24th International Conference on Information Technology, Computer and Telecommunication

ACCELERATION OF NEAR FIELD COMPUTATION IN MLFMA ON A SINGLE GPU BY GENERATING REDUNDANCY IN DATA

Morteza Sadeghi[1] and Abdolreza Torabi[2]

[1]Department of Engineering Science, University of Tehran, Tehran, IRAN
[2] Department of Engineering Science, University of Tehran, Tehran, IRAN

ABSTRACT

Improving efficiency of multi-level fast multi-pole algorithm (MLFMA) on distributed and parallel systems has been vastly studied, specially for GPUs. Unlike the far-field computation, acceleration of near-field computation in MLFMA algorithm on GPUs was of less concern in the literature, however there are some solutions that exploited special specifications of GPU's memory. This article proposes data replication for P2P operator and uses analytical performance models to determine its optimality criteria. By modelling the speedup, we found that making threads independence by creating redundancy in the data makes the algorithm for lower dense problems nearly 13 times faster than non-redundant mode.

KEYWORDS

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

1. INTRODUCTION

The FMM [1] and MLFMA [2] algorithms accelerate matrix-vector multiplication (MVM) in sort of scientific simulation applications like telecommunications, physics, mechanics, and chemistry. MLFMA reduces the complexity of MVM to *O(N)* in special cases [3]. Although that this algorithm is efficient on a single CPU, more acceleration is required for very largescale problems in supercomputers. Although that far-field computation has been optimized for more than a decade, computation of near-field has not been strongly pursued so far. Furthermore, presented optimizations did not address GPU cache bottlenecks and only followed simple GPU optimization techniques. In this paper, data redundancy and performance modelling were used to treat the performance degradation of GPU caches in nearfield computing. This paper tries to orient future researches toward accurate modelling and optimization of the behaviour of MLFMA operators on a single GPU.

Accelerating far-field computation of MLFMA algorithm using has been studied for more than a decade for large-scale problems on GPUs and GPU clusters. Pioneering works [4][5][6][7] focused on improving far field calculation. They worked on efficiently distributing computations over cluster nodes and decreased inter-node interactions at higher levels of tree. The hierarchical technique represented in [4] enhanced architectural performance by modifying the algorithm, which eliminated network communication in the middle levels of the tree and remedied the load balancing problem at higher levels. Beside this, using both CPU and GPU at the same time for computation in [5] let them to solve large scale problems with strong scalability. More partitioning strategies were applied in [7] that reduced communications in tree. Exploiting M2L operator symmetry was another approach to reduce computations and achieve higher performance in [14]. These four researches improved performance for far-field computation in different ways.

Acceleration of P2P operators has not received much attention because it can be decomposed into smaller independent subtasks and processed simultaneously with far-field processing when using CPU. As mentioned in [8], the P2P operator is the second longest operator in MLFMA, i.e., it

takes about 30% of the execution time. In this paper improvement of P2P operator is of interest. Several works [9][10][11] attempted to reduce complexity of P2P on a single GPU. These works are of interest because their method could be applied to more than one GPU since solving P2P could be beaked down to small independent sub problems. In [9] researchers used sharing memory and *on-the-fly* technique to reduce memory access. These techniques increase the memory volume required and increases computations in device but the overall speed improves. For the P2P operator specifically, they copied data to thread registers which has low access time and low capacity. By this choice each thread is limited to handle interactions of only 320 pairs of points. Research [10] focuses on improving coalesced access to memory by separating P2P interactions into two groups: interactions between a box and points inside the box, and interactions between a box and neighbouring boxes. The second set of interactions make uncoalesced memory accesses hence separating them from the first set lets the first kernel to executes faster Thus the overall execution time degrades. In NGIM algorithm [11] which is similar to MLFMA, researchers achieved speedup of 400 and more by combining on-the-fly technique and separation of interactions. They increased computations in device to reduce memory volume created by recalculation of indices and variables. Other researchers in [12] distributed solution of single level FMM for acoustic problem between CPU and GPU in multi-GPU environment where CPU computes translation and GPU computes aggregation/disaggregation and near-field computation. They found that increasing workload of GPU threads and increase number of groups -box- that each thread process makes a trade-off between execution time of CPU and GPU. They modelled the execution time and used it in [13] for finding the optimal size of group size.

The prior works [9][10][11] took decision about implementation based on their intuition and knowledge of common GPU optimization techniques and they proved the efficiency of their implementation by experimentation and empirical data. In [9] they conducted experiments on problems with different size (number of points) and tree level, but did not discuss the effect of point density on performance. The work [11] also listed the results of nine experiments with different tree levels and different densities. Their results indicates that in a problem with a fixed tree level, point density is positively corelated with speedup. However, they did not expand their result in details. One major reason for absence of performance modelling in prior works might be that GPU is fast enough even with weak optimizations. The works [12][13] on the other hand used execution time modelling that requires hardware dependant constants which are unknown before empirical tests.

In the literature of GPU cache optimization, it had been concluded that memory divergent applications suffer from high spatial locality but GPUs are not able to fully exploit it [15]. GPU caches are different from CPU caches since they are shared between many threads. This causes cache contention, false sharing and high miss-rate. In applications with irregular memory access patterns which don't use all data in a cache line [16]. Therefore, it is important to consider spatial locality as a major factor in modelling performance of GPU applications [17]. Computation of nearfield follows the stencil parallel pattern which is a kind of memory divergent application and hence it suffers from the similar false sharing problem. This fact was absent on previous researches on MLFMA. In [18] a novel approach proposed for decreasing miss-rate and increase bandwidth for applications with high degree of read only data sharing, in which the last level cache is reconfigured adaptively between shared versus private model. This adaptation is done through a light-weight performance prediction model on runtime. However, their approach was presented to serve private cache friendly applications -applications with high degree of inter cluster data sharing- which does not apply to P2P operator of MLFMA, the idea of data replication for reducing cache contention could be tested for P2P. This paper finds that modelling performance for data replication can predict speedup of algorithm based on data structure.

1.1 OUR CONTRIBUTION

As discussed in the previous section the main issue of P2P efficiency arises from its divergent memory accesses in device. The prior works tried to reduce memory usage by exploiting special specifications of GPU's memory. In this paper it is offered to restructure data - replicate data for each thread - in a way that less cache contention occurs. However proposed data redundancy introduces additional overhead, it speeds up GPU kernel because of more coalesced memory accesses. The overhead occurs in data collection phase in CPU. The volume of data transferred between GPU and RAM increases, i.e., any restructuring of data in GPU affects the way data is collected in CPU. In the other side, each thread warp faces less false sharing and cache contention.

To measure how the restructuring affects the overall speedup, it is necessary to use analytical models. While this effect had not been studied in prior works and requires another research, in this paper this effect is specifically studied for P2P operator of MLFMA. However, a performance model was presented in [13] but it is not applicable in our work because they focused on grouping threads not restructuring data. A performance model is presented in this work which is applicable to any data restructuring technique for P2P operator. The performance model relies on complexity of data collection algorithm on CPU, GPU kernel which retrieves data from memory and performs calculations and parameters of MLFMA. In this paper the locality of data access is proposed as an important factor in modelling the speedup of GPU implementations. This factor captures the effect of data restructuring on speed of data access on GPU. Although that the presented model is not accurate for predicting the exact speedup, but they give us insight on the effect of restructuring scheme. In our experiments, the box size, i.e., number of points inside a box is optimized to achieve the higher speedup. This is similar to work [13] that optimized the group size to achieve higher speed even though their work considers the whole stages of single level FMM algorithm running on heterogenous environment which is totally different from us. In our work the focus is on computation of P2P on a single GPU.

As a simple data restructuring technique, it is proposed to add redundancy to data and make all threads independent of each other, i.e., each data element is accessed by only one thread and each thread loads all data it requires by few instructions. Although that it makes extra data volume and makes data collection phase longer, but this overhead is captured by performance model and the optimum parameters that makes redundancy beneficial is achieved using the insight provided by the performance model.

The case study problem for evaluation of this paper's technique is electrical potential function for its ease of implement. The potential function is applied on a 2D PEC where the source and target (radiating and receiving) points are randomly distributed on the surface. The MLFMA formulation for this problem is taken from [14]. Because MLFMA is kernel independent, changing kernel wouldn't affect the results significantly.

Our novelty and contribution are briefly are:

- Proposing data restructuring as an optimization technique for P2P operator of MLFMA, which was not proposed yet.
- Building analytical performance prediction models to approximate effect of data restructuring of P2P operator on its speedup, which replaces previous empirical approaches.
- Introducing data access locality as an important factor in modelling speedup of algorithm on GPU, and propose a way to quantify the locality.

However, his article studies its novel proposed method using mathematical models which with our knowledge was not performed in any of previous works, it has limitation and drawbacks as:

- It doesn't model speedup of data transfer, which makes it dependant on empirical data.

- The simplicity of analytical modelling makes the proposed technique infrastructure dependent, i.e., the performance model relies on coefficients describing the unknown hardware parameters which also makes it dependant on empirical data.
- The proposed method is not automatic and applying it requires the effort of researcher to model performance squarely.
- Applying proposed methodology for larger problem using pipelining and on multiple GPUs is not studied in this article.
- The potential function used in this paper is easy to analyse and model. However, other kernels may face thread divergence or may use sharing memory, which were not considered behaviours in this article.

Considering these cons and pros, this research can be the beginning of future works on MLFMA that control speed of algorithm based on properties of device, data traversing algorithm, and MLFMA parameters.

In next section of this article, the simple implementation is presented by its analytical modelling, then the same is done for improved version. In section 3, accuracy of models is examined using empirical data. Then based on models, the efficiency of presented technique is tested on problems with various densities while running on a single GeForce 1050. Finally in sections 4,5 conclusions and future works are presented briefly.

2. PRESENTED TECHNIQUE

In this paper a performance model is built to measure the speedup of GPU kernel of P2P operator based on restructuring data. The speedup is approximated by division of two algorithms complexity, i.e., the basic algorithm and the redundant one. The complexity captures data structure, data collection algorithm, GPU kernel algorithm and problems parameters.

Here first a simple implementation of P2P operator is selected as basic model and then data redundancy is applied. The first implementation is called indexing method and the second one is called repetition method. These two typical methods are selected to approve the success of performance modelling. To simplify analytical models, none of known conventional GPU optimization techniques such as overlapping, pre-fetching, exploiting Shared Memory and dynamic parallelism were employed.

The work of [14] on solving 2D coulombic problems was used as basis of presented implementation. According to this work, a constructed tree is defined by three main parameters; N (number of samples), CT (clustering threshold) and L (three height). The clustering threshold keeps number of samples (points) per box lower than a specified number. If a box had number of samples more than CT in the process of building tree, then the level of tree increases by one so the box would have CT/4 samples. The maximum number of samples in all boxes is called t in this paper. The value of t is calculated after generating 2D mesh and is used in calculating complexity of algorithms.

Each method is described with three phases, data collection, data transfer and GPU kernel. The presented methods are expected to have opposite behaviour, i.e., the Indexing method execute faster in data collection and transfer but execute slower in GPU kernel while the Repetition method behaves oppositely.

2.1. INDEXING METHOD

In this method data is not replicated or each thread and all threads access non-contiguous memory banks. Data structure for this method compressed, hence data collection and data transfer phases are fast.

2.1.1 DATA COLLECTION

For Indexing method data are stored in seven arrays. Two of arrays are for coordination of all source and target points and one is for storing potential of source points. Other arrays keep indexes pointing to these arrays. The fourth array is for index of target points in all box. Index of source points is appended to this array while traversing boxes in Morton index. The fifth array is for starting index of each box in the previous array, namely second order index. The sixth array contains the index of the source points in neighbouring of each box in consecutive Morton order. The seventh array contains the second order index of previous array.

The algorithm (1) shows the execution order for data collection phase of Indexing method. The execution time of data collection in the CPU can be expressed by (1) where N is the problem size, m is the time of a single read or write operation in RAM, and B is the number of boxes. The number 9 in (1) refers to the number of adjacent neighbouring boxes in the 2D problem. The number of boxes is equal to 4^(L-1) where 4 is the branching factor of the tree and L is the height of the tree.

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

Algorithm 1 Data collection for Indexing method

```
Procedure P2P_Data_Collection_Indexing
  Input  tree t
  Output  targetPoints, soucePoints, potentials
          allTargets, allTargetsIndex
          allNeighburs, allNeighbursIndex
  targetPoints ← t.GetTargetPoints()
  sourcePoints ← t.GetSourcePoints()
  potentials = t.getPotentials()
  counter1 ← 0
  for all Box b ∈ t[lasteLevel].Boxes do
      for all Point p ∈ b.targetPoints do:
          allTargets.Append(p.index)
      end for
      allTargetsIndex[b] = counter1
      counter2 ← 0
      for all Box bj ∈ b.GetNeighburs()do
          for all sourec point s ∈ bj.souecPoints do
              allNeighbursIndex.Append(s.index)
              counter2 + +
          end for
      end for
      SoA.allNeighboursInBox.append(counter2)
  end for
end procedure
```

2.1.2 DATA TRANSFER

The total memory in Byte allocated for this technique is defined in (2). Point coordination and potential values are stored as Double while index values are stored as Integer. This data is transferred to GPU memory after data collection and before GPU kernel execution.

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

2.1.3 GPU KERNEL

In the indexing method, each GPU thread is responsible for calculating the near field for all target points inside a box. Each thread first extracts the second-order index of target points of its corresponding box, then extract the index of the target points in an iterative loop. In each iteration the thread extracts the second-order index of the source points corresponding to its box, and passes through them in an inner loop. Then it extracts their coordination and their potential, and finally applies the electric potential function to a pair of target and source points. Algorithm (2) displays execution order in Repetition methods GPU kernel.

Because each thread has access to data that is located in non-consecutive memory banks (seven arrays in different memory locations), memory requests lead to extreme cache misses and poor performance.

The execution time of each GPU thread is expressed in (4) where O_(1-computation) is the time of the potential function that is applied between two pairs of points. In the (4) the term t is used instead of CT because the maximum points in each box are less than or equal to t. The term t is a random variable and is not unique across different running of MLFMA, but it is certainly less than or equal to CT.

$$T_{Kernel_{Indexing}} = 4m_{Indexing} + t\left(3m_{Indexing} + 9t\left(5m_{Indexing} + O_{1-computation}\right) + m_{Indexing}\right) \quad (4)$$
$$= 4m_{ind} \times (11.25t^2 + t + 1) + 9t^2 O_{1-computation}$$

Algorithm2 GPU Kernel for Indexing method

```
Procedure P2P_GPU_Kernel_Indexing
  Input  bunBoxes
           targetPoints, sourcePoints, potentials
         allTargets, allTargetsIndex
        allNeighburs, allNeighbursIndex
  Output nearField
  tid ← threads index
  if tid < numBoxes do
    startTarget ← allTargetsIndex[ti]
    endTarget ← allTargetsIndex[ti + 1]
    startNeighbur ← allNeighbursIndex[ti]
    endNeighbur ← allNeighbursIndex[ti + 1]
    for i in startTarget: endTarget − 1 do:
      nearfield[ti] ← 0
      targetIndex ← allTargets[i]
      targetX ← targetPoints[targetIndex].GetX()
```

14: $targetY \leftarrow targetPoints[targetIndex].GetY()$
15: $for\ j\ in\ startNeighbur : endNeighbur - 1\ do:$
16: $sourceIndex \leftarrow allNeighburs[j]$
17: $sourceX \leftarrow sourcePoints[sourceIndex].GetX()$
18: $sourceY \leftarrow sourcePoints[sourceIndex].GetY()$
19: $u \leftarrow potentials[sourceIndex]$
20: $tX \leftarrow sourceX - targetX$
21: $tY \leftarrow source - targetY$
22: $nearfield[ti] += Interaction(tX, tY, u)$
23: $end\ for$
24: $end\ for$
25: $end\ if$
26: $end\ procedure$

2.2 REPETITION METHOD

In this method each thread computes an interaction between a target point and its neighbouring source points. In this method the source points around targets inside a box are replicated t times each of which for a target point. With this implementation, the time to collect and transfer data to the GPU is expected to take longer than the Indexing method, but due to serial memory requests, the GPU kernel is expected to run faster and neutralize the extra time in the collection and transfer phases.

Each target point has a maximum of 9 neighbouring boxes and each box contains at maximum CT source points. Therefore, the maximum number of the array elements assigned to each target point is equal to (2+1+9*3*t) of Double elements. The first two variables are the coordination of the target point, the next number is the number of nearby sources, and the next 9*3 members are 27 neighbours, including two coordinates and one potential value. All this data is stored in Double form but the second digit is read as an Integer.

Execution order for data collection of Repetition method is expressed in Algorithm (3). In data collection phase, for each box and for each target point inside it, all neighbouring source points are extracted and attached to data array.

The execution time of the collection phase is expressed by the following (6). Based on assumptions similar to those of the Indexing method, this relation can be summarized as (7).

$$T_{IterateAndCollect_{Repetition}} = B(3r + t(1r + 2w + find\_nei() + 1w + 9(1r + t(1r + 2w + 1r + 1w)) + 1w \quad (6)$$

$$T_{IterateAndCollect_{Repetition}} = 4^{L-1}(m_{Repetition} + 11tm_{Repetition} + 45m_{Repetition}t^2 + t * find\_nei()) \quad (7)$$

Algorithm 3 Data collection for 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: $end\ procedure$

2.2.2 GPU KERNEL

Each GPU thread is responsible for computing interactions between a target point and other source points in its close neighbourhood. Each thread requires the coordination of the target and source point, the number of source points in the nearby neighbourhood, and the potential of neighbouring source points. These elements are collected and stored consecutively for each thread, and data of all threads are stacked to form a single long array. Execution order for this kernel is displayed in Algorithm (4).

In this AoS way of storing data items, each thread requests for data in one or several adjacent memory banks. In the Indexing method data are stored in a SoA order where seven arrays were stacked and each thread accesses to seven non-adjacent memory locations. Therefor it is expected for Repetition method to face lower cache misses than Indexing method.

The execution time of each GPU thread can be shown in (9). Here $m_{Repetition}$ is equivalent to one memory access in the Repetition method. The term t is used instead of CT because each thread can access data from mostly t neighbouring source points.

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

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: $end\ procedure$

---

3 MODELLING THE SPEEDUP

The speed of execution caused by applying repetition method is obtained from the (10). Because there are factors related to memory access time in this relation and also the execution time is highly variable, it is difficult to calculate these expressions precisely. However, the (10) can be approximated as sum of separate speedups as shown in (11) where the values of α, β, γ are coefficients that can be evaluated based on hardware and operating system. By taking to account that these parameters converge to a constant value as problem size increases, then the estimation of parameters can be estimated using least square method.

$$X_{Repetition} = \frac{T_{Indexing}}{T_{Repetition}} = \frac{T_{Collection\_Indexing} + T_{Transfer\_Indexing} + T_{Kernel\_Indexing}}{T_{Collection_{Repetition}} + T_{Transfer\_Repetition} + T_{Kernel\_Repetition}} \quad (10)$$

$$X_{Repetition} = \alpha X_{Collecction\_Repetition} + \gamma X_{Transfer\_Repetition} + \beta X_{Kernel\_Reprtition} \quad (11)$$

3.1 MODELLING THE SPEEDUP OF DATA COLLECTION

The size of the data that is generated and sent to the GPU in Bytes is shown in (8). Here the term CT is used instead of t because for each thread a CT array element is reserved, hence each thread knows it's starting index.

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

Using (2) and (6), data collection speedup in the Repetition method is approximated using larger terms as expressed in (12) and (13).

$$\frac{T_{IterateAndCollect_{Indexing}}}{T_{IterateAndCollect_{Repetition}}} = \frac{N(7m_{Indexing}) + 4^{L-1}(m_{Indexing}(11 + 19t) + 3 + find\_nei())}{4^{L-1}(3m_{Repetition} + 11tm_{Repetition} + 45m_{Repetition}t^2 + t * find\_nei())} \quad (12)$$

$$\frac{T_{IterateAndCollect\_Indexing}}{T_{IterateAndCollect_{Repetition}}} \approx \frac{m_{Indexing}}{m_{Repetition}} \times \frac{1}{t} = X_m * \frac{1}{t} \quad (13)$$

where the speed of accessing RAM using the Repetition method compared to the Indexing method is denoted as X_m. This value is estimated as inverse ratio of the volume of the data in memory in (14).

$$X_m \approx \lambda \frac{Memory_{Indexing}+Memory_{result}}{Memory_{Repetition}+Memory_{result}} \approx \lambda \frac{40N+4^L(2+10t)+8N}{8N(3+27CT)+8N} \quad (14)$$

In (14) λ is a coefficient that depends on RAM and shows the effect of data volume on memory access time. (14) is simplified into (15). D is the average number of points in the box, and is obtained by dividing the total number of points N by the number of boxes L.

$$D = \frac{N}{4^{L-1}} \rightarrow X_m \approx \frac{1}{2} \times \frac{1}{D} \times \frac{10 \times \frac{1}{4} CT}{27CT} = \frac{1}{21.6D} \quad (15)$$

By combining (15) and (13) relations, the acceleration of data collection is obtained using the (16). According to this relation, the average density of points inside the boxes has a negative effect on the efficiency of the Repetition method in the data collection phase.

$$X_{IterateAndCollect\_Repetition} \approx \lambda \frac{1}{21.6tD} \quad (16)$$

3.2 Modelling the Speedup of GPU Kernel

In repetition method, a greater number of threads are used than Indexing method, that is about t times more. The overhead made by using threads more than device cores, is taken into account in speedup formulation.

The speedup of repetition method in executing GPU kernel is expressed in (17). By inserting (5) and (9) to (17) and using simplifications, the kernel speedup is approximated in (18).

$$X_{kernel\_Repetition} = \frac{T_{Kernel\_Indexing}}{T_{Kernel\_Repetition}} \times \frac{\frac{Num\ Threads\ Indexing}{Total\ Device\ Cores}}{\frac{Num\ Threads\ Repetition}{Total\ Device\ Cores}} = \frac{T_{Kernel\_Indexing}}{T_{Kernel\_Repetition}} \times \frac{4^{L-1}}{N} \quad (17)$$

$$X_{kernel_{Repetition}} = \frac{T_{Kernel_{Indexing}}}{T_{Kernel_{Repetition}}} = \frac{4m_{Indexing}(11.25t^2 + t + 1) + 9t^2 O_{1-computation}}{m_{Repetition}(9t+3) + 9tO_{1-computation}} \leq \frac{4m_{Indexing}}{3m_{Repetition}} t = \frac{4}{3} X_{mem\_repetition} \times t \quad (18)$$

The ratio of memory access time is considered constant in both Repetition and Indexing methods. It is difficult to accurately measure the memory access time but its speedup can be approximated. In this paper it is assumed that the most affecting parameter on memory access speedup is number of memory miss ratio. The memory miss ratio could be obtained from device counters but it is not beneficial in our implementation because the presented is not a kind of runtime optimization. Instead, miss rate is estimated as memory access locality. This reclaims that the more the data is located in close memory positions, the higher the cache hit rate achieves, and the slower access time to memory occurs.

$$X_{mem\_Repetition} \approx \lambda \frac{MissRatio_{Repetition}}{MissRatio_{Indexing}} \approx \lambda \frac{Locality_{Repetition}}{Locality_{Indexing}} \quad (19)$$

In (20) λ is a coefficient that depends on the GPU hardware and represents the performance of the cache. $MissRatio_{Repetition}$ is an estimation of the number of cache miss-rate, i.e., number of accesses to none neighbouring banks of memory by each thread. The number of cache miss-rate of a thread is defined as ratio of number of memory banks that a thread accesses to all memory banks that all threads access as expressed in (21).

$$Locality_{Indexing} = \frac{\#\ of\ memory\ banks\ each\ thread\ access}{total\ occupied\ memory\ banks} \quad (20)$$

In (21) if all data that a thread access are placed in less memory banks, then it faces less cache misses. Memory bank accesses in Indexing method are spread to non-contiguous memory banks because it uses SoA data scheme. In the Indexing method, each thread reads the values from seven arrays. Therefore, each threads makes at least 7 cache misses if N and t are large enough. An approximation of the indexing methods miss ratio is calculated in (21) where b is the number of bytes stored in a memory bank.

$$MissRatio_{Indexing} = \frac{\#MemoryBanks}{\left\lceil \frac{40N + 4^L(2+10t)}{b} \right\rceil} \quad (21)$$

$$\#MemoryBanks = \left\lceil \frac{1Integer}{b} \right\rceil + \left\lceil \frac{1Integer}{b} \right\rceil + \left\lceil \frac{t * Integer}{b} \right\rceil + \left\lceil \frac{9t * Integer}{b} \right\rceil + \left\lceil \frac{t * 2 * Double}{b} \right\rceil + \left\lceil \frac{9t * 3 * Double}{b} \right\rceil \quad (22)$$

In (22) terms are 2 indices that are placed in two different memory banks, t indices of target points, 9t indices of source points, 2t coordinate of target points and 27t coordinate and potential of source points. All of these elements are placed in separate memory banks. By taking the value of b as 512 Bytes for GTX1050 device, the (22) is simplified as (23).

$$MissRatio_{Indexing} \approx \frac{272t}{40N + 4^L(2+10t)} \quad (23)$$

This miss ratio is repeated for all other threads, so the value (23) must be multiplied by the number of threads in the Indexing method.

$$MissRatio_{Indexing} \approx \frac{272t}{40N + 4^L(2+10t)} \times 4^{L-1} \quad (24)$$

For Repetition method, all GPU thread read the same amount of data from the entire data array, and all of data are placed in contiguous memory banks. Therefore, the probability of a cache miss is very low and it can be roughly defined in (25).

$$MissRatio_{Repetition} = \frac{1}{N} \quad (25)$$

Putting (24) and (25) into ratio (19), ratio of cache misses in two methods is equals to (26). In this relation t could be excluded because its small value. By considering that $4^{L-1} \leq N$ the miss ratio is of order N^{-1}.

$$\frac{MissRatio_{Repetition}}{MissRatio_{Indexing}} = \frac{1}{N} \div \left(\frac{272t \times 4^{L-1}}{40N + 4^L(2+10t)}\right) = \frac{40N + 4^L}{N \times 4^{L-1}} \approx \frac{40}{4^{L-1}} + \frac{4}{N} \geq \frac{44}{N} \quad (26)$$

Putting (٢٦) in (19) gives an approximation of memory access speedup in (٢٧). By placing (٢٧) and (٢٥) in (19), the kernel acceleration of the Repetition method is calculated in (٢٨).

$$X_{mem\_Repetition} \approx \lambda \times \frac{44}{N} \quad (٢٧)$$

$$X_{kernel\_Repetition} = \frac{4}{3} X_{mem\_Repetition} \times t \times \frac{4^{L-1}}{N} \approx \frac{4}{3}\lambda \times \frac{44}{N} \times t \times \frac{4^{L-1}}{N} \approx 55.3\lambda \times t \times \frac{1}{ND} \quad (٢٨)$$

It can be argued that the speedup of the GPU kernel in the Repetition method decreases with increase in size of the problem. Since the value of t is greater than D, increasing the point density in (٢٨) has a positive effect on the GPU kernel speed and can overcome negative effect of t.

3.3 Modelling Data Transfer Speedup

The data transfer time is influenced by other factors such as hardware and operating system, but in general it is proportional to volume of data. The data volume ratio is calculated in (15) and shows that by increasing the density of points in the boxes, the speed of the Repetition method decreases.

3.4 Analysis of Model

The total speedup is calculated by (11) and by placing (15) and (16) and (2۸) relations as expressed in (2۹) where λ_{RAM} is related to performance of RAM and λ_{GPU} is related to performance of GPU DRAM. The variable t plays opposite roles in accelerating the data collection and accelerating the GPU kernel. This means that Repetition method is not always faster than the Indexing method. The value $\frac{1}{D}$ can also be factored from the whole relation. It concludes that increasing the number of boxes while keeping the N constant (and therefore decreasing D), makes the Repetition method in overall faster than Indexing method. To find a balance point between the two, it is necessary to know the value $\lambda_{RAM}$and λ_{GPU} and coefficients α, β.

Based on the proposed analytical model, it can be suggested that in order to exploit the acceleration of Repetition method, the number of boxes must be increased while keeping N constant, and the number of sample points inside them must be reduced; or it can be said that this method is better than Indexing method for problems where the density of the sample points is low.

$$X_{Repetition} = \alpha\lambda_{RAM}\frac{1}{21.6Dt} + \gamma\frac{1}{21.6D} + \beta(55.3\lambda_{GPU} \times t \times \frac{1}{ND}) \quad (2۹)$$

4 Evaluations of Model

4.1. Evaluation of Model Error

In this section the basic CPU implementation taken form [14] is compared with Indexing method and repetition method. In CPU implementation the boxes are traversed in Morton's indexing order, and for each target point in a box, its neighbouring source points are first extracted and for each source point in the E_1 neighbourhood, the potential function is executed once and its value is added with the far-field induced potential.

4.1.1. Evaluation of Data Collection Performance Model

Because the data collection is done in the CPU and is too experimental setup, the presented techniques are compared on small sized problems. Tests start with a problem of size N=5,000 to 100,000 points increasing by 5,000 at each step. Then the problem size is increased to 350,000 points with an increase of 50,000 points in each step. Default value for CT is 15 since presented technique is more efficient for sparce problems. Tree height L starts with default value of 3 and increases based on N and CT. Each test is repeated 20 times on similar tree and the running times are averaged over 20 runs.

First speedup of each method is obtained by dividing its running time by the baseline model's running time. Then speedup of Repetition method is calculated by dividing its speedup by Indexing methods speedup.

Figure 1 shows the speedup of Repetition method. The speedup is also calculated approximately in (29) based on values of t and D. The measured speedup is plotted with a solid black line in Figure 1. The vertical axis on the left shows the amount of speedup per size of the problem N. The orange dashed line is the estimated speedup using the (16) and the vertical axis on the right shows the corresponding speedup using (28) In both plots, the presented performance models can follow the same trend that empirical data follows. To fit them more accurately, it is required to find the values of λ for both (16) and (28). In this paper they are calculated by dividing the empirical runtime by performance model and then take the average value over all experiments. The value of λ for formula (16) is between 75 and 180 and the average value is 108.

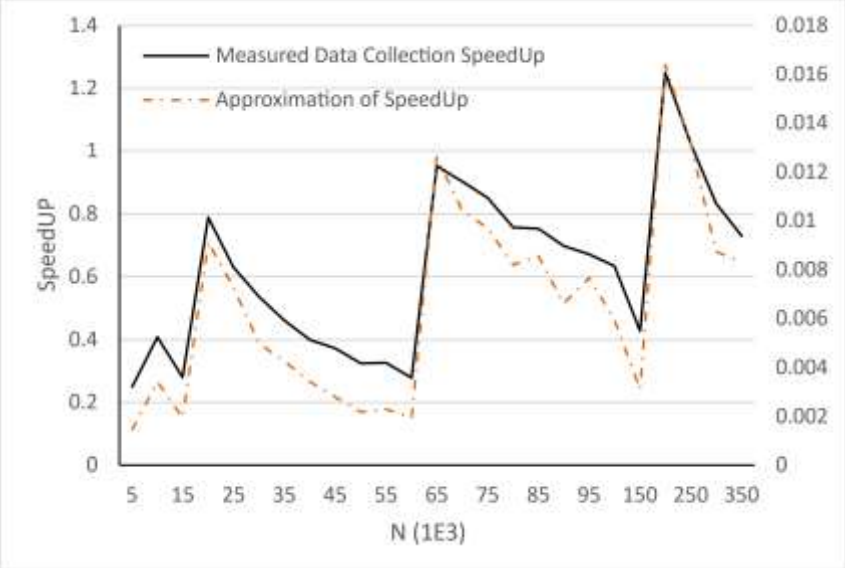

Fig. 1 – measured Repetition methods speedup (black line, left vertical axis) in data collection phase, and its estimated value using (16) (orange line, right vertical axis). The estimated value has similar trend with measured one.

The reason for the oscillation of the graph (1) is because of the density D. According to the (15), the data volume is directly related to the inverse of D. The volume of data affected by the value of D has an important effect on the execution speed of data collection. Given this value and the (16), the maximum execution speedup theoretically is expressed in (30).

$$X_{IterateAndCollec\_Repetition} \approx \lambda \frac{1}{21.6tD} \geq 1 \qquad (30)$$
$$\frac{CT}{4} \sim 4 \leq t, 1 \leq D \rightarrow X_{IterateAndCollect\_Repetition} \leq 1.25$$

4.1.2. Evaluation of GPU Kernel Performance Model

Since GPU kernel execution is much faster than collecting data on the CPU, in this section experiments are conducted on larger problems. In these tests N is increased by 1,000 per step beginning from 1,000 points until 100,000 points, then it increases with 50,000 points per step until 1,000,000 points. Value of CT and L remains constant in all tests.

Figure 2 illustrates the calculated speedup of the GPU kernel of Repetition method compared with Indexing method. The speedup is plotted in black solid line. The left axis shows the speedup on logarithmic scale in order to compensate for the effect of jumps in the value of N on the resolution of the graph. The orange dashed line shows the predicted value according to (28). This value has an overall trend similar to the measured value, but does not follow it in partial fluctuations. One of the reasons for this error is that the runtime on the GPU does is not a n accurate value.

The largest value of N where Repetition method is still faster or equal to Indexing method based on (28) and with assuming that $t \leq CT$, is calculated in (31).

$$X_{kernel\_Repetition} \approx 55.3\lambda \times t \times \frac{1}{ND} \geq 1 \rightarrow N \leq 55.3\lambda \times \frac{t}{D} \qquad (31)$$
$$t \leq CT = 15 \rightarrow N \leq 829.5\lambda \times \frac{1}{D}$$

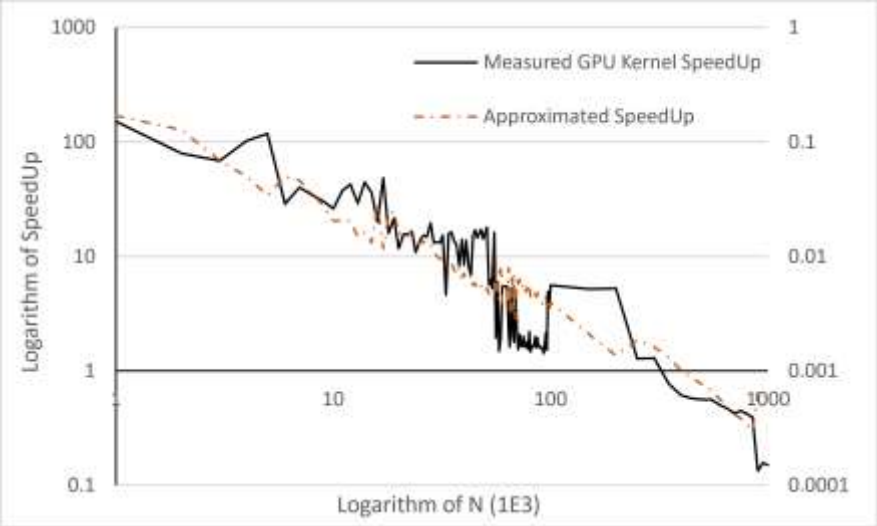

Fig. 2- measured GPU Kernel Speedup of Repetition method (Black Line, Left Vertical Axis) based on Indexing method, and its estimated value using relation 28(Orange Line, right vertical axis). Both axes are displayed on logarithmic scale. Trend.

The exact value of this relation requires the value of D and λ. Value of λ is calculated by dividing empirical data and approximated data for each experiment and taking its average value. Value of λ for GPU memory is about 640 based on experimentations. This number shows that for each unit of improvement in locality, the P2P kernel runs about 640 times faster. According to empirical data, the value of D fluctuated between 1 and 5. Given the maximum value for D, the optimal value N (assuming that t is always 15) could be calculated based on (28) as expressed in (32).

$$106{,}000 \leq N_{Optimal} \leq 530{,}000 \tag{32}$$

According to Figure 2 this value is not accurate, however it is enough accurate to select then chunk size for distributing the whole problem among multiple GPUs.

4.1.3. Estimation of Model Coefficients

In (29) calculation of the coefficients α, β, and γ allows to estimate the total acceleration time. These coefficients are calculated using data obtained from the test results performed for the data collection and GPU kernel execution phases and are expressed in (33).

$$\alpha \approx 0.82, \beta \approx 0.09, \gamma \approx 0.18 \tag{33}$$

In Figure 3, the total execution time (equivalent to the total time of data collection, data transfer and kernel execution) and the predicted value are shown according to coefficients in (33). In this Fig. the predicted values have been multiplied by empirically measured speedups not theoretically values, that is because the (29) is not accurate for transfer time. Formulation in (29) instead is useful for obtaining some insights on the effect of the algorithm design and the choice of granularity on the overall speedup. Figure 3 confirms that the idea of separation of speedups presented in (11), however it is not accurate since transfer time is modelled accurately.

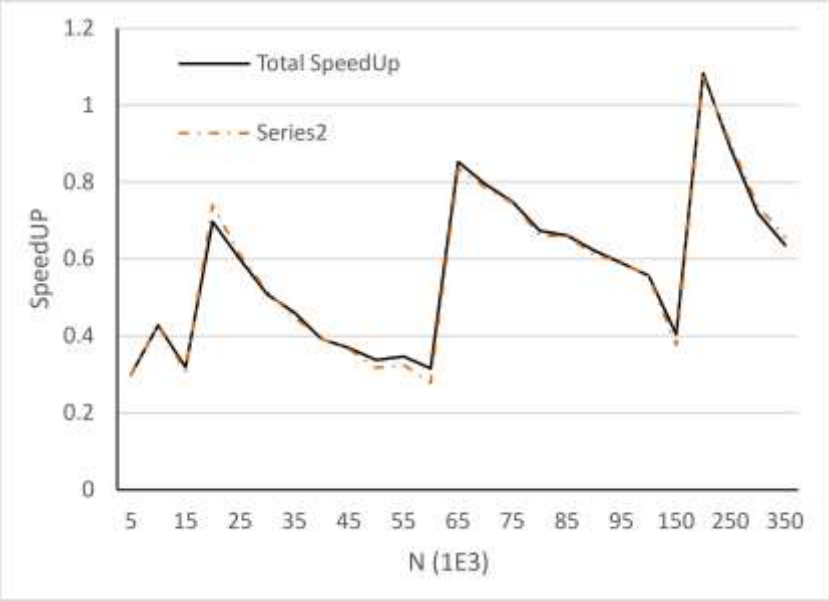

Fig. 3- total speedup of Repetition method (black line) and its estimated values according to empirical data and (33) (orange line).

The overall trends in Figure 3 is very similar with the Figure 1, i.e., data collection time is the bottleneck of the presented method with dedicated values for N,CT. The same argument can be obtained from coefficients in the (33). The GPU kernel speedup has the smallest contribution to the overall speedup. According to and Figure 2 and 3, the Repetition method is slightly faster than the Indexing method when t and D are not manipulated directly and are increased by increase in value of N and CT.

4.2. maximizing Total Speedup

Based on the results of the previous section and the (29) that summarize the analytical relations of this work, it is suggested to reduce the value of D and t while keeping N constant, to effectively use the repetition method in a particular problem. A simple way to do this is to increase the height of the tree by at least one unit after constructing the tree according to CT, i.e., increase the number of boxes by a factor of 4 (based on tree branching factor) and make D and t to a quarter of their value. By increasing the height of the tree by i units, according to the (29) and by assuming that the coefficients remain the same, the resulting acceleration will be equal to:

$$L' = L + i \rightarrow t' = \frac{t}{4^i}, D' = \frac{D}{4^i} \tag{34}$$

$$X'_{Repetition} = \alpha\lambda_{RAM}\frac{4^{2i}}{21.6Dt} + \gamma\frac{4^i}{21.6D} + \beta\left(55.3\lambda_{GPU} \times t \times \frac{1}{ND}\right) \approx 4^{2i} \times 0.82 + 0.18 \times 4^i + 0.09 \tag{35}$$

This means that the data collection speed and thus the data transfer speedup significantly increases, while the speed of the GPU kernel remains unchanged. The (35) shows that by increasing one unit of tree region by one, the Repetition method becomes more than 13 times. These numbers just give us an idea, and at runtime the acceleration times are not necessarily equal to these numbers.

To evaluate the effect of varying tree height and box density, another experiment is performed this time. For Values in (36) different trees are generated with CT=15. After building the tree, L increases or decreases based on i . For larger values of i, the boxes become smaller and t becomes lower. In the opposite side for smaller values of i the value of t and density increases. The value

of N in these problems is 4^{L-1} where L is taken as its initial value. The baseline CPU implementation is used here which is the same baseline model used in the previous section.

$$i \in \{-3,-2,-1,0,1,2,3\}$$
$$L \in \{4,5,6,7,8,9,10,11\} \quad (36)$$

In Figure 4,5,6 a comparison is made between execution time of the presented methods and baseline model, and also between GPU implementations. Figure 4 shows that the Indexing method is slower than the baseline method for low-density trees where $i \geq 1$, but for high density trees in large problems where $N \geq 65{,}536 \; and \; i = -3$ it is about 22 times faster than baseline method. Figure 5 shows that the speedup of the Repetition method was always faster than the baseline model, which shows its general efficiency, but its maximum acceleration is less than 3 when $i = -3$ compared to the baseline model.

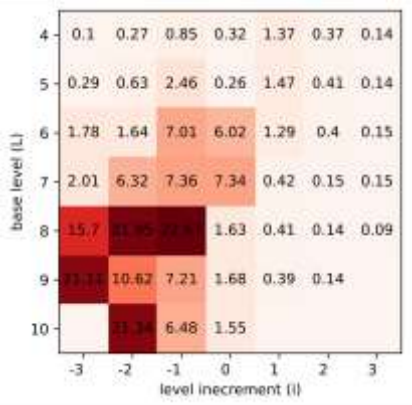

Figure 4 - Speedup of Indexing method relative to the baseline model

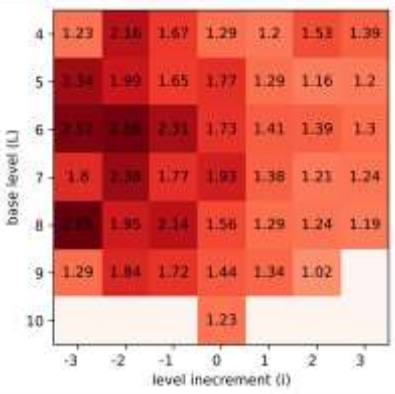

Figure 5 - Speedup of Repetition method relative to the baseline model

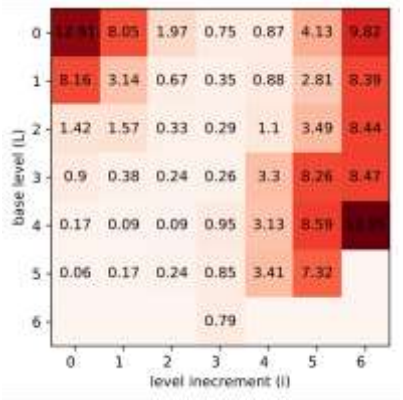

Figure 6 Speedup of Repetition method compared to Indexing method

According to Figure 6, the Repetition method is faster than the Indexing method in two regions. In the first region density is high but the problem size N is smaller than 4096. According to (28) it this region value of N is smaller which makes GPU kernel faster for Repetition method and also boxes are denser and again GPU kernel executes faster. It can also be argued that due to the small problem and the redundancy in Repetition method, the data of several consecutive threads fits in a single cache line. This reduces the cache miss rate and increases the kernel speed.

The second region is where the tree density decreases while the size of the problem increases, i.e., where $N \geq 16{,}384 \; and \; i \geq 2$. In this case, the GPU kernel runs slower according to value of N, but data transfer and data collection becomes faster due to their inverse relation with value of D and t.

In this work, two GPU implementations of P2P operator with different data structures were studied, once with compressed data and the other with redundancy in data. Since any change in data restructuring affects the data collection time in CPU, analytical performance models have been presented to track the effect of changes in algorithm design on speedup of algorithm and find the optimal value of problem parameters.

5. CONCLUSION

Accuracy of models were evaluated by showing their overall trend line. However, they were not accurate and were dependent to system setup, but they gave insight for better design of algorithm.

The optimization of MLFMA tree by controlling t and D was proposed by analytical models and the result was tested empirically. According to the empirical data, the presented modification of the algorithm archives almost 13 times speedup based on unmodified algorithm for problems with more than 200,000 source and target points and 2-4 points per box. This result was conformed to analytical modelling.

6. FUTURE WORKS

It is suggested to apply this modelling technique for other techniques of P2P or other operators of MLFMA in future works.

Also, it is recommended to predict values of λ based on device features. This bridge the gap between algorithm design and hardware specifications and helps to distribute the whole problem on more than one GPU efficiently.

Modelling of data transfer between GPU and RAM was not precisely done in this paper, however building such precise model can complete puzzle of analytical modelling.

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.

Authors

Morteza Sadeghi

He is a Ph.D. candidate in Algorithm and Computations, University of Tehran, IRAN

Seyed Abdolreza Torabi

He is an Assistant Professor in the Department of Engineering Science, University of Tehran, IRAN