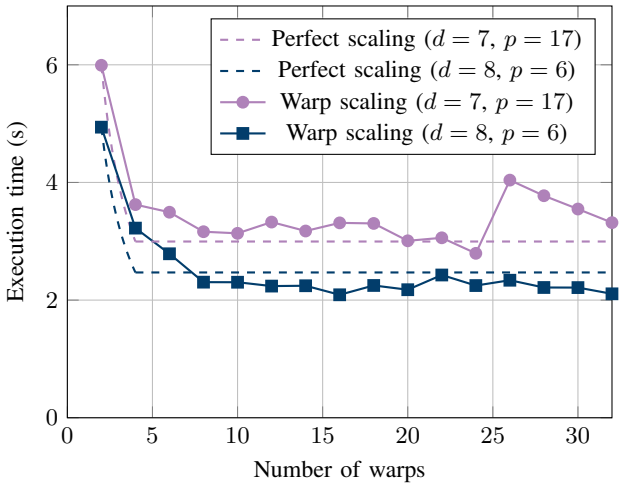


Fig. 11: Plot varying the number of threads per block launched for a grid with 2 blocks per SM. For high depth $d = 8$, $p = 6$ parameters, additional warps help alleviate latency bottlenecks compared to high multipole order $d = 7$, $p = 17$ parameters. The perfect scaling lines show how the solution would scale if it was perfectly compute-bound.

the M2L kernels are predominantly compute-bound (refer to Tab. II), despite the uncoalesced memory writes resulting from periodic-boundary conditions. This implies that the proposed memory access patterns and layout have eliminated most of the memory bottlenecks, allowing the CUDA cores to work efficiently with minimal stalling. This is the best-case scenario for an efficient GPU kernel, as managing compute-boundness in an HPC system is preferable over memory or latency-boundness. Furthermore, the solution scales independently with the number of SMs, as no inter-block dependencies have been introduced, facilitating seamless inter-SM parallelism.

The observation of compute-boundness is further reinforced by the thread scaling analysis in Fig. 11. Increasing the number of warps per block does not lead to measurable performance gains, as CUDA cores quickly become saturated. Without memory or latency constraints, additional threads offer minimal performance benefits. Conversely, for certain parameter sets (e.g., $d = 7, p = 17$), we observe a significant

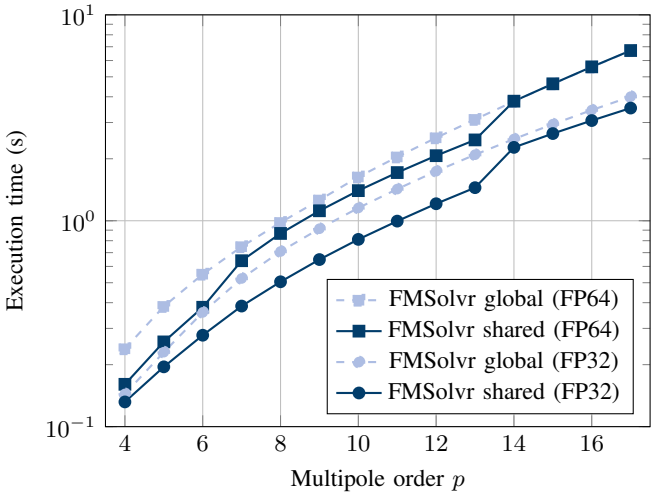


Fig. 12: Log plot showcasing the run-time improvements of utilising shared memory for the fast rotation steps. The dashed plots indicate the run-time of using global memory column vectors instead of shared memory for all E2E operators. As p increases, the L1 cache allocated for shared memory resources also increases until it overflows, resulting in execution times matching the global memory version.

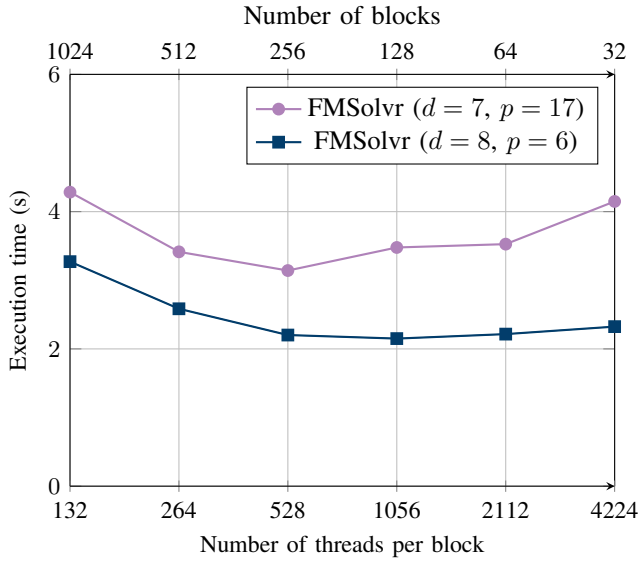


Fig. 13: Plot showing the optimal grid configuration for a set number of threads per block. For high multipole order $d = 7$, $p = 17$, the optimal is 528 blocks with 256 threads per block, while for high depth $d = 8$, $p = 6$, the optimal becomes 1056 blocks with 128 threads per block.

increase in execution time when the warp count reaches 13. At full occupancy, the solution is slower compared to using fewer warps, indicating that another bottleneck exists.

The reason for this ”bump” in execution time is predominantly due to the kernel exceeding the total L1 cache capacity of the SM, specifically for storing the cache-tiled column

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on August 26,2025 at 20:40:27 UTC from IEEE Xplore. Restrictions apply.

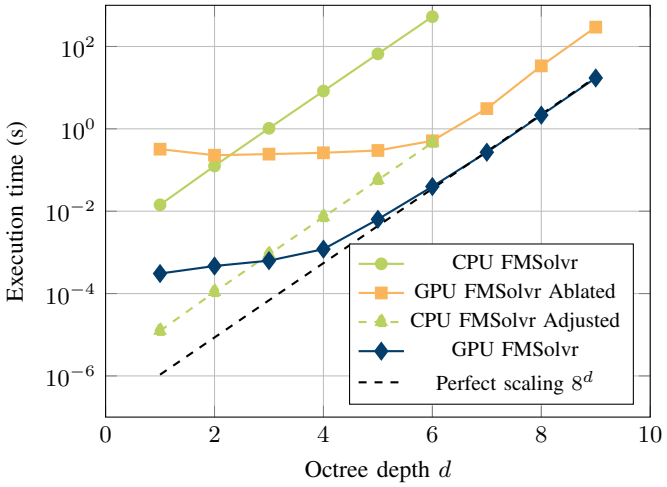


Fig. 14: Log plot describing the scaling properties of FMSolvr far-field solutions by varying depth d, while other parameters are taken from Tab. I. The test at $d = 9$ has over 536 million particles in the simulation.

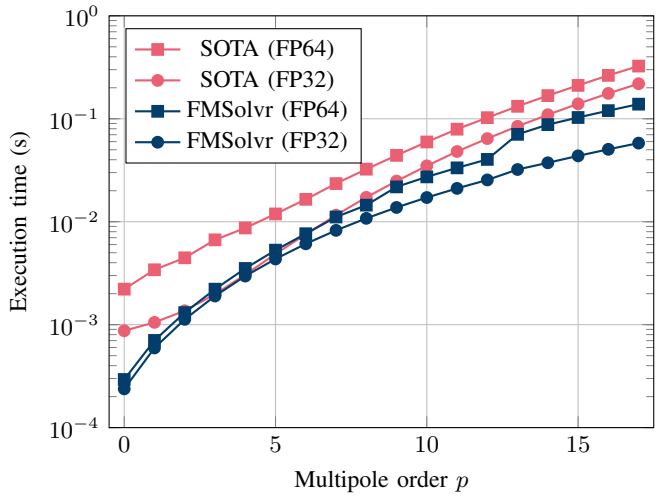


Fig. 15: Log plot comparing the proposed far-field operators versus the state-of-the-art (SOTA) [4] for single and double precision at $d = 5$ with 100 particles per box and averaged over 100 timesteps. The proposed solution on the GH200 is unilaterally faster for all values of p for both precisions.

vectors in shared memory used to process the expansions, which is showcased in Fig. 12. This is particularly visible for double precision, as we see bumps at $p = 7$ and $p = 14$, where the amount of shared memory used per block reduces the occupancy and finally exceeds the total L1 cache capacity of the SM, respectively. However, this implies that this bump is not an artifact of the solution becoming slower, but a consequence of losing access to shared memory. Without utilising shared memory at all, the scaling trend would follow the dashed plot, which is significantly slower. Therefore, this validates our

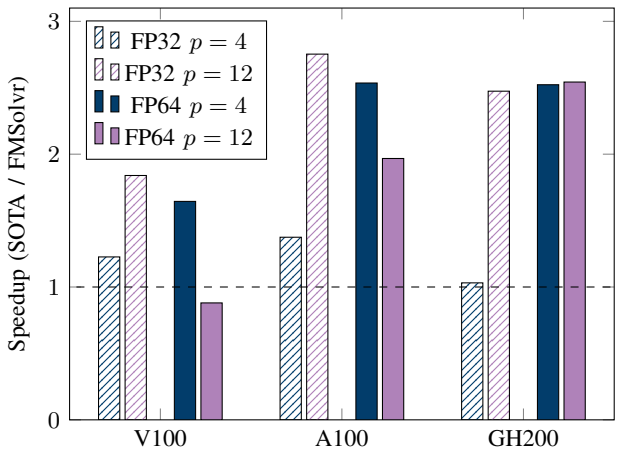


Fig. 16: Plot describing the relative speedup versus the state-of-the-art (SOTA) for $p = 4$ and $p = 12$ and single and double precisions on the V100, A100 and GH200 for $d = 5$ with 100 particles per box and averaged over 100 timesteps.

cache-tiling strategy and entails that maximising the utilisation of the available shared memory is the key for maintaining a performance advantage as p increases. One way to resolve this would be to utilise intra-expansion parallelism, where multiple threads would work on a single shared column vector concurrently. This would introduce additional synchronisation overhead and potential for load imbalance but would maintain a more stable execution-time slope overall. Otherwise, reducing the number of threads per SM will also increase the threshold of when the solution is forced to move to global memory, which can be applied with minimal performance losses, as seen in Fig. 11.

In terms of optimising grid configurations, many smaller blocks are favoured against fewer large blocks, as can be seen in Fig. 13. This is because any target-centric operators (P2M, M2M) have block-wide synchronisation overhead, and source-centric M2L has synchronisation for updating the shared read buffer in parallel, which entails favouring a larger number of independent blocks. On the other hand, warp scheduling overhead is more prevalent for higher values of p due to the increased fraction of execution time spent on processing the expansions. Therefore, we chose 528 blocks for our testing as it provided a better average-case trade-off between these stalls.

C. Comparison Against Other Solutions

Initially, we conducted baseline performance tests with the FMSolvr implementations when varying the depth parameter d, as seen in Fig. 14. The FMSolvr CPU version was run on the GH200 CPU without any vectorisation or parallelisation enabled; however, even under the optimistic assumption of ideal parallel scaling of the CPU implementation, our solution remains $11.58\times$ faster. The next test was an ablation analysis of the proposed GPU-based far-field approach to observe some of the accumulated improvements over time. Utilising source-centric M2L versus target-centric yields a 29% improvement due to fewer synchronisation statements required by

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on August 26,2025 at 20:40:27 UTC from IEEE Xplore. Restrictions apply.

the source-centric solution, coupled with significantly reduced register pressure. The L1 cache-tiling mechanism for the fast rotation algorithm provided a 104% improvement, and storing the flip matrix into constant memory yielded a 17% improvement versus global memory. Overall, the current version with operator overlapping and improvements to sorting is about $17.16\times$ faster compared to a baseline approach employing only the hybrid-centric memory access model and child-centric octree memory layout.

Subsequent comparisons involve benchmarking our solution against the current state-of-the-art (SOTA) Kohnke et al. GPU FMM far-field phase implementation [4]. It should be noted that our solution does not currently include the lattice operator, timings for which were also excluded from the competing implementation. However, this operator is inherently a $\mathcal{O}(p^4)$ operator which parallelises efficiently, and since it only operates on a single box, it does not meaningfully affect execution time. Therefore, it was excluded from this paper. Furthermore, all of our tests thus far have included memory initialisation and particle sorting, which would be part of every timestep of the FMM. However, SOTA's implementation performs sorting on the CPU and copies this data to the GPU instead, making the performance difference between these two solutions significantly greater than is presented in the plots. For the sake of a fair, academically interesting comparison between the newly designed operators, we have chosen to only compare the timings of the far-field operator kernels. Nevertheless, in Fig. 15, it can be seen that for all measured values of p for $d = 5$, our solution on the GH200 for both single and double precision is uniformly faster than SOTA. Furthermore, due to the superior $\mathcal{O}(p^3)$ prefactor time complexity of our solution versus their $\mathcal{O}(p^4)$, the difference in execution time grows as p increases.

It should be noted that for $d = 4$, our solution becomes faster than the SOTA approach starting at a crossover point of $p = 14$ for single precision (and remains consistently faster for double precision), which increases as d decreases due to the aforementioned hardware utilisation overhead. It was not possible to directly compare at $d > 5$ due to SOTA's implementation limitations. Comparing the performance on other GPU hardware, as seen in Fig. 16, similar performance trends can be seen on other GPUs as well. Only the V100, due to having significantly less L1 cache per SM, running with double precision and $p = 12$ induces a performance overhead due to the aforementioned bump, making it 12% slower than SOTA. However, adopting intra-expansion parallelisation, as employed by Kohnke et al., could mitigate these issues by consolidating shared memory usage, thereby alleviating cache pressure. Intra-expansion parallelisation techniques used within P4 shift can be adopted by fast rotation operators since the expansion data structure and dependencies remain the same.

V. Conclusion

This paper validated an efficient algorithm design for the far-field phase specialised for dense systems with $1/r^k$ potentials of the Fast Multipole Method on a GPU, which maintains a $\mathcal{O}(p^3)$ prefactor time complexity within a GPU-aware framework. It outperforms the state-of-the-art GPU FMM far-field implementation consistently across all tested values of p at depth $d = 5$ [4]. For example, for $p = 12$ for single precision FP variables, it is $2.47\times$ faster, with the performance improvement continuing to grow due to the lower prefactor complexity. This paper validated the use of fast rotation operators for a massively parallel FMM. The new algorithm uses a hybrid-centric memory access algorithm, significantly optimising the trade-off between the number of redundant memory accesses and efficient coalesced temporal memory access. This is coupled with a novel child-centric octree design that additionally provides contiguous memory access patterns for maximising cache lane utilisation without the need for explicit batching or active data structure transposition. Finally, the entire CUDA cache hierarchy is utilised, maximising data reuse for the cache-tiled fast rotation algorithm, resulting in a compute-bounded algorithm.

This work not only contributes to the research of massively parallel 1/r FMM designs on GPUs, but these concepts also benefit other parallel algorithm architectures. Through close interdisciplinary development between mathematicians and computer scientists, this paper identified the major bottlenecks of the algorithm and utilised existing literature to resolve them. Future improvements could include adopting intra-expansion parallelisation strategies to address current bottlenecks, specifically the high cache pressure and suboptimal GPU hardware utilisation for shallow trees. Furthermore, this new design could spark a massively parallel sparse far-field for the FMM, which would involve solving fundamental load-balancing and data access problems, which would also enable efficient open-boundary conditions. Furthermore, additional research to improve the accuracy of the FMM, especially for single precision execution, and affording lower values of p to reach the same results would be an important path forward. The algorithm exploits almost all of the concepts that make a parallel program fast under the CUDA programming model, implying a design that fits the hardware from the ground up. We believe that these insights lay a strong foundation for future advancements for the FMM – one which embraces strong scaling with an asymptotically efficient foundation to boot. This far-field solution, coupled with an existing P2P operator and an MPI framework, could run on large HPC machines and compete with other state-of-the-art N-body solvers, setting a new standard fit for the era of Exascale computing.

References

[1] L. Greengard and V. Rokhlin, "A fast algorithm for particle simulations," J. Comput. Phys., vol. 73, no. 2, pp. 325–348, 1987.

[2] B. A. Cipra, "The Best of the 20th Century: Editors Name Top 10 Algorithms," SIAM News, vol. 33, no. 4, p. 2, 2000. [Online]. Available: https://archive.siam.org/news/news.php?id=637.

[3] M. J. Abraham, T. Murtola, R. Schulz, et al., "GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers," SoftwareX, vol. 1, pp. 19–25, 2015.

[4] B. Kohnke, C. Kutzner, and H. Grubmüller, "A GPU-accelerated fast multipole method for GROMACS: performance and accuracy," J. Chem. Theory Comput., vol. 16, no. 11, pp. 6938–6949, 2020.

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on August 26,2025 at 20:40:27 UTC from IEEE Xplore. Restrictions apply.

[5] J. Jiménez-Vicente and E. Mediavilla, "Fast multipole method for gravitational lensing: application to high-magnification quasar microlensing," Astrophys. J., vol. 941, no. 1, p. 80, 2022.

[6] M. Schaller, J. Borrow, P. W. Draper, et al., "Swift: A modern highly-parallel gravity and smoothed particle hydrodynamics solver for astrophysical and cosmological applications," Mon. Not. R. Astron. Soc., 2024.

[7] H. Dachsel, "Corrected Article: 'An error-controlled fast multipole method' [J. Chem. Phys. 131, 244102 (2009)]," J. Chem. Phys., vol. 132, no. 11, 2010.

[8] K. N. Kudin and G. E. Scuseria, "Revisiting infinite lattice sums with the periodic fast multipole method," J. Chem. Phys., vol. 121, no. 7, pp. 2886–2890, 2004.

[9] E. Agullo, B. Bramas, and O. Coulaud, *ScalFMM: A parallel, C++-based FMM-library*, 2014. [Online]. Available: https://solverstack.gitlabpages.inria.fr/ScalFMM/.

[10] J. Board and L. Schulten, "The fast multipole algorithm," Comput. Sci. Eng., vol. 2, no. 1, pp. 76–79, 2000.

[11] W. D. Elliott and J. A. Board, Jr., "Fast Fourier transform accelerated fast multipole algorithm," SIAM J. Sci. Comput., vol. 17, no. 2, pp. 398–415, 1996.

[12] L. Morgenstern, D. Haensel, A. Beckmann, and I. Kabadshow, "NUMA-Awareness as a Plug-In for an Eventify-Based Fast Multipole Method," in *Proc. Int. Conf. Comput. Sci.*, 2020, pp. 428–441.

[13] A. Lengvenis, "Where FMM and GPUs Collide: A whirlwind tale of how the rotation operator finds its match," Jülich Supercomputing Centre, [unpublished] Tech. Rep., 2023. Supervised by Dr. Ivo Kabadshow.

[14] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton, "LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales," Comput. Phys. Commun., vol. 271, p. 108171, 2022.

[15] T. Darden, D. York, and L. Pedersen, "Particle mesh Ewald: An N log(N) method for Ewald sums in large systems," J. Chem. Phys., vol. 98, no. 12, pp. 10089–10092, 1993.

[16] U. Essmann, L. Perera, M. L. Berkowitz, T. Darden, H. Lee, and L. G. Pedersen, "A smooth particle mesh Ewald method," J. Chem. Phys., vol. 103, no. 19, pp. 8577–8593, 1995.

[17] M. J. Harvey and G. De Fabritiis, "An implementation of the smooth particle mesh Ewald method on GPU hardware," J. Chem. Theory Comput., vol. 5, no. 9, pp. 2371–2377, 2009.

[18] J. Xu, W. Ge, Y. Ren, and J. Li, "Implementation of particle-mesh Ewald (PME) on graphics processing units," Chin. J. Comput. Phys., vol. 27, no. 4, pp. 548, 2010.

[19] A. Arnold, F. Fahrenberger, C. Holm, et al., "Comparison of scalable fast methods for long-range interactions," Phys. Rev. E, vol. 88, no. 6, p. 063308, 2013.

[20] A. Iupinov, "Implementation of the Particle Mesh Ewald method on a GPU," Tech. Rep., 2016, p. 11.

[21] L. A. Barba and R. Yokota, *ExaFMM: An open-source library for fast multipole methods*, 2011. [Online]. Available: https://www.bu.edu/exafmm/

[22] L. Greengard, "The rapid evaluation of potential fields in particle systems", MIT Press, 1988.

[23] C. A. White and M. Head-Gordon, "Rotating around the quartic angular momentum barrier in fast multipole method calculations," J. Chem. Phys., vol. 105, no. 12, pp. 5061–5067, 1996.

[24] N. A. Gumerov and R. Duraiswami, "Fast multipole methods on graphics processors," J. Comput. Phys., vol. 227, no. 18, pp. 8290–8313, 2008.

[25] A. G. Garcia, A. Beckmann, and I. Kabadshow, "Accelerating an FMM-based Coulomb solver with GPUs," in *Software for Exascale Computing-SPPEXA 2013-2015*, Springer, 2016, pp. 485–504.

[26] B. Kohnke, T. R. Ullmann, A. Beckmann, I. Kabadshow, D. Haensel, L. Morgenstern, et al., "Gromex: a scalable and versatile fast multipole method for biomolecular simulation," in Software for Exascale Computing-SPPEXA 2016-2019, Springer International Publishing, 2020, pp. 517–543.

[27] E. A. Toivanen, S. A. Losilla, and D. Sundholm, "The grid-based fast multipole method–a massively parallel numerical scheme for calculating two-electron interaction energies," Phys. Chem. Chem. Phys., vol. 17, no. 47, pp. 31480–31490, 2015.

[28] H. J. C. Berendsen, D. van der Spoel, and R. van Drunen, "GROMACS: A message-passing parallel molecular dynamics implementation," Comput. Phys. Commun., vol. 91, no. 1-3, pp. 43–56, 1995.

[29] NVIDIA Corporation, *CUDA C++ Programming Guide*, 2022. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed: 2024-03-30.

[30] N. Kovachki, Z. Li, B. Liu, K. Azizzadenesheli, K. Bhattacharya, A. Stuart, and A. Anandkumar, "Neural Operator: Learning Maps Between Function Spaces With Applications to PDEs," J. Mach. Learn. Res., vol. 24, no. 89, pp. 1–97, 2023. [Online]. Available: http://jmlr.org/papers/v24/21-1524.html

[31] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, A. Stuart, K. Bhattacharya, and A. Anandkumar, "Multipole graph neural operator for parametric partial differential equations," Adv. Neural Inf. Process. Syst., vol. 33, pp. 6755–6766, 2020.

[32] I. Kabadshow and H. Dachsel, "The error-controlled fast multipole method for open and periodic-boundary conditions," in *Fast Methods for Long-Range Interactions in Complex Systems, IAS Series*, vol. 6, pp. 85–114, 2011.

[33] L. Nylons, M. Harris, and J. Prins, "Fast n-body simulation with CUDA," in *GPU Gems*, vol. 3, pp. 62–66, 2007.

[34] H. B. Li, "A CUDA Implementation of the All-Pairs N-Body Algorithm," Appl. Mech. Mater., vol. 711, pp. 308, 2014.

[35] TOP500, *TOP500 List - November 2024*, 2024. [Online]. Available: https://top500.org/lists/top500/2024/11/. Accessed: 2025-02-25.

[36] UKRI Excalibur Project, *Excalibur Projects*, 2024. [Online]. Available: https://excalibur.ac.uk/projects. Accessed: 2024-04-09.

[37] W. Dehnen, "A fast multipole method for stellar dynamics," 2014. [Online]. Available: arXiv:1405.2255 [astro-ph.IM].

[38] D. Malhotra and G. Biros, "PVFMM: A parallel kernel independent FMM for particle and volume potentials," Commun. Comput. Phys., vol. 18, no. 3, pp. 808–830, 2015.

[39] B. Bramas, "TBFMM: A C++ generic and parallel fast multipole method library," J. Open Source Softw., vol. 5, no. 56, p. 2444, 2020.

[40] Admin Magazine, "Porting CUDA to HIP," 2023. [Online]. Available: https://www.admin-magazine.com/HPC/Articles/Porting-CUDA-to-HIP. Accessed: 2023-04-10.

[41] N. Kondratyuk, V. Nikolskiy, D. Pavlov, and V. Stegailov, "GPU-accelerated molecular dynamics: State-of-art software performance and porting from Nvidia CUDA to AMD HIP," Int. J. High Perform. Comput. Appl., vol. 35, no. 4, pp. 312–324, 2021.

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on August 26,2025 at 20:40:27 UTC from IEEE Xplore. Restrictions apply.