Journal of Algorithms and Computation

journal homepage: http://jac.ut.ac.ir

Acceleration of Near Field Computation of MLFMA on a single GPU by Generating Redundancy in Data

M. Sadeghi[*1] and A. Torabi[†1]

[1] School of Engineering Science, College of Engineering, University of Tehran

ABSTRACT

The efficiency of the Multilevel Fast Multipole Algorithm (MLFMA) on distributed and parallel systems, especially on GPUs, has been the focus of extensive researches. While there has been considerable emphasis on improving far-field computations within MLFMA, the acceleration of near-field computations on GPUs has not been as thoroughly investigated. While some existing approaches improve GPU memory performance using common, intuitive ideas without analytical modelling, this paper aims to leverage analytical performance models to make more informed decisions regarding the P2P operator through data replication. Our model indicates that applying data redundancy in Global Memory as a form of restructuring can enhance the algorithm's performance by nearly 13 times for lower-density problems, compared to a baseline implementation that relies on the SoA scheme.

Keywords: Multilevel Fast Multi-Pole Algorithm, Graphics Processors, Performance Evaluation

ARTICLE INFO

Article history:
Research paper
Received 16, December 2024
Accepted 28, December 2024
Available online 30, December 2024

AMS subject classification: 68T09.

[*] Corresponding author: A. Torabi, Email:ab.reza.torabi@ut.ac.ir
[†] sadeghi.morteza@ut.ac.ir

1 Introduction

The Fast Multipole Method (FMM) [1] and the Multilevel Fast Multipole Algorithm (MLFMA) [2] are designed to accelerate matrix-vector multiplication (MVM) in various scientific simulations, such as telecommunications, physics, mechanics, and chemistry. MLFMA can reduce the complexity of MVM to O(N) in certain scenarios [3]. While MLFMA is effective on a single CPU, additional acceleration is required for large-scale problems on supercomputers. Although far-field computation has been optimized for more than a decade, near-field computation has not received the same level of attention. Many existing optimizations tend to overlook GPU cache bottlenecks and rely on basic GPU optimization techniques. This paper proposes the use of data redundancy and performance modeling to tackle the degradation of GPU cache performance in near-field computation, aiming to direct future research towards accurate modeling and optimization of MLFMA operators on a single GPU.

Far-field computation in MLFMA has been thoroughly investigated for large-scale applications on GPUs and GPU clusters. Initial studies [4][5][6][7] concentrated on enhancing far-field calculations by effectively distributing computations across cluster nodes and minimizing inter-node interactions at higher levels of the tree. The hierarchical approach in [4] improved performance by modifying the algorithm to eliminate network communication at intermediate tree levels and to better balance the load at higher levels. Similarly, [5] utilized both CPU and GPU resources simultaneously to address large-scale problems with strong scalability. Additionally, [7] implemented partitioning strategies to reduce communication within the tree, while [14] took advantage of the symmetry of the M2L operator to decrease computations and improve performance.

In contrast, the acceleration of P2P operators has not been as widely studied, as they can be divided into smaller, independent subtasks that can be processed concurrently with far-field computations on the CPU. As highlighted in [8], the P2P operator is the second most time-consuming component in MLFMA, accounting for about 30% of the total execution time. This paper aims to enhance the performance of the P2P operator. Several studies [9][10][11] have sought to reduce the complexity of P2P operations on a single GPU. These approaches are particularly significant because they can be scaled to multiple GPUs, given that P2P problems can be decomposed into smaller, independent subproblems.

In [9], researchers utilized shared memory and on-the-fly techniques to minimize memory access, which increased the memory volume and computations on the device while improving overall speed. For the P2P operator, they transferred data to thread registers, which have low access times and limited capacity, restricting each thread to managing interactions between 320 point-pairs. Study [10] enhanced coalesced memory access by categorizing P2P interactions into two groups: those within a box and those between neighboring boxes. The latter group led to uncoalesced memory accesses, so separating them allowed the first kernel to execute more quickly, thereby reducing overall execution time. In [11], researchers achieved a speedup of over 400 by integrating on-the-fly techniques with interaction separation, which increased device computations and

decreased memory usage. Additionally, [12] distributed the solution of a single-level FMM for acoustic problems between the CPU and GPU in a multi-GPU setup, discovering that increasing the workload for GPU threads and the number of groups each thread processes creates a trade-off between CPU and GPU execution times. They modeled execution time and applied it in [13] to identify the optimal group size.

One overlooked aspect of these studies is that their implementation decisions were guided by intuition and known GPU optimization techniques and validating their effectiveness through experimental results, rather than modeling. Study [9] performed experiments with various problem sizes and tree levels but did not investigate how point density influenced performance. In [11], results from nine experiments with different tree levels and densities indicated that speedup has a positive correlation with point density at a fixed tree level; however, they did not provide further details on these findings. In other studies [12][13] researchers employed execution time modeling, which necessitates hardware-dependent constants that cannot be determined until after system profiling.

Data replacement or redundancy is a well-established technique in GPU literature that has been studied. It has been concluded that memory-divergent applications on GPUs experience high spatial locality, yet GPUs are unable to fully utilize this characteristic [15]. Unlike single core caches, GPU caches are shared among multiple threads, which can result in cache contention, false sharing, and elevated miss-rates, particularly in applications with irregular memory access patterns [16]. Consequently, spatial locality is a crucial factor in modeling the performance of GPU applications [17]. Near-field computation exhibits a stencil parallel pattern, classifying it as a memory-divergent application that also faces similar false-sharing and cache-contention challenges. This issue has been largely overlooked in previous MLFMA research. One study [18] introduced a novel method to reduce miss-rates and enhance bandwidth for applications with significant read-only data sharing by adaptively reconfiguring the last-level cache between shared and private models. However, this approach was aimed at private cache-friendly applications, which do not apply to the P2P operator in MLFMA.

This paper investigates the use of data replication to alleviate cache contention for the P2P operator and demonstrates that performance modeling can effectively predict algorithm speedup based on the employed data structure.

1.1 Our Contribution

As discussed, the main challenge with P2P efficiency arises from divergent memory accesses on the device. Previous studies have attempted to minimize memory usage by taking advantage of GPU memory characteristics. This paper proposes a restructuring of data -specifically replicating data for each thread- to reduce cache contention. Although data redundancy introduces some overhead, it enhances the speed of GPU kernels by reducing false sharing and cache contention. The overhead primarily occurs during the data collection phase on the CPU, as replicating data

increases the amount of data transferred between the GPU and RAM. We aim to find the optimal point in this tradeoff using a mathematical model. To assess the impact of data restructuring on overall speedup, analytical models are essential. Although [13] presented a performance model, it is not applicable in this context, as it concentrated on thread grouping rather than data restructuring.

This paper introduces a performance model that can be applied to any data restructuring technique for the P2P operator. The model takes into account the complexity of data collection on the CPU, GPU kernel execution, and MLFMA parameters. Data access locality is proposed as a key factor in modeling the speedup of GPU implementations, capturing how data restructuring affects data access speed. Although the model does not yield exact predictions of speedup, it can approximate the optimal parameters for the algorithm that result in faster execution on a single device. Additionally, it can estimate the segment size when dividing the entire problem into segments for parallel execution on a cluster. Our model can find optimal value for box size (i.e., the number of points per box) to achieve greater speedup, similar to the approach in [13], which optimized group size for single-level FMM algorithms in heterogeneous environments.

In this paper, A straightforward data restructuring technique is proposed: introducing redundancy to ensure thread independence, meaning that each data element is accessed by only one thread, and each thread loads all necessary data with minimal instructions. While this approach increases the data volume and prolongs the data collection phase, the overhead is accounted for in the performance model, which helps identify optimal parameters that make redundancy advantageous.

The case study problem used to evaluate this technique is the electrical potential function due to its ease of implementation. The potential function is applied to a 2D Perfect Electric Conductor (PEC) where the source and target (radiating and receiving) points are randomly distributed across the surface. The MLFMA formulation for this problem is derived from [14]. Since MLFMA is kernel-independent, changing the kernel is not expected to significantly impact the results.

Our novelty and contributions can be summarized as follows:

- Data-restructuring is proposed as an optimization technique for the P2P operator of MLFMA, a concept that has not been previously introduced.
- An analytical performance prediction models is developed to estimate the impact of data restructuring on the speedup of the P2P operator, replacing earlier practical approaches.
- Data locality is introduced as a significant factor in modeling the speedup of algorithms on GPUs and a method is proposed to quantify it.

Although the innovation presented in this article has not, to our knowledge, been addressed in prior works, it does have limitations and drawbacks, including:

- It does not model the speedup of data transfer, making it reliant on empirical data.
- The simplicity of the analytical modeling makes the proposed technique infrastructure-dependent; the performance model relies on coefficients that describe unknown hardware parameters, which also makes it dependent on empirical data.

- The proposed method is not automatic and requires researchers to invest effort in accurately modeling performance.
- The application of the proposed methodology to larger problems using pipelining and across multiple GPUs is not addressed in this article.
- While the potential function used in this paper is straightforward to analyze and model, other kernels may encounter thread divergence or utilize shared memory which are not considered in this study.

Considering these pros and cons, this research can serve as a foundation for future work on MLFMA that aims to control the speed of algorithms based on device properties, data structure, and MLFMA parameters.

In the next section of this article, we present a simple implementation along with its analytical modeling, followed by a similar approach for the improved version. Section 3 examines the accuracy of the models using empirical data. Based on these models, we then test the efficiency of the proposed technique on problems with varying densities while running on a single GeForce 1050. Finally, sections 4 and 5 provide brief conclusions and suggestions for future work.

2 Methodology

In this paper, a performance model is developed to evaluate the speedup of the GPU kernel for the P2P operator based on data restructuring. The speedup is approximated by comparing the complexities of two algorithms: the baseline algorithm and the redundant one. This complexity encompasses the data structure, data collection algorithm, GPU kernel algorithm, and MLFMA parameters.

Initially, a simple implementation of the P2P operator is chosen as the baseline implementation which uses SoA data structure, followed by another implementation with data redundancy. The first implementation is referred to as the indexing method, while the second is called the repetition method. These two typical approaches are selected to validate the effectiveness of the performance modeling. To simplify the analytical models, no conventional GPU optimization techniques -such as overlapping, pre-fetching, exploiting shared memory, or dynamic parallelism- were utilized.

The work of [14], which addresses solution of 2D Coulombic problems, serves as the foundation for the presented implementation. According to this study, a MLFMA tree is defined by three main parameters: N (the number of samples), CT (the clustering threshold), and L (the tree height). The clustering threshold ensures that the number of samples (points) per box remains below a specified limit. If a box contains more samples than CT during the tree-building process, the tree level increases by one and all boxes split into four smaller boxes which contain $\frac{CT}{4}$ samples. The maximum number of samples in all boxes after fixing tree level is denoted as t in this paper. The value of t is calculated after generating the 2D mesh and is used in determining the complexity of the algorithms.

Each method is described in three phases: data collection, data transfer, and GPU kernel execution. The behavior of the two methods presented is contrasting; the indexing method is expected to perform faster during data collection and transfer but slower during GPU kernel execution, whereas the repetition method is anticipated to demonstrate the opposite behavior.

2.1. Indexing Method

In this method, data is not replicated for each thread, and all threads access non-adjacent memory banks. The data structure for this method is compressed, which results in faster data collection and data transfer phases.

2.1.1 Data Collection

In the Indexing method, data is organized into seven arrays. Two of these arrays are used to store the coordinates of all source and target points, while one array holds the potential values of the source points. The remaining arrays maintain indexes that reference these arrays. The fourth array contains the indices of target points within each box, with the indices of source points being appended to this array as the boxes are traversed in Morton order. The fifth array records the starting index of each box in the previous array, referred to as the second-order index. The sixth array includes the indices of the source points that are neighbors of each box, arranged in consecutive Morton order. Finally, the seventh array contains the second-order indices from the previous array.

Algorithm (1) in the appendix illustrates the execution order for the data collection phase of the Indexing method. The execution time for data collection on the CPU can be expressed by equation (1), where N represents the problem size, $m_{indexing}$ denotes the time for a single read or write operation in RAM, and B is the number of boxes. The number 9 in equation (1) corresponds to the count of adjacent neighboring boxes in the 2D problems. The total number of boxes is 4^{L-1}, where 4 is the branching factor of the tree and L is number of levels of the tree.

$$T_{iterate\ and\ collect\ \_\ indexing} = N\left(7m_{Indexing}\right) + 4^{L-1}(m_{Indexing}(11 + 19t) + 3) \quad (1)$$

2.1.2 Data Transfer

The total memory allocated for this technique, measured in bytes, is defined in equation (2). The point coordinates and potential values are stored as doubles, while the index values are stored as integers. This data is transferred to GPU memory after the data collection phase and prior to the execution of the GPU kernel.

$$Memory_{Indexing} = 5NDouble + 4^{L-1}(2 + t + 9t)Integer = 40N + 4^{L}(2 + 10t) \quad (2)$$

2.1.3 GPU Kernel

In the indexing method, each GPU thread is tasked with calculating the near field for all target points within a box. Each thread begins by extracting the second-order index of the target points associated with its corresponding box, followed by an iterative loop to retrieve the indices of the target points. During each iteration, the thread extracts the second-order index of the source points relevant to its box and iterates through them in an inner loop. It then retrieves their coordinates and potential values, ultimately applying the electric potential function to a pair of target and source points. Algorithm (2) in the appendix outlines the execution order in the GPU kernel for the Repetition method.

However, since each thread accesses data located in non-consecutive memory banks (seven arrays stored in different memory locations), memory requests result in significant cache misses, leading to poor performance.

The execution time for each GPU thread is expressed in equation (3), where $O_{1-computati}$ represents the time taken by the potential function applied between two pairs of points. In equation (3), the term t is used instead of CT, as the maximum number of points in each box is less than or equal to t. The term t is a random variable and may vary across different runs of MLFMA, but it is guaranteed to be less than or equal to CT.

$$\begin{aligned} T_{Kernel\_Indexing} &= 4m_{Indexing} \\ &+ t\left(3m_{Indexing} + 9t\left(5m_{Indexing} + O_{1-computation}\right) + m_{Indexing}\right) \\ &= 4m_{ind} \times (11.25t^2 + t + 1) + 9t^2 O_{1-computation} \end{aligned} \tag{3}$$

2.2 Repetition Method

In this method, each thread computes the interaction between a target point and its neighboring source points. The source points surrounding the targets within a box are replicated t times, with each replication corresponding to a target point. While the time required to collect and transfer data to the GPU is expected to be longer than in the Indexing method, the GPU kernel is anticipated to execute faster due to serial memory requests, thereby offsetting the additional time spent in the collection and transfer phases.

Each target point can have a maximum of 9 neighboring boxes, and each box can contain up to CT source points. Consequently, the maximum number of array elements allocated to each target point is equal to $(2 + 1 + 9 \times 3 \times t)$ Double elements. The first two variables represent the coordinates of the target point, the next number accounts for the number of nearby sources, and the subsequent 9×3 elements correspond to the 27 neighbors, which include two coordinates and one potential value. All this data is stored in Double format, but the second digit is parsed as an integer.

The execution order for the data collection phase of the Repetition method is outlined in Algorithm (3) in the appednix. During the data collection phase, for each box and each target point within it, all neighboring source points are extracted and appended to the data array.

The execution time for the collection phase is expressed in equation (4). Based on assumptions similar to those made for the Indexing method, this relationship can be summarized in equation (5).

$$T_{IterateAndCollect_{Repetition}} = B(3r + t(1r + 2w + find\_nei() + 1w + 9\big(1r + t(1r + 2w + 1r + 1w)\big) + 1w \quad (4)$$

$$T_{IterateAndCollect_{Repetition}} = 4^{L-1}(m_{Repetition} + 11tm_{Repetition} + 45m_{Repetition}t^2 + t * find\_nei()) \quad (5)$$

2.2.2 GPU Kernel

Each GPU thread is tasked with computing interactions between a target point and other source points in its immediate neighborhood. Each thread requires the coordinates of the target and source points, the number of source points in the nearby area, and the potential values of the neighboring source points. These elements are collected and stored consecutively for each thread, and the data from all threads are combined to form a single long array. The execution order for this kernel is illustrated in Algorithm (4) in the appendix.

In this Array of Structures (AoS) approach to storing data items, each thread requests data from one or several adjacent memory banks. In contrast, the Indexing method employs a Structure of Arrays (SoA) format, where seven arrays are stacked, and each thread accesses seven non-adjacent memory locations. Therefore, it is anticipated that the Repetition method will experience fewer cache misses compared to the Indexing method.

The execution time for each GPU thread can be expressed in equation (6). Here, $m_{Repetition}$ represents the time for a single memory access in the Repetition method. The term t is used instead of CT, as each thread primarily accesses data from up to t neighboring source points.

$$T_{Kernel\_Repetition} = 3m_{Repetition} + 9t\big(4m_{Repetition} + O_{1-computation}\big) \quad (6)$$

3 Modelling the Speedup

The speed of execution resulting from the application of the repetition method is derived from equation (7). Due to the presence of factors related to memory access time in this relationship, along with the highly variable nature of execution time, it is challenging to calculate these expressions with precision. However, equation (7) can be approximated as the sum of individual

speedups, as shown in equation (8), where the values of α, β, and γ are coefficients that can be determined based on the specific hardware and operating system.

By considering that these parameters converge to a constant value as the problem size increases, the estimation of these coefficients can be performed using the least squares method.

$$X_{Repetition} = \frac{T_{Indexing}}{T_{Repetition}} = \frac{T_{Collection\_Indexing} + T_{Transfer\_Indexing} + T_{Kernel\_Indexing}}{T_{Collection_{Repetition}} + T_{Transfer\_Repetition} + T_{Kernel\_Repetition}} \quad (7)$$

$$X_{Repetition} = \alpha X_{Collecction\_Repetition} + \gamma X_{Transfer\_Repetition} + \beta X_{Kernel\_Reprtition} \quad (8)$$

3.1 Modelling the Speedup of Data Collection

The size of the data generated and sent to the GPU, measured in bytes, is represented in equation (9). In this context, the term CT is used instead of t because a CT array element is reserved for each thread, allowing each thread to know its starting index.

$$Memory_{Repetition} = N(3 + 27\ CT)Double = 8N(3 + 27CT) \quad (9)$$

Using equations (1) and (5), the data collection speedup in the Repetition method can be approximated by considering larger terms, as expressed in equations (10) and (11).

$$\frac{T_{IterateAndCollect_{Indexing}}}{T_{IterateAndColle\ \ _{Repetition}}} = \frac{N\left(7m_{Indexing}\right) + 4^{L-1}(m_{Indexing}(11 + 19t) +\ 3 + find\_nei())}{4^{L-1}(3m_{Repetition} + 11tm_{Repetition} + 45m_{Repetitio}\ \ t^2 + t * find\_nei())} \quad (10)$$

$$\frac{T_{IterateAndCollect\_Indexing}}{T_{IterateAndCollect_{Repetition}}} \approx \frac{m_{Indexing}}{m_{Repetition}} \times \frac{1}{t} =\ X_m * \frac{1}{t} \quad (11)$$

In this context, the speed of accessing RAM using the Repetition method compared to the Indexing method is denoted as X_m. This value is estimated as the inverse ratio of the volume of the data in memory, as shown in equation (12).

$$X_m \approx \lambda_{RAM} \frac{Memory_{Indexing} + Memory_{result}}{Memory_{Repetition} + Memory_{result}} \approx \lambda_{RAM} \frac{40N + 4^L(2+10t) + 8N}{8N(3+27CT) + 8N} \quad (12)$$

In equation (12), the term λ_{RAM} is a coefficient that depends on the RAM and indicates the effect of data volume on memory access time. Equation (12) can be simplified into equation (13). Here,

D represents the average number of points in a box, which is calculated by dividing the total number of points N by the number of boxes L.

$$D = \frac{N}{4^{L-1}} \rightarrow X_m \approx \frac{1}{2} \times \frac{1}{D} \times \frac{10 \times \frac{1}{4} CT}{27CT} = \frac{1}{21.6D} \tag{13}$$

By combining equations (13) and (11), the acceleration of data collection is derived using equation (14). According to this relationship, the average density of points within the boxes negatively impacts the efficiency of the Repetition method during the data collection phase.

$$X_{IterateAndCollect\_Repetition} \approx \lambda_{RAM} \frac{1}{21.6tD} \tag{14}$$

3.2 Modelling the Speedup of GPU Kernel

In the Repetition method, a greater number of threads are utilized compared to the Indexing method, approximately t times. The overhead incurred by using more threads than the number of device cores is considered in the speedup formulation.

The speedup of the Repetition method during GPU kernel execution is expressed in equation (15). By substituting equations (3) and (6) into equation (15) and applying simplifications, the kernel speedup is approximated in equation (16).

$$\begin{aligned} X_{kernel\_Repetition} &= \frac{T_{Kernel\_Indexing}}{T_{Kernel\_Repetition}} \times \frac{\frac{Num\ Threads\ Indexing}{Total\ Device\ Cores}}{\frac{Num\ Threads\ Repetition}{Total\ Device\ Cores}} \\ &= \frac{T_{Kernel\_Indexing}}{T_{Kernel\_Repetition}} \times \frac{4^{L-1}}{N} \end{aligned} \tag{15}$$

$$\begin{aligned} X_{kernel_{Repetition}} &= \frac{T_{Kernel_{Indexing}}}{T_{Ker\quad Repetition}} = \frac{4m_{Indexing}(11.25t^2 + t + 1) + 9t^2 O_{1-computation}}{m_{Repetition}(9t + 3) + 9tO_{1-computation}} \\ &\leq \frac{4m_{Indexing}}{3m_{Repetition}} t = \frac{4}{3} X_{mem\_repetition} \times t \end{aligned} \tag{16}$$

The ratio of memory access time is considered constant for both the Repetition and Indexing methods. While accurately measuring memory access time can be challenging, its speedup can be approximated. In this paper, it is assumed that the primary factor influencing memory access speedup is the memory miss ratio. Although the memory miss ratio can be obtained from device counters, this approach is not advantageous for our implementation, as the presented method does not involve runtime optimization. Instead, the miss-rate is estimated based on memory access locality. This implies that the closer the data is located in memory, the higher the cache hit rate, resulting in faster access times to memory.

$$X_{mem\_Repetition} \approx \lambda_{GPU} \frac{MissRatio_{Repetition}}{MissRatio_{Indexing}} \approx \lambda_{GPU} \frac{Locality_{Repetition}}{Locality_{Indexing}} \quad (17)$$

In equation (17), λ_{GPU} is a coefficient that depends on the GPU hardware and represents the performance of the cache. The term $MissRatio_{Repetition}$ is an estimation of the cache miss-rate, specifically the number of accesses to non-neighboring memory banks by each thread. The $Locality$ is correlated with miss ration. The $Locality$ or cache miss-rate for a thread is defined as the ratio of the number of memory banks accessed by that thread to the total number of memory banks accessed by all threads, as expressed in equation (18).

$$Locality_{Indexing} = \frac{\#\ of\ non-adjacent\ memory\ banks\ each\ thread\ access}{total\ occupied\ memory\ banks} \quad (18)$$

In equation (18), if all the data accessed by a thread is located in fewer memory banks, the thread will experience fewer cache misses. In the Indexing method, memory bank accesses are distributed across non-adjacent memory banks due to the use of a Structure of Arrays (SoA) data scheme. Each thread in the Indexing method reads values from seven arrays, which means that each thread incurs at least 7 cache misses when N and t are sufficiently large. Valuer for $Locality$ for the Indexing method is calculated in equation (19), where b represents the number of bytes stored in a memory bank.

$$Locality_{Indexing} = \frac{\#MemoryBanks}{\left\lceil \frac{40N + 4^L(2+10t)}{b} \right\rceil} \quad (19)$$

$$\begin{aligned} \#MemoryBanks &= \left\lceil \frac{1Integer}{b} \right\rceil + \left\lceil \frac{1Integer}{b} \right\rceil + \left\lceil \frac{t * Integer}{b} \right\rceil + \left\lceil \frac{9t * Integer}{b} \right\rceil \\ &+ \left\lceil \frac{t * 2 * Double}{b} \right\rceil + \left\lceil \frac{9t * 3 * Double}{b} \right\rceil \end{aligned} \quad (20)$$

In equation (20), the terms include 2 indices that are stored in two different memory banks, t indices of target points, $9t$ indices of source points, $2t$ coordinates of target points, and $27t$ coordinates and potential values of source points. All of these elements are stored in separate memory banks. By taking the value of b as 512 bytes for the GTX 1050 device, equation (20) is simplified to equation (21).

$$Locality_{Indexing} \approx \frac{272t}{40N + 4^L(2+10t)} \quad (21)$$

This $Locality$ is applicable to all other threads, so the value in equation (21) must be multiplied by the number of threads used in the Indexing method. This multiplication accounts for the cumulative effect of cache misses across all threads, providing a more comprehensive estimate of the overall cache miss-rate for the method.

$$Locality_{Indexing} \approx \frac{272t}{40N + 4^L(2 + 10t)} \times 4^{L-1} \quad (22)$$

In the Repetition method, all GPU threads read the same amount of data from the entire data array, and all of this data is stored in contiguous memory banks. As a result, the probability of a cache miss is significantly reduced because of better data access locality. $Locality$ for Repetition method si defined in equation (23).

$$Locality_{Repetition} = \frac{1}{N} \quad (23)$$

By substituting equations (22) and (23) into the ratio expressed in equation (19), the ratio of cache misses between the two methods is given by equation (24). In this relation, t can be excluded due to its relatively small value. Considering that number of boxes are smaller than number of points or $4^{L-1} <= N$, the miss ratio is of the order $O(N^{-1})$.

$$\frac{Locality_{Repetition}}{locality_{Indexing}} = \frac{1}{N} \div \left(\frac{272t \times 4^{L-1}}{40N + 4^L(2 + 10t)}\right) = \frac{40N + 4^L}{N \times 4^{L-1}} \approx \frac{40}{4^{L-1}} + \frac{4}{N} \geq \frac{44}{N} \quad (24)$$

By substituting equation (24) into equation (17), an approximation of memory access speedup is obtained in equation (25). Furthermore, by incorporating equations (25) and (16), the kernel acceleration of the Repetition method can be calculated as shown in equation (26).

$$X_{mem\_Repetition} \approx \lambda_{GPU} \times \frac{44}{N} \quad (25)$$

$$X_{kernel_{Repetition}} = \frac{4}{3} X_{mem_{Repetition}} \times t \times \frac{4^{L-1}}{N} \quad (26)$$

$$\approx \frac{4}{3}\lambda_{GPU} \times \frac{44}{N} \times t \times \frac{4^{L-1}}{N} \approx 55.3\lambda_{GPU} \times t \times \frac{1}{ND}$$

It can be argued that the speedup of the GPU kernel in the Repetition method decreases as the size of the problem increases. Since the value of t is greater than D, increasing the point density in equation (26) positively impacts the GPU kernel speed and can mitigate the negative effects associated with t.

3.3 Modelling Data Transfer Speedup

The time required for data transfer is affected by various factors, including hardware and the operating system; however, it typically correlates with the amount of data. The data volume ratio, as calculated in (13), indicates that an increase in the density of points within the boxes leads to a reduction in the speed of the Repetition method.

3.4 Analysis of Model

The overall speedup is determined using (8) and incorporates the relationships from (14), (16), and (25), which is outlined in equation (27). The variable t has opposing effects on the acceleration of data collection and the GPU kernel. This implies that the Repetition method is not always more efficient than the Indexing method. Additionally, the term $\frac{1}{D}$ can be factored out of the entire equation. It can be concluded that increasing the number of boxes while keeping N constant (and thus reducing D) results in the Repetition method being generally faster than the Indexing method.

$$X_{Repetition} = \alpha\lambda_{RAM}\frac{1}{21.6Dt} + \gamma\frac{1}{21.6D} + \beta(55.3\lambda_{GPU} \times t \times \frac{1}{ND}) \quad (27)$$

To identify an optimal balance between the two methods, it is essential to know the values of λ_{RAM}, λ_{GPU}, and the coefficients α and β. According to the proposed analytical model, it is recommended that to maximize the acceleration of the Repetition method, the number of boxes should be increased while maintaining a constant N, and the number of sample points within them should be decreased. In other words, this method is more advantageous than the Indexing method for scenarios where the density of sample points is low.

4 Evaluation of Model

4.1. Evaluation of Model Error

This section compares the basic CPU implementation derived from [14] with the Indexing method and the Repetition method. In the CPU implementation, the boxes are processed in Morton's indexing order. For each target point within a box, neighboring source points are initially extracted, and for each source point in the E_1 neighborhood, the potential function is executed once, with its value being summed with the far-field induced potential.

4.1.1. Evaluation of Data Collection Performance Model

Due to the data collection being conducted on the CPU and the experimental nature of the setup, the techniques presented are compared using small-sized problems. The tests begin with a problem size of $N = 5{,}000$ and increase to $100{,}000$ points in increments of $5{,}000$. Subsequently, the problem size is raised to $350{,}000$ points, with increments of $50{,}000$ points. The default value for CT is set to 15, as the techniques are more effective for sparse problems. The tree height L starts at a default value of 3 and increases based on N and CT. Each test is repeated 20 times on a similar tree, and the running times are averaged over these 20 runs.

The initial speedup for each method is calculated by dividing its running time by the running time of the baseline model. The speedup of the Repetition method is then determined by dividing its speedup by that of the Indexing method. Figure 1 illustrates the speedup of the Repetition method. Additionally, the speedup is approximately calculated in (27) based on the values of t and D. The measured speedup is represented by a solid black line in Fig.1. The vertical axis on the left

indicates the speedup amount relative to the problem size N, while the orange dashed line represents the estimated speedup using (14), with the vertical axis on the right showing the corresponding speedup calculated using (26). In both plots, the performance models presented align with the trend observed in the empirical data. To achieve a more accurate fit, it is necessary to determine the values of λ for both (14) and (26). In this paper, these values are calculated by dividing the experimental running time by the performance model and then averaging the results across all experiments. The value of λ for formula (14) ranges between 75 and 180, with an average value of 108.

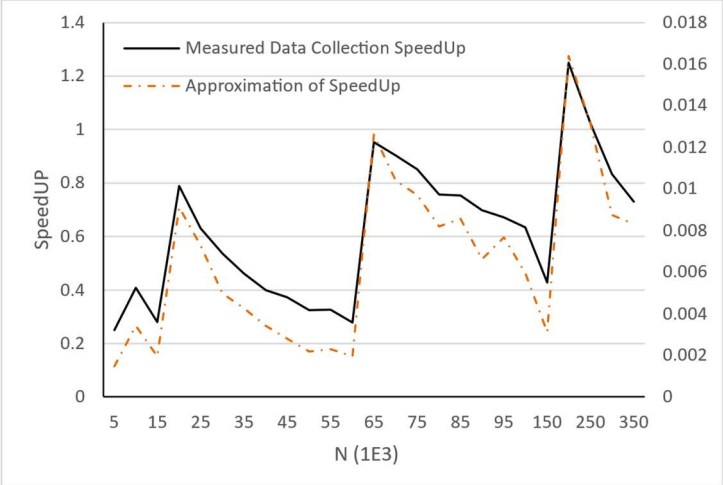

Figure 1 – The speedup of the Repetition method during the data collection phase is shown by the black line on the left vertical axis, while the estimated value using (14) is represented by the orange line on the right vertical axis. The estimated value exhibits a similar trend to the measured speedup.

The oscillation observed in Fig.1 is attributed to the density D. As indicated in (13), the data volume is inversely related to D. The volume of data influenced by the value of D significantly impacts the execution speed of data collection. Considering this relationship and (14), the theoretical maximum execution speedup is expressed in (28).

$$X_{IterateAndCollec\_Repetition} \approx \lambda \frac{1}{21.6tD} \geq 1 \tag{28}$$

$$\frac{CT}{4} \sim 4 \leq t, 1 \leq D \rightarrow X_{IterateAndCollect\_Repetition} \leq 1.25$$

4.1.2. Evaluation of GPU Kernel Performance Model

Since the execution of the GPU kernel is significantly faster than data collection on the CPU, this section focuses on experiments involving larger problems. In these tests, N is increased by 1,000 at each step, starting from 1,000 points up to 100,000 points, and then increased by 50,000 points

per step until reaching 1,000,000 points. The values of CT and L remain constant throughout all tests.

Fig.2 displays the calculated speedup of the GPU kernel for the Repetition method in comparison to the Indexing method, represented by a solid black line. The left axis uses a logarithmic scale for speedup to mitigate the impact of jumps in the value of N on the graph's resolution. The orange dashed line indicates the predicted value based on (26). While this predicted value generally follows a similar overall trend to the measured speedup, it does not align perfectly during certain fluctuations. One reason for this discrepancy is that the running time on the GPU very variable.

The maximum value of N for which the Repetition method remains faster than or equal to the Indexing method, based on (26) and assuming that $t \leq CT$, is calculated in (29).

$$X_{kernel\_Repetition} \approx 55.3\lambda \times t \times \frac{1}{ND} \geq 1 \rightarrow N \leq 55.3\lambda \times \frac{t}{D} \qquad (29)$$

$$t \leq CT = 15 \rightarrow N \leq 829.5\lambda \times \frac{1}{D}$$

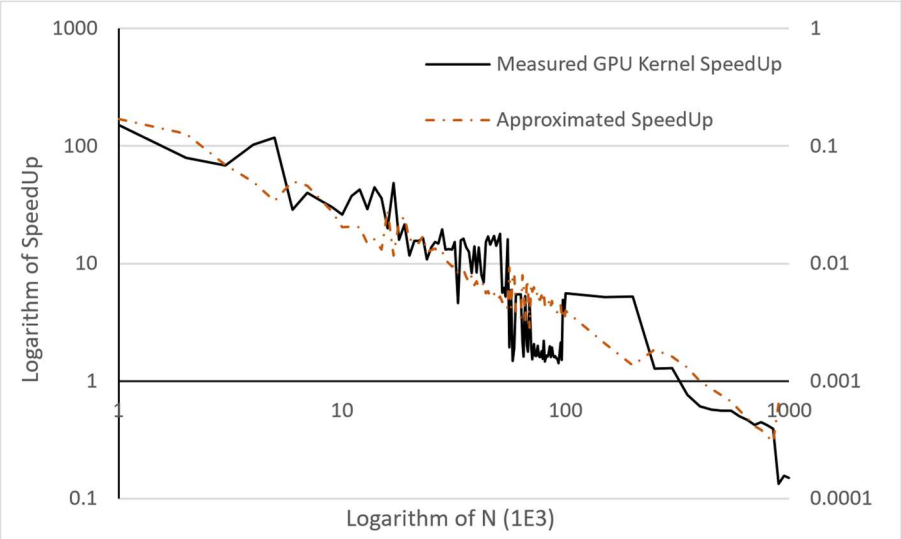

Figure 2 – Modeling GPU Kernel SpeedUp.
The measured GPU kernel speedup of the Repetition method is represented by the black line on the left vertical axis, based on the Indexing method, while the estimated value using relation (26) is shown by the orange line on the right vertical axis. Both axes are displayed on a logarithmic scale, illustrating the trend

The precise value of this relation necessitates the determination of D and λ. The value of λ is calculated by dividing the empirical data by the approximated data for each experiment and then averaging the results. Based on experimentation, the value of λ for GPU memory is approximately 640, indicating that for each unit of improvement in locality, the P2P kernel operates about 640 times faster. According to the empirical data, the value of D varied between 1 and 5. Using the

maximum value for D, the optimal value of N (assuming that t is consistently 15) can be calculated based on (26), as expressed in (30).

$$106{,}000 \leq N_{Optimal} \leq 530{,}000 \tag{30}$$

According to Fig.2, this value may not be entirely accurate; however, it is sufficiently precise for selecting the segment size to distribute the entire problem across multiple GPUs.

4.1.3. Estimation of Model Coefficients

In (27), the calculation of the coefficients α, β, and γ enables the estimation of the total acceleration time. These coefficients are derived from the data obtained during the tests conducted for both the data collection and GPU kernel execution phases, and they are expressed in (31).

$$\alpha \approx 0.82, \beta \approx 0.09, \gamma \approx 0.18 \tag{31}$$

In Fig.3, the total execution time -which encompasses the total time for data collection, data transfer, and kernel execution- is displayed alongside the predicted value based on the coefficients in (31). In this figure, the predicted values have been multiplied by the measured speedups rather than the theoretical values, as (27) is not accurate for transfer time. Instead, the formulation in (27) is useful for gaining insights into the impact of algorithm design and the choice of granularity on overall speedup. Fig.3 supports the concept of separating speedups as presented in (8); however, it is not entirely accurate since the transfer time is not modeled accurately.

The overall trends in Fig.3 closely resemble those in Fig.1, indicating that data collection time is the bottleneck of the presented method for the specified values of N and CT. This conclusion is also supported by the coefficients in (31). The speedup from the GPU kernel contributes the least to the overall speedup. Based on Fig.2 and 3, the Repetition method is marginally faster than the Indexing method when t and D are not directly manipulated but instead increase with the values of N and CT.

4.2. maximizing Total Speedup

Based on the results from the previous section and the analytical relations summarized in (27), it is recommended to reduce the values of D and t while keeping N constant to effectively utilize the Repetition method for a specific problem. One straightforward approach to achieve this is to increase the height of the tree by at least one unit after constructing it according to CT. This adjustment would increase the number of boxes by a factor of 4 (based on the tree's branching factor) and reduce D and t to a quarter of their original values.

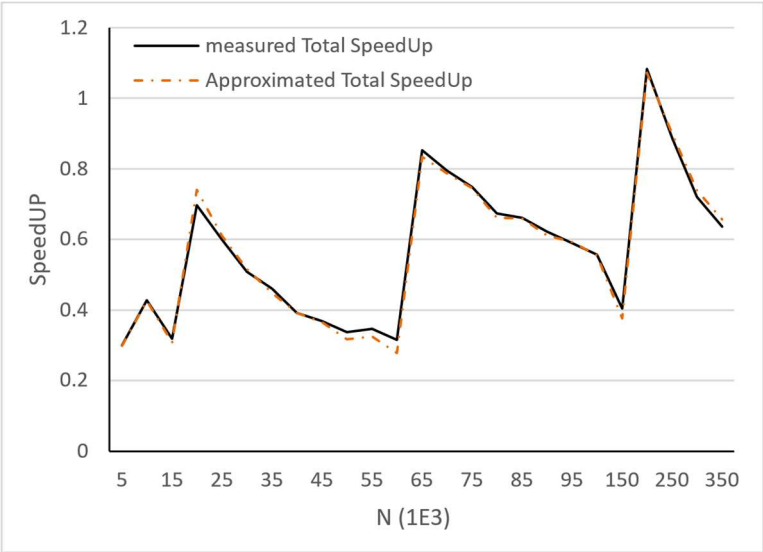

Figure 3 – Modeling Total SpeedUp.

The total speedup of the Repetition method is represented by the black line, while the estimated values based on empirical data and (31) are shown by the orange line.

By increasing the height of the tree by i units, and assuming that the coefficients remain constant, the resulting acceleration can be expressed as:

$$L' = L + i \rightarrow t' = \frac{t}{4^i}, D' = \frac{D}{4^i} \tag{32}$$

$$X'_{Repetition} = \alpha\lambda_{RAM}\frac{4^{2i}}{21.6Dt} + \gamma\frac{4^i}{21.6D} + \beta\left(55.3\lambda_{GPU} \times t \times \frac{1}{ND}\right) \tag{33}$$

$$\approx 4^{2i} \times 0.82 + 0.18 \times 4^i + 0.09$$

This indicates that the speed of data collection, and consequently the speedup in data transfer, significantly increases, while the speed of the GPU kernel remains unchanged. Equation (33) demonstrates that increasing the tree height by one unit results in the Repetition method becoming more than 13 times faster. However, these figures are merely indicative, and the actual acceleration times during runtime may not necessarily match these values.

To assess the impact of varying tree height and box density, another experiment is conducted. For the values in (34), different trees are generated with CT set to 15. After constructing the tree, L is adjusted based on the value of i. For larger values of i, the boxes become smaller, leading to a decrease in t. Conversely, for smaller values of i, both t and density increase. The value of N in these problems is defined as 4^{L-1}, where L is taken as its initial value. The baseline CPU implementation used in this experiment is the same as the baseline model referenced in the previous section.

$$i \in \{-3,-2,-1,0,1,2,3\} \quad (34)$$

$$L \in \{4,5,6,7,8,9,10,11\}$$

This indicates that the speed of data collection, and consequently the speedup in data transfer, significantly increases, while the speed of the GPU kernel remains unchanged. Equation (33) demonstrates that increasing the tree height by one unit results in the Repetition method becoming more than 13 times faster. However, these figures are merely indicative, and the actual acceleration times during runtime may not necessarily match these values.

To assess the impact of varying tree height and box density, another experiment is conducted. For the values in (34), different trees are generated with CT set to 15. After constructing the tree, L is adjusted based on the value of i. For larger values of i, the boxes become smaller, leading to a decrease in t. Conversely, for smaller values of i, both t and density increase. The value of N in these problems is defined as 4^{L-1}, where L is taken as its initial value. The baseline CPU implementation used in this experiment is the same as the baseline model referenced in the previous section.

According to Fig.6, the Repetition method outperforms the Indexing method in two distinct regions. The first region is characterized by high density, with the problem size N being smaller than 4096. In this region, as indicated by (26), the smaller value of N allows the GPU kernel to execute faster for the Repetition method, while the denser boxes further enhance the GPU kernel's performance. Additionally, it can be argued that due to the small problem size and the redundancy inherent in the Repetition method, the data from several consecutive threads fits into a single cache line. This reduces the cache miss-rate and subsequently increases the kernel speed.

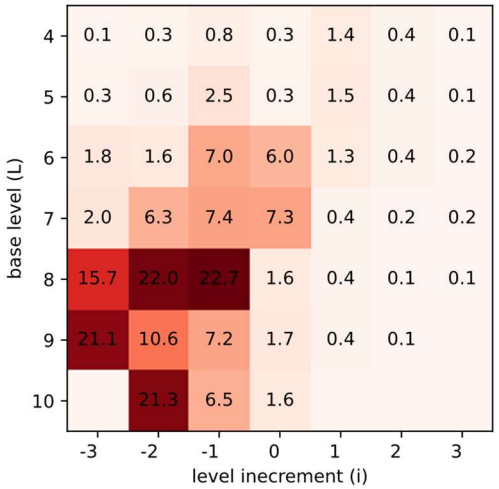

Figure 4 – Speedup of the Indexing method relative to the baseline model.

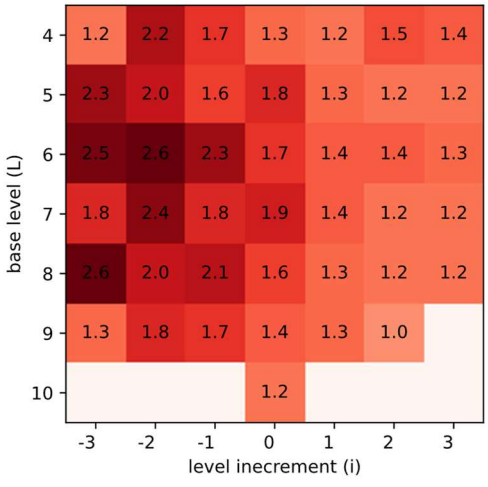

Figure 5 – Speedup of the Repetition method relative to the baseline model.

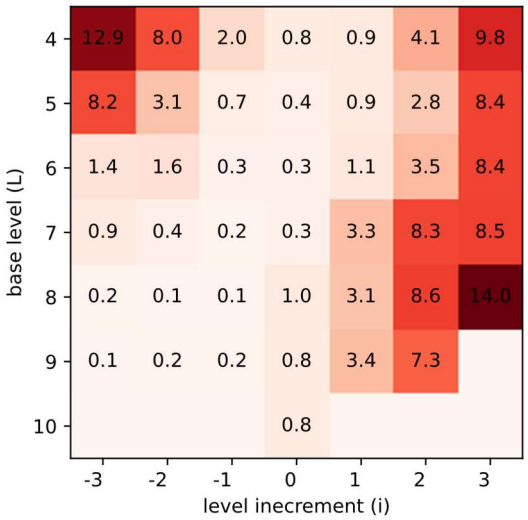

Figure 6 – Speedup of the Repetition method compared to the Indexing method.

The second region occurs when the tree density decreases while the problem size increases, specifically when $N \geq 16{,}384$ and $i \geq 2$. In this scenario, although the GPU kernel runs slower due to the value of N, the data transfer and data collection processes become faster because of their inverse relationship with D and t.

In this work, two GPU implementations of the P2P operator using different data structures were examined: one with compressed data and the other with redundancy in data. Since any alteration in data restructuring impacts the data collection time on the CPU, analytical performance models have been developed to track the effects of changes in algorithm design on the speedup of the algorithm and to identify the optimal values for the problem parameters.

5 Conclusion

The accuracy of the models was assessed by examining their overall trend lines. While they were not precise and were influenced by the system setup, they provided valuable insights for improving algorithm design. The optimization of the MLFMA tree by managing t and D was proposed through analytical models, and the results were empirically tested. According to the empirical data, the proposed modification of the algorithm achieves nearly 13 times speedup compared to the unmodified algorithm for problems involving more than 200,000 source and target points, with 2-4 points per box. This result aligns well with the findings from the analytical modeling.

6 Future Works

It is recommended to apply this modeling technique to other P2P methods or additional operators within the MLFMA framework in future research. Furthermore, it is advisable to predict the values of λ based on the specific features of the hardware. This approach would bridge the gap between algorithm design and hardware specifications, facilitating the efficient distribution of the entire problem across multiple GPUs.

While the modeling of data transfer between the GPU and RAM was not conducted with precision in this paper, developing a more accurate model in this area could enhance the overall framework of analytical modeling.

References

[1] Rokhlin, Vladimir. "Rapid solution of integral equations of scattering theory in two dimensions." Journal of Computational physics 86.2 (1990): 414-439.

[2] Song, Jiming, Cai-Cheng Lu, and Weng Cho Chew. "Multilevel fast multipole algorithm for electromagnetic scattering by large complex objects." IEEE transactions on antennas and propagation 45.10 (1997): 1488-1493.

[3] Gumerov, Nail A. and Ramani Duraiswami. “Fast Multipole Methods for the Helmholtz Equation in Three Dimensions.” (2005).

[4] Gurel, Levent, and Özgür Ergul. "Hierarchical parallelization of the multilevel fast multipole algorithm (MLFMA)." Proceedings of the IEEE 101.2 (2012): 332-341.

[5] Dang, Vinh, Nghia Tran, and Ozlem Kilic. "Scalable fast multipole method for large-scale electromagnetic scattering problems on heterogeneous CPU-GPU clusters." IEEE Antennas and Wireless Propagation Letters 15 (2016): 1807-1810.

[6] Yang, Ming-Lin, et al. "A ternary parallelization approach of MLFMA for solving electromagnetic scattering problems with over 10 billion unknowns." IEEE transactions on antennas and propagation 67.11 (2019): 6965-6978.

[7] Kohnke, Bartosz, Carsten Kutzner, and Helmut Grubmuller. "A CUDA fast multipole method with highly efficient M2L far field evaluation." Biophysical Journal 120.3 (2021): 176a.

[8] Agullo, Emmanuel, et al. "Task-based FMM for multicore architectures." SIAM Journal on Scientific Computing 36.1 (2014): C66-C93.

[9] Cwikla, M., J. Aronsson, and V. Okhmatovski. "Low-frequency MLFMA on graphics processors." IEEE Antennas and Wireless Propagation Letters 9 (2010): 8-11.

[10] Guan, Jian, Su Yan, and Jian-Ming Jin. "An OpenMP-CUDA implementation of multilevel fast multipole algorithm for electromagnetic simulation on multi-GPU computing systems." IEEE transactions on antennas and propagation 61.7 (2013): 3607-3616.

[11] Li, Shaojing, et al. "Fast electromagnetic integral-equation solvers on graphics processing units." IEEE Antennas and Propagation Magazine 54.5 (2012): 71-87.

[12] López-Portugués, Miguel, et al. "Acoustic scattering solver based on single level FMM for multi-GPU systems." Journal of Parallel and Distributed Computing 72.9 (2012): 1057-1064.

[13] López-Fernández, Jesús Alberto, Miguel López-Portugués, and José Ranilla. "Improving the FMM performance using optimal group size on heterogeneous system architectures." The Journal of Supercomputing 73 (2017): 291-301.

[14] Wang, Yang. The fast multipole method for two-dimensional coulombic problems: Analysis, implementation and visualization. University of Maryland, College Park, 2005.

[15] Lal, Sohan, and Ben Juurlink. "A quantitative study of locality in GPU caches." Embedded Computer Systems: Architectures, Modeling, and Simulation: 20th International Conference, SAMOS 2020, Samos, Greece, July 5–9, 2020, Proceedings 20. Springer International Publishing, 2020.

[16] Lal, Sohan, Bogaraju Sharatchandra Varma, and Ben Juurlink. "A quantitative study of locality in GPU caches for memory-divergent workloads." International journal of parallel programming 50.2 (2022): 189-216.

[17] Wang, Lu, et al. "MDM: The GPU memory divergence model." 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2020.

[18] Zhao, Xia, et al. "Adaptive memory-side last-level GPU caching." Proceedings of the 46th international symposium on computer architecture. 2019.

Appendix: Algorithms

Algorithm (1): Data Collection in Indexing Method

1: $Procedure\ P2P\_Data\_Collection\_Indexing$
2: $Input\ \ tree\ t$
3: $Output\ \ targetPoints, soucePoints, potentials$
$allTargets, allTargetsIndex$
$allNeighburs, allNeighbursIndex$
4: $targetPoints \leftarrow t.GetTargetPoints()$
5: $sourcePoints \leftarrow t.GetSourcePoints()$
6: $potentials\ =\ t.getPotentials()$
7: $counter1 \leftarrow 0$
8: $for\ all\ Box\ b\ \in\ t[lasteLevel].Boxes\ do$
9: $for\ all\ Point\ p\ \in\ b.targetPoints\ do:$
10: $allTargets.Append(p.index)$
11: $end\ for$
12: $allTargetsIndex[b] = counter1$
13: $counter2 \leftarrow 0$

```
14:      for all Box bj ∈ b.GetNeighburs()do
15:          for all sourec point s ∈ bj.souecPoints do
16:              allNeighbursIndex.Append(s.index)
17:              counter2 + +
18:          end for
19:      end for
20:      SoA.allNeighbursInBox.append(counter2)
21:   end for
22: end procedure
```

Algorithm (2): GPU Kernel in Indexing Method

```
1:  Procedure P2P_GPU_Kernel_Indexing
2:    Input   bunBoxes
                targetPoints, sourcePoints, potentials
              allTargets, allTargetsIndex
              allNeighburs, allNeighbursIndex
3:    Output  nearField
4:    tid ← threads index
5:    if tid < numBoxes do
6:      startTarget ← allTargetsIndex[ti]
7:      endTarget ← allTargetsIndex[ti + 1]
8:      startNeighbur ← allNeighbursIndex[ti]
9:      endNeighbur ← allNeighbursIndex[ti + 1]
10:     for i in startTarget: endTarget − 1 do:
11:       nearfield[ti] ← 0
12:       targetIndex ← allTargets[i]
13:       targetX ← targetPoints[targetIndex].GetX()
14:       targetY ← targetPoints[targetIndex].GetY()
15:       for j in startNeighbur : endNeighbur − 1 do:
16:         sourceIndex ← allNeighburs[j]
17:         sourceX ← sourcePoints[sourceIndex].GetX()
18:         sourceY ← sourcePoints[sourceIndex].GetY()
19:         u ← potentials[sourceIndex]
20:         tX ← sourceX − targetX
21:         tY ← source − targetY
22:         nearfield[ti] += Interaction(tX, tY, u)
23:       end for
24:     end for
25:   end if
```

26: $end\ procedure$

Algorithm (3): Data Collection in Repetition Method

1: $Procedure\ P2P\_Data\_Collection\_Repetition$
2: $Input\ \ tree\ t$
3: $Output\ \ dataArray$
4: $paddingSize$
$\leftarrow 2 + 1 + 9 * 3 * t.CT$
5: $index\ \leftarrow\ 0$
6: $for\ all\ Box\ b\ \in\ t[lasteLevel].Boxes\ do$
7: $for\ all\ Point\ pi\ \in\ b.targetPoints\ do$:
8: $dataAray[index] \leftarrow\ pi.GetCoordination().x$
9: $dataAray[index + 1] \leftarrow pi.GetCoordination().y$
10: $numNeighburs = 0$
11: $for\ all\ Box\ bj\ \in\ b.GetNeighburs()do$
12: $for\ all\ sourec\ point\ s \in bj.souecPoints\ do$
13: $dataArray[index +\ 3 + numNeighburs * 3] \leftarrow\ s.GetCoordinates().x$
14: $dataArray[index +\ 3 + numNeighburs * 3 + 1] \leftarrow s.GetCoordinates().y$
15: $dataArray[index + 3 +\ numNeighburs * 3 + 2] \leftarrow s.GetPotential()$
16: $numNeighburs + +$
17: $end\ for$
18: $end\ for$
19: $dataArray[index\ +\ 2] =\ numNeighburs$
20: $end\ for$
21: $end\ for$
22: $end\ procedure$

Algorithm (4): GPU Kernel for Repetition Method

1: $Procedure\ P2P\_GPU\_Kernel\_Repetition$
2: $Input\ \ \ dataArray, paddingSize$
$numTargets$
3: $Output\ \ nearField$
4: $tid \leftarrow threads\ index$
5: $if\ tid < numTargets\ do$
6: $offset \leftarrow ti * paddingSize$
7: $targetX\ \leftarrow dataArray[offset]$
8: $targetY \leftarrow dataArray[offset + 1]$

9: $numNeighburs \leftarrow dataArray[offset + 2]$
10: $nearField[ti] = 0$
11: $for\ i\ in\ 0{:}\ numNeighburs\ do$:
12: $targetX \leftarrow dataArray[offset * 3 + i * 3]$
13: $targetY \leftarrow dataArray[offset * 3 + i * 3 + 1]$
14: $u = dataArray[offset * 3 + i * 3 + 2]$
15: $tX = targetX - sourceX$
16: $tY = targetY - source$
17: $nearfield[ti] += Interaction(tX, tY, u)$
18: $end\ for$
19: $end\ if$
20: $end\ procedure$