Fast Multipole Method for Large-Scale Electromagnetic Scattering Problems on GPU Cluster and FPGA-Accelerated Platforms

V. Dang, Q. Nguyen, and O. Kilic

Department of Electrical Engineering and Computer Sciences
The Catholic University of America, Washington, DC 20064, USA
13dang@cardinalmail.cua.edu, 93nguyen@cardinalmail.cua.edu, and kilic@cua.edu

Abstract ─ The fast multipole method (FMM) for large-scale electromagnetic scattering problems is implemented on high performance computing (HPC) platforms and its performance is investigated in terms of accuracy, speedup, and scalability. The HPC platforms include a 13-node graphical processing unit (GPU) cluster, and a field programmable gate array (FPGA)-based high performance reconfigurable computer (HPRC). The details of the implementations and the performance achievements are shown and analyzed. We demonstrate a scalable parallelization while maintaining a good degree of accuracy.

Index Terms - Electromagnetic scattering, fast multipole method (FMM), FPGA, GPU, high performance reconfigurable computer (HPRC), iterative solvers, and method of moments.

I. INTRODUCTION

Modeling large-scale objects is a challenging problem in electromagnetics community due to the excessively heavy requirements of memory and computational resources. Since it has an important role in the research of target identification, or the stealth and anti-stealth technology, many numerical techniques have been developed over past two decades to reduce this burden without significant loss of accuracy, including adaptive integral method (AIM) [1], impedance matrix localization (IML) [2], and fast multipole method (FMM) [3]. Compared with the others, FMM is among the most suitable techniques for large-scale problems. It reduces the computational complexity of method of moments (MoM) from $O(N^3)$ to $O(N^{3/2})$ where N denotes the number of unknowns, whereas AIM and IML have the complexity of $O(N^{3/2}\log N)$ and $O(N^2\log N)$, respectively.

Many authors have investigated the parallelization of FMM and its multi-level version (MLFMA) [4-12] on CPU clusters in solving problems of hundreds of thousands to millions of unknowns. Others used FMM to solve large acoustic problems on multi-node GPU systems [13-14], or implemented MLFMA on GPUs, [15].

This paper is a continuation of our previous efforts [16, 17], and investigates the parallelization of FMM for electromagnetics structures on two HPC platforms. The first platform includes 13 nodes populated with a Nvidia Tesla M2090 GPU. The second platform is an FPGA-based SRC-7 system, which includes a single Altera Stratix EP4SE530 FPGA. In this paper, we enhance the GPU cluster implementation in [16] by the use of the two workload partitioning techniques among the computing nodes, namely group-based and direction-based distributions. The group-based distribution technique is applied for the calculation of the near components of the impedance matrix, while the direction-based distributions are used in the far component computation as will be discussed later in the implementation section. Our previous work in [16] utilized only the group-based approach, which resulted in more communication overhead. Regarding the FPGA platform, this paper provides an entire FMM implementation whereas our previous work in [17] utilized the FPGA only for the near component calculations of the impedance matrix. More details of the entire FMM implementation on FPGA is provided in the implementation section.

Submitted On: July 23, 2013
Accepted On: Jan. 19, 2014

For the sake of validation of accuracy with analytical methods, the work presented here focuses on canonical problems such as scattering from a sphere. The work can easily be extended to real-life problems involving complicated structures.

The rest of the paper is organized as follows. Section II provides an overview of FMM. We present the implementation of FMM on HPC platforms in section III. Performance metrics for evaluation are presented in section IV. The experimental results are discussed in section V, followed by the conclusions in section VI.

II. OVERVIEW OF THE FAST MULTIPOLE METHOD (FMM)

The fundamental principles of FMM and its applications in electromagnetics have been well studied in literature [3-4]. In this section, we provide a brief overview to help our discussion on its parallel implementations discussed in section III.

Like in MoM, FMM solves for the linear equation system created in the form of $ZI = V$ where I represents the unknown currents, V depends on the incident field, and Z is the impedance matrix. The main idea in FMM is the grouping concept as shown in Fig. 1, where the N edges in the mesh of a given structure are categorized into M localized groups based on their proximity. According to this approach, two interaction types can be defined: near and far, as depicted in Fig. 1. These different types allow the system matrix to be split into two components, Z_{near} and Z_{far}, as shown in equation (1),

$$ZI = \left(Z_{near} + Z_{far}\right)I = V. \tag{1}$$

The near term comprises of interactions between spatially close edges, and is computed and stored in a similar manner to MoM [18]. The interactions between the remaining edges that are spatially far from each other constitute the far term. The advantage of separating the Z matrix into two components is that the Z_{far} matrix does not need to be computed and stored ahead of time. Instead it is factorized into radiation, T_1^E, T_2^E, receive, R_1^E, R_2^E and translation functions, T_L. Equation (2) depicts these functions based on the electric-field integral equation (EFIE) formulation,

$$Z_{far} = \frac{jk\eta}{4\pi}\left(\int d^2\hat{\mathbf{k}}\; R^E_{1,\mathbf{r}_{im}}\left(\hat{\mathbf{k}}\right).T_L\left(k,\hat{\mathbf{k}},\mathbf{r}_{mm'}\right).T^E_{1,\mathbf{r}_{i'm'}}\left(\hat{\mathbf{k}}\right) - \frac{1}{k^2}\int d^2\hat{\mathbf{k}}\; R^E_{2,\mathbf{r}_{im}}\left(\hat{\mathbf{k}}\right).T_L\left(k,\hat{\mathbf{k}},\mathbf{r}_{mm'}\right).T^E_{2,\mathbf{r}_{i'm'}}\left(\hat{\mathbf{k}}\right)\right), \tag{2}$$

where

$$T^E_{1,\mathbf{r}_{i'm'}}\left(\hat{\mathbf{k}}\right) = \int_S \mathbf{f_n}\left(\mathbf{r}_{i'm'}\right).e^{j\hat{\mathbf{k}}\cdot\mathbf{r}_{i'm'}}\,dS',$$
$$T^E_{2,\mathbf{r}_{i'm'}}\left(\hat{\mathbf{k}}\right) = \int_S \nabla'.\mathbf{f_n}\left(\mathbf{r}_{i'm'}\right).e^{j\hat{\mathbf{k}}\cdot\mathbf{r}_{i'm'}}\,dS', \tag{3}$$

$$R^E_{1,\mathbf{r}_{im}}\left(\hat{\mathbf{k}}\right) = \int_S \mathbf{f_n}\left(\mathbf{r}_{im}\right).e^{-j\hat{\mathbf{k}}\cdot\mathbf{r}_{im}}\,dS,$$
$$R^E_{2,\mathbf{r}_{im}}\left(\hat{\mathbf{k}}\right) = \int_S \nabla.\mathbf{f_n}\left(\mathbf{r}_{im}\right).e^{-j\hat{\mathbf{k}}\cdot\mathbf{r}_{im}}\,dS, \tag{4}$$

$$T_L\left(k,\hat{\mathbf{k}},\mathbf{r}_{mm'}\right) = \frac{k}{4\pi}\sum_{l=0}^{L}(-j)^{l+1}(2l+1)\, h_l^{(2)}\left(k\left|\mathbf{r}_{mm'}\right|\right)P_l\left(\hat{\mathbf{k}}\cdot\mathbf{r}_{mm'}\right). \tag{5}$$

In the equations above, the prime syntax denotes the source points, and i and m are indices that refer to the edges and groups in the mesh, respectively. The vector $\mathbf{r}_{ab}$ implies the direction from point b to a. The unit vector $\hat{\mathbf{k}}$ denotes the K possible field directions in κ space, $\mathbf{f}(\mathbf{r})$ denotes the associated basis function, $h_l^{(2)}(x)$ is the spherical Hankel function of the second kind, and $P_l(x)$ is the Legendre polynomial.

At this stage, the Z matrix is known and the unknown values for I can be solved for iteratively using equation (1). Each component of the voltage term V_i is calculated as in equation (6) using the matrix-vector multiplication (MVM),

$$V_i = \sum_{i'=1}^{N} Z_{ii'}I_{i'} = \sum_{i'=1}^{N} Z_{near,ii'}I_{i'} + \sum_{i'=1}^{N} Z_{far,ii'}I_{i'}, \tag{6}$$

where the near component is based on MoM and the far component is computed from equation (7) as,

$$\sum_{i'=1}^{N} Z_{far,ii'}I_{i'} = \frac{jk\eta}{4\pi}\left(\int d^2\hat{\mathbf{k}}\; R^E_{1,\mathbf{r}_{im}}\left(\hat{\mathbf{k}}\right).\sum_{m'\notin B_m} T_L\left(k,\hat{\mathbf{k}},\mathbf{r}_{mm'}\right).\sum_{i'\in G_{m'}} T^E_{1,\mathbf{r}_{i'm'}}\left(\hat{\mathbf{k}}\right)I_{i'} - \frac{1}{k^2}\int d^2\hat{\mathbf{k}}\; R^E_{2,\mathbf{r}_{im}}\left(\hat{\mathbf{k}}\right).\sum_{m'\notin B_m} T_L\left(k,\hat{\mathbf{k}},\mathbf{r}_{mm'}\right).\sum_{i'\in G_{m'}} T^E_{2,\mathbf{r}_{i'm'}}\left(\hat{\mathbf{k}}\right)I_{i'}\right), \tag{7}$$

where G_m denotes all elements in the m^{th} group, and B_m denotes all nearby groups of the m^{th} group.

The details of the parallelization of FMM will be discussed in the following section.

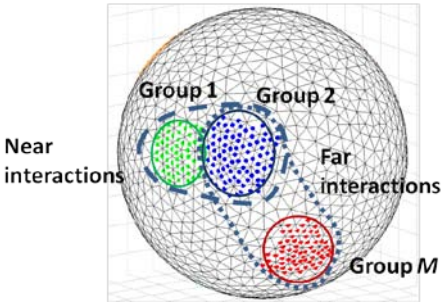

Fig. 1. Grouping in FMM (*N* edges, *M* groups).

III. HPC IMPLEMENTATION OF FMM

We consider two platforms for the parallel implementation of FMM; namely a GPU cluster and an FPGA-based HPRC system. In this section we first discuss the different architectures and programming models of these systems, and then provide the details of our HPC implementations on both platforms.

A. HPC architectures and programming models

The first platform utilized in our FMM implementation is the GPU cluster, which consists of 13 computing nodes. Each node has a dual 6-core 2.66 GHz Intel Xeon processor, 48 GB RAM along with one Nvidia Tesla M2090 GPU running at 1.3 GHz with 6 GB of GPU memory. The nodes are interconnected through the InfiniBand interconnection, as shown in Fig. 2. The cluster populates CUDA v4.2 and MVAPICH2 v1.8.1 (a well-known implementation of message passing interface (MPI)). Two parallel programming approaches of CUDA and MPI are combined to provide the use of GPU programming across the cluster.

The second platform is an SRC-7 MAPstation workstation, which consists of one general purpose microprocessor subsystem and one series J MAP reconfigurable processor subsystem, see Fig. 3. The microprocessor board, which is based on a dual-core 3.00 GHz Intel Xeon processor and 6 GB RAM, is connected to the MAP board through the series D SNAP interconnect. The SNAP card plugs into the memory DIMM slot on the microprocessor motherboard to provide a high data transfer rate between the boards. The MAP board is composed of one control FPGA (Altera Stratix EP2S130) and one user FPGA (Altera Stratix EP4SE530), which operate at 200 MHz. It also contains 16 on-board memory (OBM) banks with a total capacity of 64 MB, and two simultaneously accessible 1 GB global common memory (GCM) banks. SRC's proprietary Carte-C programming environment is used in the FPGA-based code development. The programming model offers a compromise between high-level languages (HLLs), e.g. C, and hardware description languages (HDLs), e.g. VHDL, to abstract underlying hardware design details and streamline the disparate design flows [19].

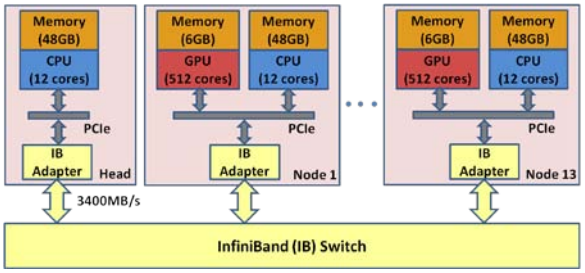

Fig. 2. System architecture of GPU cluster.

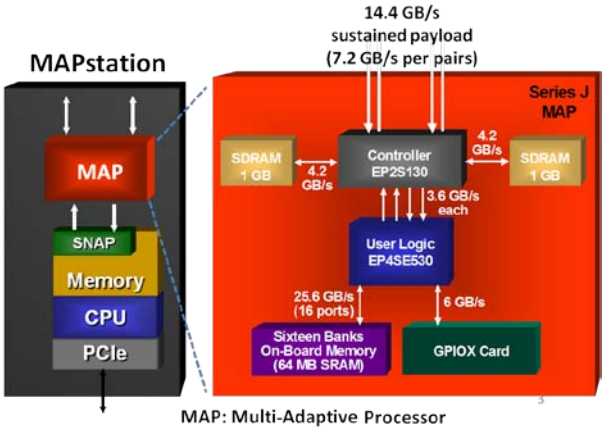

Fig. 3. System architecture of SRC MAPstation.

The two platforms utilize two different programming models. The CUDA used by the GPU follows the single program multiple data (SPMD) model [20], which allows data parallelism (wide parallelism). Under the SPMD scenario, thousands of threads execute the same program on different GPU cores, simultaneously operating on different data sets in parallel. On the other hand, FPGA programming mainly supports

task parallelism (deep parallelism) by utilizing pipelining technique. It is interesting to mention that data parallelism can also be allowed in FPGA programming. However, this depends on the availability of on-chip resources. The SRC's Carte programming environment allows the user full control of data utilization in terms of pipelining and parallelization, whereas with the CUDA environment the user is oblivious to how the GPU is scheduled. Thus, CUDA is very easy to use whereas Carte requires both software and hardware programming skills.

B. HPC implementations

The FMM algorithm comprises three main steps: pre-processing, processing, and post-processing (see Fig. 4). The pre-processing step involves reading the geometry mesh and dividing edges into localized groups. The processing step involves five tasks as shown in Fig. 4. The matrix components for near interactions, the radiation/receive functions, the translation matrix, and the V vector are calculated and stored. Iterative methods, such as biconjugate gradient stabilized method (BiCGSTAB) [21], are employed for solving the linear system. Finally, the electromagnetic quantities of interest, e.g. scattered fields, are calculated in the post-processing step. Based on our profiling results, the processing step consumes the most execution time in the algorithm. Hence, it is selected as the candidate for hardware parallelization, while the other steps are handled on the CPU.

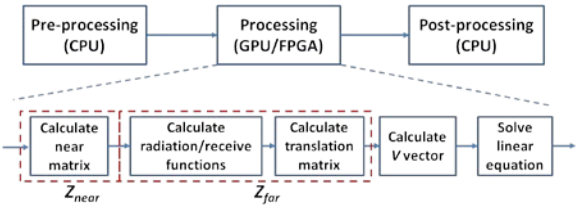

Fig. 4. FMM processing step flowchart.

B.1. GPU cluster implementation

The geometry mesh data resulting from the pre-processing step in Fig. 4 is transferred to the GPU memory once at the beginning of the processing step and the entire computation is performed on the GPU afterward. The parallelization of the processing step in GPU cluster implementation is performed at two levels: (i) among computing nodes using MPI library, and (ii) within GPU per node using CUDA programming model. The workload of computational tasks in the processing step, as shown in Fig. 4, are equally distributed to computing nodes such that each node holds the same amount of workload and the inter-node communication is minimized. Two partitioning techniques, which are defined as group-based distribution and direction-based distribution, are exploited to achieve the balanced workload distribution among the computing nodes. The first technique involves the uniform distribution of M groups among n computing nodes. The second technique, which was suggested in [5], involves the distribution of independent computation for each sample in κ space among the nodes. Within each node, the CUDA thread-block model is utilized to calculate the workload assigned to that node. The remaining parts of this section highlight the implementation details of each computational task in the processing step.

1) Near interaction calculations

Our earlier work [22] on implementing MoM on multiple GPUs is leveraged for the first task of the processing step (see Fig. 4), namely calculation of Z_{near}, which utilizes conventional MoM. Using the group-based partitioning technique, the rows of the Z_{near} matrix are assigned to the computing nodes with the assumption that each node has approximately an equal number of Z_{mn} elements, as shown in Fig. 5, where N_{group} denotes the average number of edges per group, and M_{node} is defined as the average number of groups per node [16]. At a given node, each sparse row is handled by a CUDA block in which a CUDA thread calculates one element, Z_{mn}, of that row. For further details of the GPU cluster implementation, the readers are referred to our previous work in [16-17].

2) Far interaction calculations

The second task is the calculation of the far interactions, which consists of the calculations of three functions: radiation, T^E, receive, R^E, and translation, T_L. As seen in equations (3) and (4), the radiation and receive functions are in the form of complex conjugate of each other. Thus their implementations are identical. In contrast to our

previous work in [16-17] in which the group-based partitioning was used for far interactions, the direction-based partitioning strategy is applied to the GPU implementation of this task due to the fact that each sample in κ space is completely independent of other samples. This efficient workload distribution guarantees the minimum communication at the end of the matrix-vector multiplication as discussed later in the next section. As shown in Fig. 6, following the κ space distribution, each node handles calculations of M groups for K_{node} ($\approx K/n$) directions. Given this amount of workload per node, the CUDA kernel is launched with $M \bullet K_{node}$ blocks such that each block performs N_{group} radiation/receive function calculations at a given direction and each thread evaluates a single function.

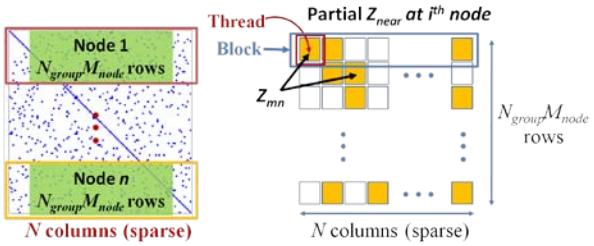

Fig. 5. Workload distribution and CUDA implementation of the Z_{near} matrix.

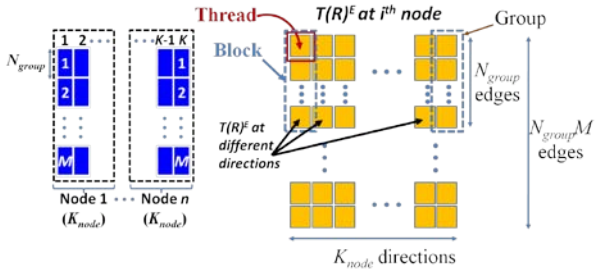

Fig. 6. Workload distribution and CUDA implementation of the radiation/receive functions.

The next task for the far interactions is the calculation of the translation matrix, T_L, which contains all the translation operators among far groups. The workload of the T_L matrix calculations is also distributed across the nodes following the direction-based technique since it has to be evaluated for the entire κ space. Each CUDA block is assigned to compute one sparse row of the T_L matrix for a given direction and each thread computes one element in that row, see Fig. 7.

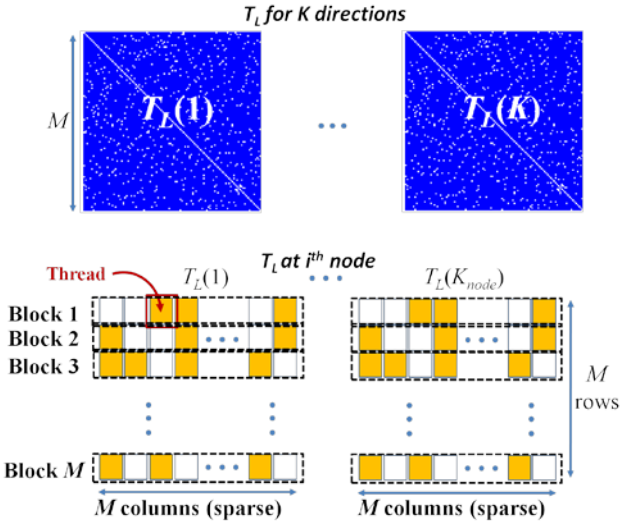

Fig. 7. Workload distribution and CUDA implementation of the translation matrix.

3) V vector calculations

The V vector calculations are simply the evaluations of incident electric fields at each triangle in the geometry mesh. The details of the computation, which render itself to parallel implementation can be found in [18].

4) Solution of the linear equation

The final task is the solution for the linear equation using iterative solver, i.e., BiCGSTAB algorithm [21]. In this algorithm, each iteration involves matrix-vector multiplications (MVMs), i.e., $Z.I$, which constitute the most time consuming part of the solution. The calculation of $Z_{far}I$ comprises three stages: aggregation, translation, and disaggregation, (the readers are referred to [3] for further details), while $Z_{near}I$ is simply a regular sparse MVM. This section only discusses the GPU cluster parallelization of $Z_{far}I$, as shown in Fig. 8. It should be noted that the other parts of the BiCGSTAB are also performed on GPU. They involve basic linear algebra operations leveraged from the CUBLAS library [23].

The unknown currents are distributed across the computing nodes using the group-based partitioning technique. During the iterative linear solution, each node calculates the estimated values

of its assigned unknowns and updates all nodes. In the aggregation stage, each node computes the radiated fields for all M groups for the K_{node} directions by multiplying the unknowns with their corresponding radiation functions, T^E, and accumulating within each group. The CUDA execution model assigns one group for a specific direction to each block in which each thread performs one multiplication. The CUDA kernel also requires parallel reductions within every block to sum the per-thread results to give the radiated fields for each group.

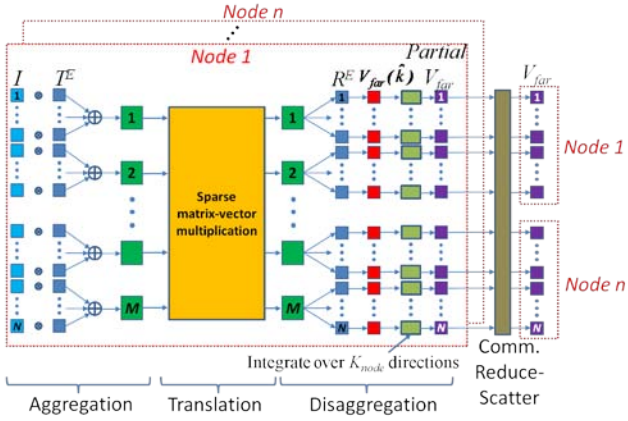

Fig. 8. Parallelization of the far MVM.

In the translation stage, the received fields for each associated direction are calculated from the multiplication of the translation matrix, T_L, and the radiated fields. Since the translation matrix is sparse, the CUDA implementation simply performs K_{node} normal sparse MVMs.

In the disaggregation stage, the received field quantities of all M groups are multiplied with the corresponding receive functions, R^E, and integrated over the partitioned κ space in each node. Similar to the aggregation stage, the CUDA kernel is invoked with M blocks each thread of which computes one disaggregation followed by the integration for one partial result of the far term V_i as presented in equation (6). Finally, at the end of the MVM, the partial results from all nodes are summed together and all nodes are updated. This is accomplished through the reduce-scatter communication.

The solution of the linear equation requires communication only at two steps: (i) before starting the MVM to update the estimated values for the unknowns among the nodes; (ii) after the disaggregation stage of the MVM to update the $Z_{far}I$ results among the nodes. Due to the efficient use of the group-based and direction-based distribution schemes, the inter-node communication overhead among the nodes is reduced. This overhead is further reduced as it is performed directly among GPU memory spaces using MVAPICH2's GPU-to-GPU feature.

B.2. FPGA implementation

The SRC-7 MAPstation contains a single user FPGA, thus in this paper the FPGA parallelization is performed on a single node. However, this work can be easily extended to work on a multi-node system using MPI librabry. As in the GPU implementation, the FPGA implementation in this work also focuses on the parallelization of the processing step of Fig. 4. However, only four tasks of the processing step, namely calculating matrix components for near interactions, calculating the radiation/receive functions, calculating the V vector, and iterative linear solution, are fully pipelined on FPGA. Due to its complex recursive computation relating to the evaluations of spherical Hankel functions and Legendre polynomials, as shown in equation (5), the remaining task, which is calculating the translation matrix, is not a good candidate for the FPGA implementation and thus is handled by the CPU. Since the on-board memory (OBM) is limited to a total capacity of 64 MB, the entire workload cannot fit on a single FPGA chip. Therefore, the workload of each computational task is equally divided in a group-wise manner into sequential chunks. Each chunk is then computed using the pipelining technique. Currently, the FPGA logic resource of our SRC-7 computer limits the number of pipelines in each task to one. In spite of that, our implementation can be modified with minimal effort to work with fewer chunks and more pipelines on larger resource FPGA systems. The remaining parts of this section highlight the FPGA implementation details of each computational task.

1) Near interaction calculations

The first task is the calculation of Z_{near}, see Fig. 4, which utilizes conventional MoM. Using the group-wise partitioning technique, the computation of Z_{near}'s rows are divided into

$N_{chunknear}$ chunks (see Fig. 9) where MAX_OBM is the maximum size of a single OBM bank. At the beginning of the calculation, the geometry mesh data is transferred from the CPU memory to the OBM. Per chunk, the row and column indices of the corresponding Z_{mn} elements are transferred to OBM followed by fully pipelined computations. The evaluation of each Z_{mn}, [18], is separated into two loops running concurrently. The first loop computes the elements under the integration sign. For the second loop, we leverage our earlier work in [22] on the integration implementation. The streaming technique is exploited, as shown in yellow blocks in Fig. 9, to enable computation and data transfer overlapped. Accordingly, each result of Z_{mn} is streamed out to CPU memory immediately as soon as it is computed. The process is continued for the rest of chunks.

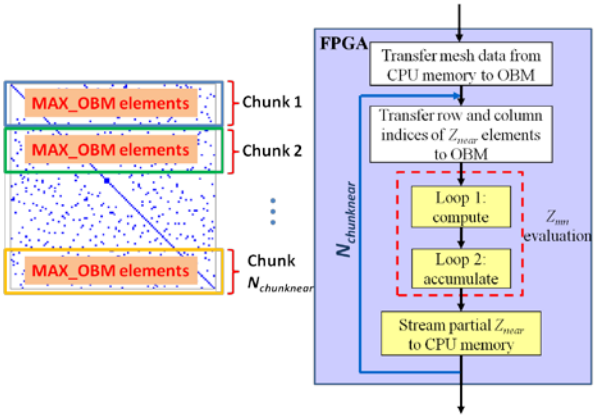

Fig. 9. Workload distribution and FPGA implementation of the Z_{near} matrix.

2) Far interaction calculations

The second task is the far interaction calculations. As mentioned before, for the FPGA implementation we only consider the radiation, T^E, and receive, R^E, functions, and their implementations are identical since they are complex conjugates of each other. For the sake of simplicity, the group-wise partitioning technique is also used to divide the radiation/receive functions to N_{chunk} chunks (see Fig. 10). The radiation/receive functions must be evaluated at all K directions. Before calculations, the group and direction data are transferred from the CPU memory to the OBM. Per chunk, the geometry mesh data is transferred to OBM before the fully pipelined computation starts. The use of streaming, as shown in yellow blocks in Fig. 10, allows the computations to continue while the data is being transferred out. The results are stored in the global-common memory (GCM) instead of the CPU memory to avoid the overhead of communication through SNAP card.

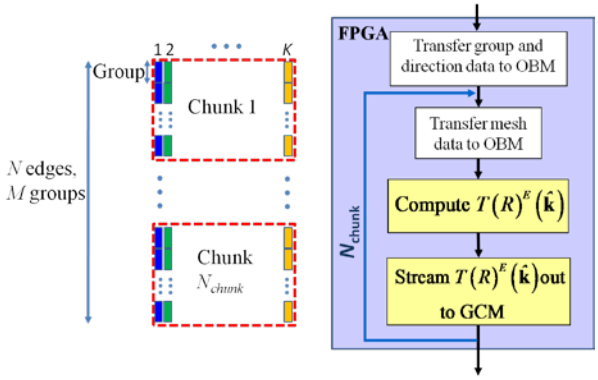

Fig. 10. Workload distribution and FPGA implementation of the radiation/receive functions.

3) V vector calculations

As in the GPU cluster implementation, the calculations of the V vector are parallelized on the hardware. The FPGA implementation details are not discussed in this paper due to its simplicity.

4) Solution of the linear equation

Similar to the GPU implementation, this section focuses on the $Z_{far}I$ matrix-vector multiplication (MVM) in BiCGSTAB algorithm. The calculation of $Z_{far}I$ comprises three stages: aggregation, translation, and disaggregation. The parallelization of far MVM follows the flowchart presented in Fig. 8, except one node is available to us in the HPRC system and is used to handle the entire computation. Thus, communication is not required at the end of the MVM. In terms of FPGA pipelining, it can be observed in equation (7) that these three stages are similar and involve multiplications and accumulations (integrations). Hence, a generic implementation, as shown in Fig. 11, can be applied to all stages. The symbols ***Input_A***, ***Input_B***, and ***Output_C*** are used to represent the input and output variables, and ***acc_length*** is the length of the accumulation for each stage (see Table I).

Before calculations, the ***Input_A*** is transferred to the OBM from the CPU memory. Using the group-wise partitioning technique, per chunk, the ***Output_C*** are then computed by multiplying each value of ***Input_A*** with its corresponding value of ***Input_B***, and accumulating on ***acc-length*** sequences. All computation loops are fully pipelined such that ***Input_B*** is streamed in from the GCM while ***Output_C*** are streamed out to the GCM.

Table I: General symbols for each stage in the far MVM.

	Aggrega-tion	***Translation***	***Disaggrega-tion***
Input_A	unknown currents	radiated fields of M groups	received fields of M groups
Input_B	radiation functions	translation matrix	receive functions
Output _C	radiated fields of M groups	received fields of M groups	$Z_{far}I$
acc-length	number of edges per group	number of elements per T_L matrix's row	K

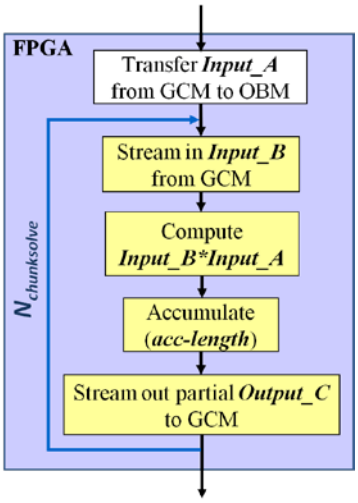

Fig. 11. Generic FPGA implementation of three stages of the far MVM.

IV. PERFORMANCE METRICS

In our performance analysis, we assume two models. The first model is the fixed-workload model (Amdahl's Law) [24] where the computational workload is fixed and equally distributed among the processing elements (PE) (i.e., computing nodes for GPU implementation and MAP cards for FPGA implementation) as the number of processing elements increases. The second model is the fixed-time model (Gustafson's Law) [24] where larger computational workloads (larger problem size) are used while maintaining the same performance as the number of processing elements is increased.

The performance of our HPC implementations is evaluated in reference to its single CPU implementation. In our analysis, we consider the computation time, T_{comp}, which is defined as the time spent on GPU or FPGA, as well as the total execution time, T_{total}, which is the sum of the computation time and the overhead, T_{comm}, which is associated with all communications between processing elements (GPUs or FPGAs) and CPUs, as given in equation (8),

$$T_{total}^{PE} = T_{comp}^{PE} + T_{comm}^{PE}. \tag{8}$$

Two metrics are investigated for performance comparisons between platforms: (i) speedup, and (ii) scalability. The speedup, S, is defined as the ratio of time required by a single CPU to carry out the total workload, $T^{CPU}(1,D)$, to the time required by multiple hardware processing elements for their associated workload, $T^{PE}(N_{PE},D_{PE})$ as in equation (9),

$$S(N_{PE}) = \frac{T^{CPU}(1,D)}{T^{PE}(N_{PE},D_{PE})}, \tag{9}$$

where D_{PE} is the workload for a single processing element, N_{PE} is the number of processing elements, D is the total workload assigned. In the fixed-workload model, the workload per element D_{PE} is adjusted with the number of processing elements, $D_{PE}=D/N_{PE}$. In the fixed-time model, the total workload D is adjusted with the number of processing elements, $D=D_{PE}*N_{PE}$.

Finally, the scalability factor, Ω, is defined as the normalized speedup of multiple processing elements in reference to a single processing element, as given by equation (10),

$$\Omega(N_{PE}) = \frac{S(N_{PE})}{S(1)}. \tag{10}$$

V. EXPERIMENTAL RESULTS

The implementation on both platforms is done using single precision. The implementation for

GPU is parallelized using up to 13 nodes, while the FPGA implementation is performed on a single node due to the available configuration in our lab. In reviewing our experimental results, first we verify the accuracy and then investigate the performance in terms of the two metrics: speedup and scalability.

A. Accuracy

We validate the accuracy of the implementation on both platforms by calculating the radar cross section (RCS) of a 5.4 λ diameter (58 K unknowns) PEC sphere illuminated by an x-polarized normally incident field. The RCS is compared with the results using Mie scattering. It can be observed from Fig. 12 that the two HPC results and the analytical solutions show a good agreement.

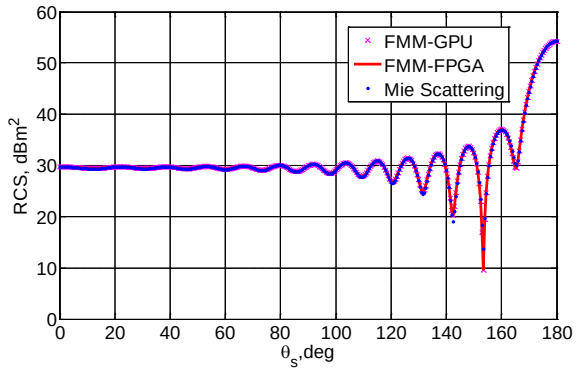

Fig. 12. RCS of a 5.4λ diameter PEC sphere.

B. Performance evaluation for GPU cluster

Two experiments are conducted with a PEC sphere following fixed-workload model and fixed-time model as discussed in section IV. In the fixed-workload model, the sphere diameter is chosen as d = 15.75 λ corresponding to 506 K unknowns. The size of the problem demands the use of at least 8 nodes to satisfy the required GPU memory. The speedup factor increases from 755 for 8 nodes to 1,152 for 13 nodes as observed in Fig. 13. Since each node processes less workload, the GPU execution time decreases as the number of nodes increases. The difference observed between the speedup of total execution time and computation time is due to the inter-node communication overhead.

In the fixed-time model, the sphere diameter for a single node is chosen as d = 7.45λ, which fully utilizes the single GPU memory with 113 K unknowns. As the number of nodes increases, the workload at each node remains constant enabling the solution for a 17.96 λ diameter sphere with 656 K unknowns for 13 nodes. We observe in Fig. 14 that the GPU implementation outperforms the CPU by achieving a speedup of 1,133 for 13 nodes.

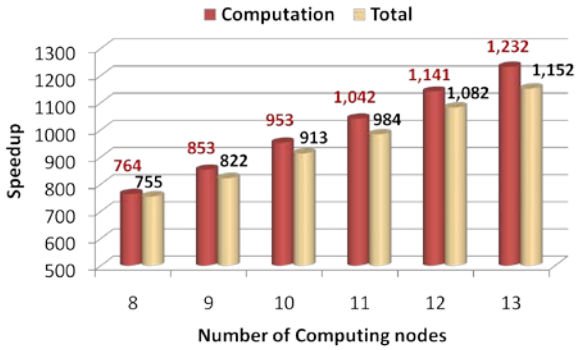

Fig. 13. Speedup of GPU cluster (Amdahl's Law, single CPU execution time ≈ 11 hours).

Finally, we compare the scalability of the GPU cluster implementation for both experiments. The scalabilities for the computation speedup and the total speedup in comparison to the linear theoretical scalability are demonstrated in Fig. 15 (fixed-workload model) and Fig. 16 (fixed-time model). It can be seen in both figures that the computation speedup scales identically to the theoretical linear expectation demonstrating our efficient hardware implementation. The total speedup scales closely to the theoretical expectation demonstrating our efficiency in reducing the inter-node communication overhead.

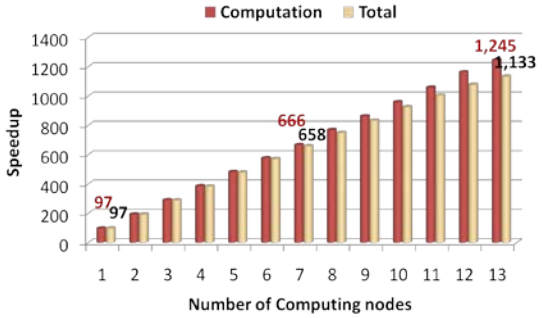

Fig. 14. Speedup of GPU cluster (Gustafson's Law, single CPU execution time ≈ 1.26 hours).

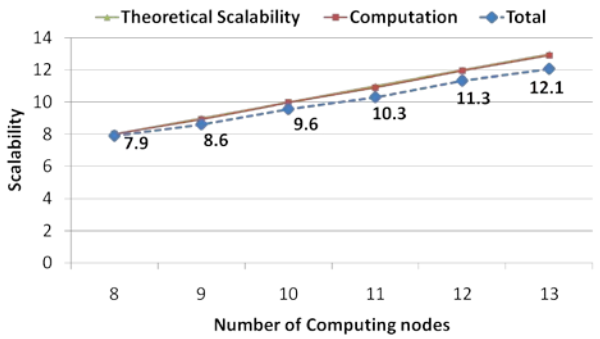

Fig. 15. Scalability of GPU implementation (fixed-workload model).

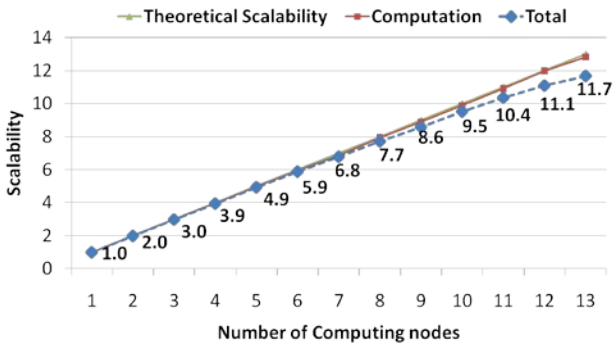

Fig. 16. Scalability of GPU implementation (fixed-time model).

C. Performance evaluation for FPGA workstation

Due to our available configuration of the SRC-7 MAPstation workstation, the experiment is carried out on a single FPGA fully utilizing its memory by choosing d = 5.4 λ (39 K unknowns). Therefore, our performance analysis is limited to the speedup of a single node, which is observed to be a factor of 2. For the same problem size, the GPU implementation achieves a total speedup factor of 77.5, outperforming the FPGA implementation. However, it should be noted that in the current FPGA implementation, due to the complex recursive nature as mentioned in section III.B.2, the translation matrix calculation task is handled on the CPU. This contributes to a decreased performance compared to that of GPU where all tasks are fully parallelized. Moreover, the FPGA system has significantly more limited resources in terms of clock speed and memory. In particular, the FPGA operates at 200 MHz whereas the GPU runs at 1.3 GHz, and the on-board memory of FPGA is limited to a total capacity of 64 MB compared with 6 GB of GPU memory. The memory limit results in the fact that each computational task can only be parallelized in a chunk-wise manner, which is part of the reason for the less impressive FPGA performance. This is despite the fact that the pipelining technique is efficiently utilized in each chunk.

VI. CONCLUSIONS

In this paper, the FMM algorithm is implemented on two HPC platforms, a 13-node GPU cluster and a single FPGA SRC workstation, for large-scale electromagnetic scattering problems. It is shown that for the same degree of accuracy, the GPU implementation outperforms the CPU implementation in terms of speedup by a factor of 1,133 for problem sizes with more than half million unknowns. Currently, the maximum problem size that can be handled by our GPU cluster implementation is limited by the GPU memory, which is 6 GB per node in our cluster. Larger problem sizes can be handled by the cluster by fully utilizing all available system resources including the CPU and GPU memories. We also observe that the GPU cluster implementation demonstrates a favorable scalability characteristic as the number of nodes increases, which proves a highly efficient parallelization scheme, which reduces the inter-node communication overhead. The paper also demonstrates a speedup factor of two for the FPGA implementation. Although, the comparison of performance in terms of speedup reveals that the GPU implementation surpasses the FPGA implementation, it should be noted that the FPGA system has significantly more limited resources than GPU in terms of frequency (200 MHz versus 1.3 GHz) and on-board memory (64 MB versus 6 GB). In the near future, when FPGA computers are equipped with larger resources and operate at higher frequencies, the authors believe a comparable performance with GPU can be achieved.

ACKNOWLEDGMENT

This material is based upon work supported by, or in part by, the U.S. Army Research Laboratory and the U.S. Army Research Office under contract/grant number W911NF-09-1-0123.

REFERENCES

[1] E. Bleszynski, M. Bleszynski, and T. Jaroszewicz, "AIM: Adaptive integral method for solving large-scale electromagnetic scattering and radiation problems," *Radio Science*, vol. 31, no. 5, pp. 1225-1251, 1996.

[2] F. Canning, "The impedance matrix localization (IML) method for moment-method calculations," *IEEE Antennas Propagat. Mag.*, vol. 32, no. 5, pp. 18-30, 1990.

[3] R. Coifman, V. Rokhlin, and S. Wandzura, "The fast multipole method for the wave equation: a pedestrian prescription," *IEEE Antennas Propagat. Mag.*, vol. 35, no. 3, pp. 7-12, June 1993.

[4] J. Song and W. Chew, "Multilevel fast multipole algorithm for solving combined field integral equations of electromagnetic scattering," *Microw. Opt. Tech. Lett*., vol. 10, pp. 14-19, Sep. 1995.

[5] C. Waltz, K. Sertel, M. Carr, B. Usner, and J. Volakis, "Massively parallel fast multipole method solutions of large electromagnetic scattering problems," *IEEE Trans. Antennas Propag.*, vol. 55, no. 6, pp. 1810-1816, 2007.

[6] S. Velamparambil, J. Schutt-Aine, J. Nickel, J. Song, and W. Chew, "Solving large scale electromagnetic problems using a Linux cluster and parallel MLFMA," in *IEEE Antennas Propag. Soc. Int. Symp.*, vol. 1, pp. 636-639, 11-16 July 1999.

[7] S. Velamparambil and W. Chew, "Analysis and performance of a distributed memory multilevel fast multipole algorithm," *IEEE Trans. Antennas Propag.*, vol. 53, no. 8, pp. 2719-2727, August 2005.

[8] E.-L. Lu and D. Okunbor, "A massively parallel fast multipole algorithm in three dimensions," in *Proc. IEEE High Perform. Distrib. Comput. Int. Symp.*, pp. 40-48, August 1996.

[9] E.-L. Lu and D. Okunbor, "Parallel implementation of 3d FMA using MPI," in *Proc. MPI Developer's Conf.*, pp. 119-124, July 1996.

[10] S. Velamparambil, W. Chew, and M. Hastriter, "Scalable electromagnetic scattering computations," in *IEEE Antennas Propag. Soc. Int. Symp.*, vol. 3, pp. 176-179, 2002.

[11] G. Sylvand, "Performance of a parallel implementation of the FMM for electromagnetics applications," *Int. J. Numer. Meth. Fluids*, vol. 43, no. 8, pp. 865-879, Nov. 2003.

[12] O. Ergul and L. Gurel, "Efficient parallelization of the multilevel fast multipole algorithm for the solution of large-scale scattering problems," *IEEE Trans. Antennas Propag.*, vol. 56, no. 8, pp. 2335-2345, August 2008.

[13] M. López-Portugués, J. López-Fernández, J. Ranilla, R. Ayestarán, and F. Las-Heras, "Parallelization of the FMM on distributed-memory GPGPU systems for acoustic-scattering prediction," *J. Supercomput.*, vol. 64, no. 1, pp. 17-27, April 2013.

[14] M. López-Portugués, J. López-Fernández, J. Menéndez-Canal, A. Rodríguez-Campa, and J. Ranilla, "Acoustic scattering solver based on single level FMM for multi-GPU systems," *J. Parallel Distrib. Comput.*, vol. 72, no. 9, pp. 1057-1064, Sep. 2012.

[15] M. Cwikla, J. Aronsson, and V. Okhmatovski, "Low-frequency MLFMA on graphics processors," *IEEE Antennas Wireless Propag. Lett.*, vol. 9, pp. 8-11, 2010.

[16] Q. Nguyen, V. Dang, O. Kilic, and E. El-Araby, "Parallelizing fast multipole method for large-scale electromagnetic problems using GPU clusters," *IEEE Antennas Wireless Propag. Lett.*, vol. 12, pp. 868-871, July 2013.

[17] V. Dang, Q. Nguyen, O. Kilic, and E. El-Araby, "Fast multipole method for large-scale electromagnetic scattering problems using high performance computers," in *The 29th International Review of Progress in Applied Computational Electromagnetics (ACES 2013)*, Monterey, CA, USA, 24-28 March 2013.

[18] S. Rao, D. Wilton, and A. Glisson, "Electromagnetic scattering by surfaces of arbitrary shape," *IEEE Trans. Antennas Propag.*, vol. 30, no. 3, pp. 409-418, May 1982.

[19] E. El-Araby, O. Kilic, and V. Dang, "Exploiting FPGAs and GPUs for electromagnetics applications: interferometric imaging in random media case study," *The Applied Computational Electromagnetics Society (ACES) Journal*, vol. 27, no. 2, Feb. 2012.

[20] F. Darema, "The SPMD model: past, present and future," in *Proc. 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing interface,* Lecture Notes In Computer Science, vol. 2131, pp. 1, Sep. 2001.

[21] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, Philadelphia, PA: SIAM, 1994.

[22] O. Kilic, E. El-Araby, Q. Nguyen, and V. Dang, "Bio-inspired optimization for electromagnetic structure design using full-wave techniques on GPUs," *Int. J. Numer. Model.*, vol. 26, no. 6, pp. 649-669, November/December 2013.

[23] NVIDIA Corporation, *CUDA Toolkit 4.2 CUBLAS Library,* Santa Clara, CA, Feb. 2012.

[24] K. Hwang and Z. Xu, *Scalable Parallel Computing: Technology, Architecture, Programming*, New York, NY: McGrawHill, 1998.

Vinh Dang *received hi*s B.Sc. (2003) and M. Eng. (2006) degrees in Electrical Engineering from the Posts and Telecommunications Institute of Technology and the University of Technology, in Vietnam, respectively. Prior to 2010, he was a Lecturer at the School of Electrical Engineering, International University. He is currently a PhD. candidate and a Graduate Research Assistant in the Department of Electrical Engineering and Computer Science, the Catholic University of America (CUA). His research interests include high performance computing, numerical electromagnetics with applications to radiation, scattering and remote sensing.

Quang Nguyen received his B.Sc. (2009) and M.Eng. (2011) degrees in Electrical Engineering from the International University, Vietnam, and the Catholic University of America, USA, respectively. He is currently a PhD. student and a Graduate Research Assistant in the Department of Electrical Engineering and Computer Science, Catholic University of America. His research interests include bio-inspired optimization methods, numerical electromagnetics with applications to radiation, scattering and remote sensing.

Dr. Ozlem Kilic is an Associate Professor in the Department of Electrical Engineering and Computer Science of the Catholic University of America. Prior to joining CUA, she was an Electronics Engineer at the U.S. Army Research Laboratory, Adelphi MD. Dr. Kilic has over five years of industry experience at COMSAT Laboratories as a Senior Engineer and Program Manager with specialization in satellite, link modeling and analysis. Her research interests include numerical electromagnetics, antennas, wave propagation, satellite communications systems, and microwave remote sensing. She is an Associate Editor of IEEE Antennas and Propagation Magazine and Applied Computational Electromagnetics Society Journal. She serves as Member at Large for USNC-URSI.