CHAPTER

19 Fast Electromagnetic Integral Equation Solvers on Graphics Processing Units

Shaojing Li, Ruinan Chang, and Vitaliy Lomakin

In this chapter, we present volumetric and surface integral equation (IE) electromagnetic solvers implemented on graphics processing units (GPUs). The IEs are discretized into a set of algebraic equations that are solved iteratively. The matrix-vector product in the iterative solution is evaluated via the nonuniform grid interpolation method (NGIM), which allows evaluating the electromagnetic field from N sources at N observers in $O(N)$ or $O(N\log N)$ operations. We describe how the NGIM and IE solvers can be implemented on GPUs. Specifically, we present a "one-thread-per-observer" "on-fly" approach allowing for an efficient utilization of the GPU architectures, including shared memory, coalesced global memory access, and massive parallelization. We demonstrate that the presented approaches allow achieving significant GPU speed-up rates for problems of a large computational size. The presented methods and approaches can be used not only for electromagnetic problems but also for many other problem types.

19.1 PROBLEM STATEMENT AND BACKGROUND

Computational methods for the electromagnetic analysis of large-scale and complex systems are essential for our ability to design and characterize practical devices and systems. Integral equation (IE) solvers are a powerful tool for the electromagnetic analysis. IEs allow focusing the solution only on the structure of interest (without a need to discretize the surrounding volumes), incorporate exact radiation conditions, and do not lead to numerical dispersion. These properties make IEs attractive for simulating complex and large-scale structures.

Several integral equation formulations can be used. Volumetric integral equations represent the structure's volume in terms of equivalent unknown currents or fields. For example, assuming that the structure under consideration comprises only bulk dielectric (nonmagnetic) materials, a volume integral equation can be written in the following form

$$\frac{\mathbf{D}(\mathbf{r})}{\varepsilon_r} - \nabla \iiint\limits_V \frac{\mathbf{e}^{-jk_0|\mathbf{r}-\mathbf{r}'|}}{|\mathbf{r}-\mathbf{r}'|}\nabla' \bullet (k_e\mathbf{D}(\mathbf{r}'))dV' - k_0^2 \iiint\limits_V \frac{\mathbf{e}^{-jk_0|\mathbf{r}-\mathbf{r}'|}}{|r-r'|}k_e\mathbf{D}(\mathbf{r}')dV' = 4\pi\varepsilon_0\mathbf{E}^{inc}(\mathbf{r});\ \mathbf{r}\in V, \tag{1}$$

where k_0 is the free space wavenumber, $\mathbf{E}^{inc}$ is the incident field, $\mathbf{D}$ is the electric flux density, $|\mathbf{r}-\mathbf{r}'|$ is the distance between the source point $\mathbf{r}'$ and observation point $\mathbf{r}$, ε_r is the relative permittivity of the material under consideration, and $k_e = 1-1/\varepsilon_r$ represents the contrast between the free space permittivity and the material permittivity.

Surface integral equations define equivalent unknown currents on the structure's surface only. For example, for a surface problem comprising a structure made of a perfect electric conductor residing in free space, an electric field IE can be written for an unknown surface current $\mathbf{J}_s$ distributed on the surface S of the structure in the following mixed potential form

$$\widehat{\mathbf{n}} \times \left(j\omega\mu \iint\limits_S \frac{e^{-jk_0|\mathbf{r}-\mathbf{r}'|}}{|\mathbf{r}-\mathbf{r}'|}\mathbf{J}_s\left(\mathbf{r}'\right)ds' - \nabla \iint\limits_S \frac{e^{-jk_0|\mathbf{r}-\mathbf{r}'|}}{j\omega\varepsilon|\mathbf{r}-\mathbf{r}'|}\nabla'\cdot\mathbf{J}_s\left(\mathbf{r}'\right)ds' \right) = 4\pi\widehat{\mathbf{n}}\times\mathbf{E}^i(\mathbf{r});\ \mathbf{r}\in S. \quad (2)$$

To solve Eqs. (1) and (2), the structure under consideration is meshed over volume or surface elements, which typically are tetrahedrons volumes or triangles for surfaces. The unknowns are expanded via a number N of basis function [1]. These expansions are substituted in the IE and are subsequently tested (i.e., integrated) with testing functions (that may be chosen the same as the basis functions). The IE is then transformed into a set of algebraic equations

$$\sum_{n=1}^{N_s} Z_{mn}\, j_n = V_m, \quad (3)$$

where Z_{mn} are elements of the impedance matrix that represent a discretized form of the integro-differential operators in Eqs. (1) and (2) and V_m is the tested (known) incident field. For example, for the surface integral equation, Z_{mn} and V_m are given by

$$Z_{mn} = k_0 \iint\limits_{Sm} d\mathbf{r}\mathbf{f}_m(\mathbf{r}) \bullet \iint\limits_{S_n} \frac{\mathbf{f}_n(\mathbf{r}')e^{-jk_0|\mathbf{r}-\mathbf{r}'|}}{|\mathbf{r}-\mathbf{r}'|}d\mathbf{r}' - \frac{1}{k_0}\iint\limits_{Sm} d\mathbf{r}\nabla\bullet\mathbf{f}_m(\mathbf{r}) \iint\limits_{S_n} \frac{\nabla'\bullet\mathbf{f}_n(\mathbf{r}')e^{-jk_0|\mathbf{r}-\mathbf{r}'|}}{|\mathbf{r}-\mathbf{r}'|}d\mathbf{r}',$$

$$V_m = j\frac{4\pi}{\eta}\iint\limits_{Sm} \mathbf{f}_m(\mathbf{r})\bullet\mathbf{E}^{inc}d\mathbf{r}, \quad (4)$$

where $\mathbf{f}_m$ are vector basis and testing functions. Similar expressions for the impedance matrix can be given also for the volume integral equation (Eq. (1)). For nonoverlapping basis and testing functions, the integrals in Eq. (4) are computed using quadrature rules of a certain order, whereas for overlapping basis and testing functions, the singular behavior of the integral kernel is accounted for via an analytical integration.

Solving the system (Eqs. (3)–(4)) iteratively requires evaluating the Helmholtz type field potential via the following discrete transformation

$$u(\mathbf{r}_m) = \sum_{n=1;n\neq m}^{N} \frac{e^{-jk_0|\mathbf{r}_m-\mathbf{r}_n|}}{|\mathbf{r}_m-\mathbf{r}_n|}Q_n, \quad m=1,2,\ldots,N \quad (5)$$

Here, the potential $u(\mathbf{r}_m)$ at the observation locations $\mathbf{r}_m$ is evaluated by a discrete convolution of the Green's function $G(\mathbf{r}_m,\mathbf{r}_n) = \exp\left(-jk_0\,|\mathbf{r}_m-\mathbf{r}_n|\right)/|\mathbf{r}_m-\mathbf{r}_n|$ and the sources Q_n are colocated with the observers. In the framework of the above IE, the potential $u(\mathbf{r}_m)$ represents each one of the three components of the vector potential and the scalar potential (i.e., four components total), which result from the sources Q_n representing the three components of the current and the charge together with proper quadrature weights. The total number of sources is N, and it is proportional to the number of discretization elements. In the low-frequency regime, the computational domain is small in terms of the wavelength ($D \ll \lambda$, or $D \sim \lambda$) and the source density is prescribed by the particular problem (e.g., by the geometrical). In the special case of $k_0 = f = 0$, the potential satisfies the Poisson equation, and the problem is considered to be static. In the high-frequency regime, the computational domain is large in terms of the wavelength ($D \gg \lambda$), and the source density is determined by the wavelength according to the Nyquist criterion. In the mixed-frequency regime, the computational domain is large in terms of the wavelength, and it has dense source constellations, typically representing sharp geometrical features.

Rapidly evaluating the potential in the form of Eq. (5) is important for solving electromagnetic IEs, but it also is applied in many other areas, including micromagnetics, acoustics, and elastodynamics. Moreover, the methods presented here can be directly applied, without any modification, to other potential types with many additional applications (e.g., London or Van-der-Waals potentials).

Accelerating the direct evaluation of Eq. (5) through parallelization on either CPU clusters or GPU-CPU heterogeneous hardware platforms has been investigated in various research fields, such as a standard "N-body" problem in astrophysics and molecular dynamics [2, 3] or a matrix-vector multiplication problem in electromagnetics [4]. However, the $O(N^2)$ computational complexity essentially prevents the simulation of a reasonable large problem on desktop machines or even midsize clusters. Fast methods, such as Fast Multipole Methods (FMMs) [5–14], Fast-Fourier-Transforms (FFT)-based methods [15–18], H- or H2-matrices-based methods [19–22], or interpolation-based methods [23–28], reduce the computational complexity to $O(N)$ or $N \log(N)$. However, porting these methods to GPUs is not a trivial task and a relatively small number of works have reported on their implementations [29–31].

The goal of this work is to introduce a method that evaluates the potential in Eq. (5) efficiently using CUDA on graphics processor units (GPUs) with computational complexity (i.e., the computational time of number of operation) of $O(N)$ or $O(N \log N)$. This method is further coupled with an integral equation solver to rapidly solve the EFIE equation.

19.2 ALGORITHMS INTRODUCTION

This section describes a highly efficient GPU implementation of a modification of the nonuniform grid interpolation method (NGIM) [32], for the fast evaluation of the potential $u(\mathbf{r}_m)$ in Eq. (5). The NGIM exploits the fact that the field potential far from a source distribution is a function with a known asymptotic behavior. This behavior allows smoothing the fast spatial variations of the potential, computing it on a sparse grid, and interpolating to the required observation points. The algorithm is implemented using a hierarchical domain decomposition method in which the domain is subdivided via an octal tree into a hierarchy of levels comprising subdomains of different sizes. Near- and far-field subdomains are identified, and the interpolation procedures are implemented for the sufficiently separated subdomains. This algorithm achieves the computational cost of $O(N)$ in the low-frequency regime, $O(N \log N)$ in the high-frequency regime, and between these costs in the mixed-frequency regime.

Similar to the FMMs, NGIMs can handle nonuniform geometries and can have the same asymptotic cost for volumetric and surface problems. The NGIM differs from the FMMs in that it relies on direct spatial interpolations, which are natural operations for GPUs so can be better adapted to GPU architectures and achieve higher computational throughput. Moreover, the same NGIM can be applied to static ($k_0 = 0$) and dynamic ($k_0 \neq 0$) problems, as well as to problems with other kernels without major changes in its structure and mathematical operations, as opposed to FMMs. In addition, NGIM has a simple structure and does not require any special function evaluations, which facilitates its implementations on GPU systems, as will be shown in the section titled “GPU Implementations.”

19.3 ALGORITHM DESCRIPTION

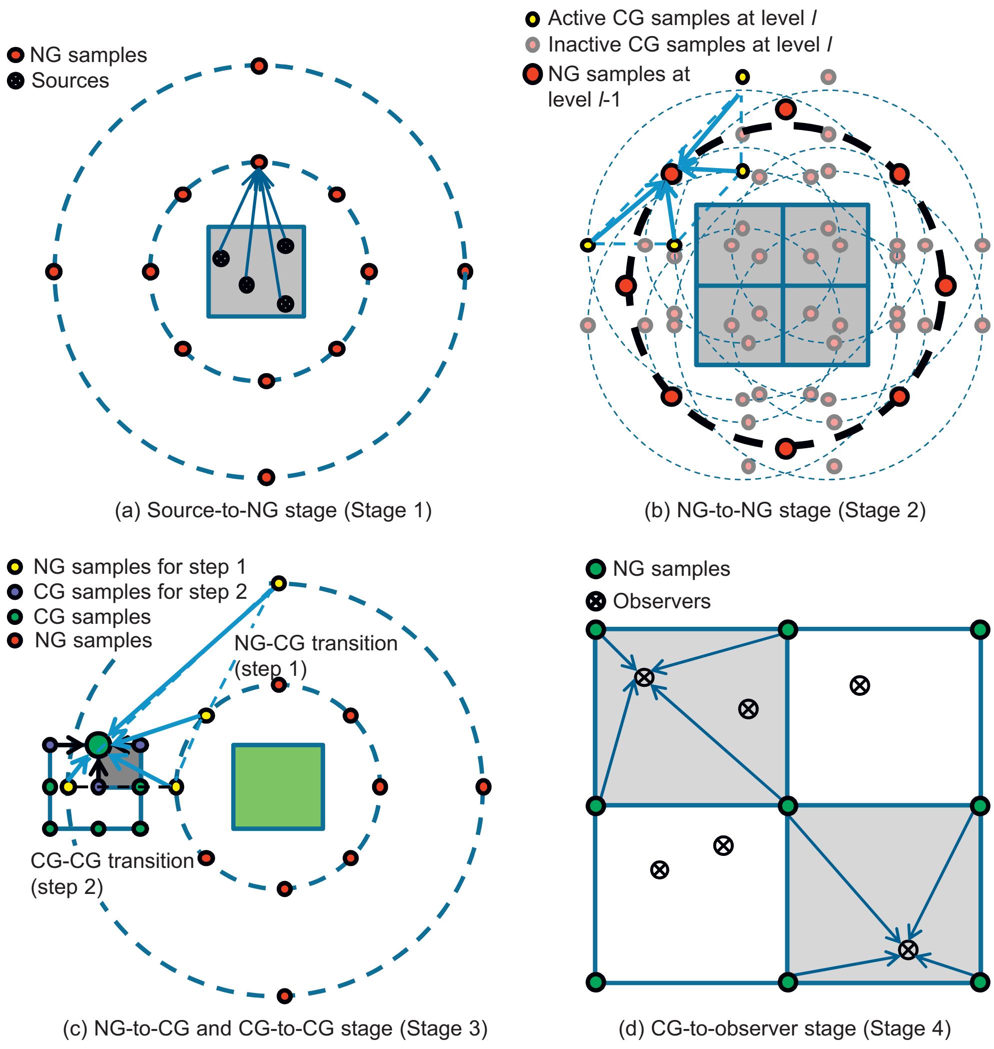
In this section we present a description of the NGIM and its implementation scheme. The presentation is given in a summary form. A more complete description of the algorithm can be found in [23–27] and [32].

19.3.1 Grid Construction

The NGIM divides the computational domain into a hierarchy of boxes containing sources and observers. At any subdivision level, each box is treated as a “parent” box and recursively is divided into eight “child” boxes at lower level. This process stops until boxes at the finest level contain less than a prescribed number of sources. Then, for a certain observation point, near-field and far-field boxes are identified for a distance larger or smaller than a predefined value (e.g., for distances twice the box diameter). The field potentials contributed by sources in the near-field boxes are evaluated one by one directly via superposition. The fields due to the sources in far-field boxes are aggregated to the box center and had their field interpolated on observer through several grids of sample points across different levels. This procedure, illustrated in Figure 19.1, is similar to other multilevel algorithms.

The field outside a group of sources is amplitude- and phase-compensated with respect to the common distance from the group center. The resulting slowly varying field can then be sampled at a sparse nonuniform grid (NG) and calculated at all desirable observation locations via inexpensive local (e.g., Lagrange) interpolation. The density and specific position of NG samples points are determined mathematically [26, 32] in preprocessing stage and remain the same in the entire algorithm or IE solver. In a low-frequency regime, a total of $O(N)$ NG samples are required to sample the field across all levels of the boxes, but in a high-frequency regime, this number is $O(N \log N)$. Generally speaking, in either domain, the higher the accuracy requirement, the denser the NG grid.

The frequency of the computations also affects another aspect of the algorithm. If the frequency of the problem is low, the field transition from NG samples to the final observation points are not done directly, but via another set of intermediate Cartesian Grids (CGs). The CGs can save the computational cost because in low-frequency applications, a small number of sampling points are sufficient to sample a slow varying field generated by sources outside a computational domain. In a high-frequency regime, the density of CGs should be the same as the density of the observers so that using CGs would not lead to any computational savings; therefore, CGs are not used in the high-frequency regime. In the mixed-frequency regime, CGs are built for lower levels with sufficiently small boxes (low-frequency levels)



NG samples
Sources
(a) Source-to-NG stage (Stage 1)
Active CG samples at level *l*
Inactive CG samples at level *l*
NG samples at level *l*-1
(b) NG-to-NG stage (Stage 2)
NG samples for step 1
CG samples for step 2
CG samples
NG samples
NG-CG transition (step 1)
CG-CG transition (step 2)
(c) NG-to-CG and CG-to-CG stage (Stage 3)
NG samples
Observers
(d) CG-to-observer stage (Stage 4)

FIGURE 19.1

The 2-D illustration of the far-field stages of NGIM.

and are omitted for higher levels where the field between boxes varies too rapid even when they are well separated from each other (high-frequency levels).

19.3.2 Algorithm

From the preceding section, the fields at observers are obtained via a sequence of interpolations. In the low-frequency regime, for example, the field at observers is interpolated via CGs, which is in turn obtained from NGs of certain boxes and CGs of their parent boxes. The fields at NGs are obtained, via interpolation, from their child boxes, except on the lowest level they are calculated directly from the sources. This process, which can be described as a stage-by-stage procedure, includes four stages:

- Stage 0 (near-field evaluation): All near-field interactions between the sources in the near-field boxes at the finest level are computed via direct superposition. This step is completely independent of the rest of the algorithm and can be separately implemented and executed in parallel with all other stages on multi-GPU systems.
- Stage 1 (finest level NG construction): The computation starts with directly computing the field values on an NGs serving as sample points for all source boxes in the finest level.
- Stage 2 (aggregation of NGs/upward pass): The field values at the NGs of the boxes at coarser levels are computed by aggregation from their child boxes on finer levels. Such aggregation involves local interpolation and common distance compensation in the amplitude and phases between the corresponding NGs.
- Stage 3 (NG to CG transitions and CG decomposition/downward pass): Local CGs are constructed as an intermediate step to do the interpolation in the desirable observation boxes. Field values at CGs of an observer box on a specific level come from two contributions. The first contribution is from same level interaction-list boxes, which satisfy the following condition. The interaction-list boxes of an observer box have their parent boxes as neighbors of the observer box, except those have already been taken into considerations in the near-field stage (corresponds to influences of the source of "medium distance"). The second contribution of field at CGs comes from the parent box. This contribution is valid only for boxes below the interface level. CGs of an observer box obtain field value from these two sources through interpolations, from NG samples in the former case, and from CG samples in the latter case.
- Stage 4 (CG to observation point): The field values at observation points are obtained by local interpolations from the CGs on the finest level of the domain subdivision. The whole process has a computational cost of $O(N)$. The use of local interpolation guarantees the automatic adaptivity to geometrical features because the NGs and CGs are built and processed only around locations where sources and observers are present.

One thing worth special attention is that, as discussed in the previous paragraph, there is no CG constructed for computations in a high-frequency regime, so the CG-CG interpolations do not exist, and the whole downward pass disappears. In this regime, fields are directly interpolated from the NGs at each level to observers. In mixed-frequency regime, the downward pass exists partially, for "low-frequency" levels only; thus, the downward pass possesses a hybrid scheme of the "direct NG-observer interpolation" and "NG-CG-observer interpolation."

19.4 GPU IMPLEMENTATIONS

In the implementation of the NGIM, the unique programming mechanisms and hardware arrangement of GPUs are critical to the efficiency of the algorithm. These include the coalesced accessing of the global memory, the utilization of shared memory, and the minimal atomic parallelization unit, "*warp.*" All these concepts and mechanisms have been discussed extensively in NVIDIA [33] as well as many other works related to scientific computing on GPUs [34]. These concepts have significant effects on the time and memory consumption of the NGIM on GPUs and result in a number of important modifications in the data structure of the code as compared with the CPU implementations. The implementation of the NGIM on GPUs follows the same ”stage-by-stage” protocol as that on CPUs, yet extensive changes are made to parallelize the operations and utilize tools provided by CUDA.

19.4.1 Preprocessing and Initialization Stages

In the preprocessing stage, all vectors, matrices, and other data structures used by the NGIM are initiated. The initialization includes memory allocation in global memory of GPU and copying coordinates of sources and observers to the allocated matrices as well as reshaping and copying auxiliary matrices — for example, indexing and storing the near-field boxes and interaction-list boxes for each observer box. This task is done only once in standard IE application because the geometry of the computational domain usually remains the same during the whole problem-solving process.

In addition to the memory transfer operations, one crucial task done in the preprocessing stage specifically for GPU is rearranging the source storage so that sources belong to the same box are situated contiguously in the memory. This is critical for the GPU to adopt “coalesced accessing” to accelerate the memory handling, which will be described in detail later in this chapter. Fortunately, the hierarchical structure of boxes still allows this to be done once for all levels as sources belong to the same box on lower levels always belong to the same box on higher levels, too.

In the GPU implementation of NGIM, grid tabulation is significantly shortened compared with its CPU sequential counterparts. The position of NG or CG samples and their mutual interpolation coefficient are computed *on-fly* where needed. This *on-fly* approach reduces the memory consumption and the total memory access time. As a result, the preprocessing time of the GPU code is reduced, making the code more efficient and practical. In addition, the overall code speed may increase in spite of a larger number of operations. It is noted that using a similar approach for FMM type methods is also possible, but it may be somewhat less efficient owing to more complex operations involved (e.g., these methods require computing special functions).

19.4.2 Near-Field Computation

In this stage, the fields at the observers are evaluated directly via Eq. (5) by adding up the field contributions from sources belonging to the level L boxes in the near-field region of the observer’s box. Methods to parallelize this stage also apply to direct evaluations of the classical “n-body” problems [2]. Because the computational domain has already been divided into boxes with sources and observers arranged box by box, the traversing within the list of observers and sources are done in a box-by-box manner.

Mathematically, the near-field computation is a sparse matrix-vector multiplication. However, for large problem sizes the memory limitations may be an obstacle. Therefore, instead of defining sparse matrix-vector products, we compute all the fields on-fly by direct superposition. Figure 19.2 shows the flowchart diagram of CPU and GPU implementation of NGIM, respectively, and Figure 19.3 shows the thread arrangement and assignment during the memory loading and the computation of the same stage. The key points are summarized here.

(1) We adopted “one-thread-per-observer” type of parallelization in which one thread is responsible for calculating the field value at one observer.

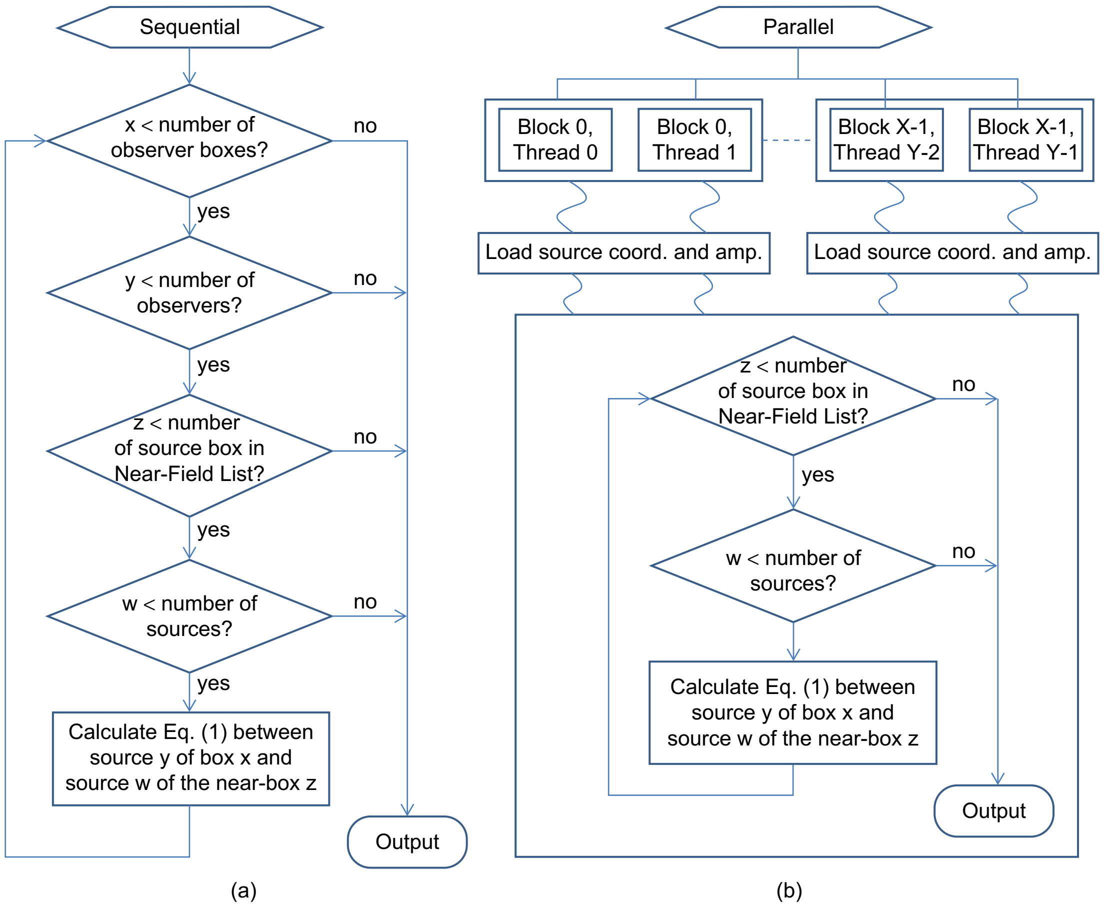

Sequential
x < number of observer boxes?
no
yes
y < number of observers?
no
yes
z < number of source box in Near-Field List?
no
yes
w < number of sources?
no
yes
Calculate Eq. (1) between source y of box x and source w of the near-box z
Output
(a)
Parallel
Block 0, Thread 0
Block 0, Thread 1
Block X-1, Thread Y-2
Block X-1, Thread Y-1
Load source coord. and amp.
Load source coord. and amp.
z < number of source box in Near-Field List?
no
yes
w < number of sources?
no
Calculate Eq. (1) between source y of box x and source w of the near-box z
Output
(b)

FIGURE 19.2

The flow chart of CPU and GPU NGIM. (a) The sequential version of the near-field stage of the NGIM involves a four-level loop that takes into account that each source-observer pair satisfies the Near-Field criterion. (b) In the corresponding parallel version of the near-field stage of the NGIM, two levels of loop are spread onto parallel stream processors of GPU. X and Y are the number of observer boxes and number of observers in each box, respectively. Coalesced memory loading is utilized and shown in detail in Figure 19.3.

(2) The number of threads per block can be chosen by the user or determined by properties of the hardware, number of unknowns of the problem, and the source/observer distribution automatically. However, only threads for observers of the same box are bundled to form a block. One or several thread blocks may be launched to handle a certain box when the number of observers within the box exceeds the user-defined number or the hardware limitation. To achieve a better performance, the number of threads per block should be greater than 32 because this is the size of a warp.

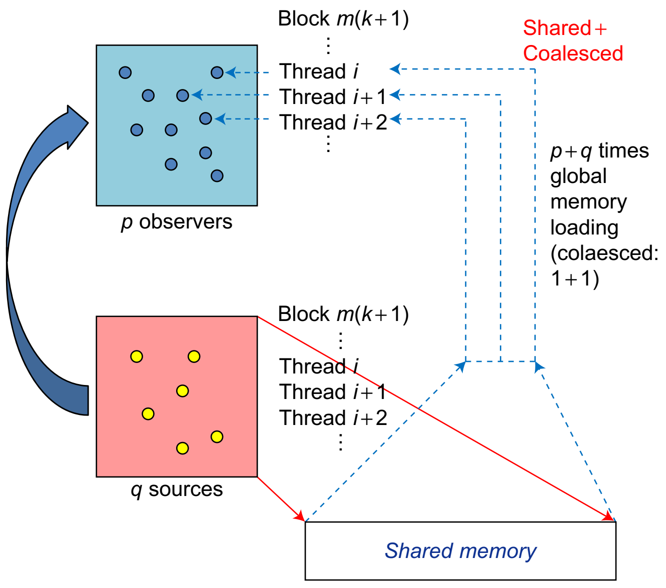


FIGURE 19.3

Memory access pattern of threads within the same block. Coalesced global memory is utilized to accelerate the memory loading.

(3) The exact same source information (coordinates and amplitudes) is required to compute the near field of the observers in the same box, so threads assigned for these observers first cooperatively load the source information they all need before they synchronize their progress and distribute themselves into the task they are assigned. Because the information of any sources in the same box is located contiguously in the global memory, coalesced loading can always be utilized.

(4) One-, two-, or three-dimensional grids of blocks can be used to handle all nonempty boxes. In the current implementation, we use two-dimensional grids of blocks because they allow any practical number of blocks to be launched for each individual kernel.

(5) Some intrinsic mathematical functions are used to accelerate the computations. These functions include single precision versions of sin and cos functions, used to evaluate the complex exponential in the Greens' function in Eq. (5). Other instruction level techniques include replacing the integer division and modulo operations with bitwise shifting and AND operations when the dividend is power of 2 [33].

(6) The data type "float2" is used in the near-field and following stages because it can be directly mapped to the "complex" data type we use in our CPU code written in Fortran. In the CUDA compiler, the operator overload mechanism is introduced so that the operations on float2 can be defined exactly as those for complex numbers.

Table 19.1 shows the computational time of the near-field stage on CPU and GPU. The CPU timing results were obtained on a single core of an Intel Xeon X5248 with a 3.2-GHz CPU using Intel Fortran

Table 19.1 The Computational Times and Speed-Up Ratios of the Near-Field Stage on CPU (Xeon X5248) and GPU (GeForce GTX 480). N_p is the Average Number of Sources per Box on Level L. The Relation Between N_p and N is $N_p = N/8^L$

N_p	L	CPU*	GPU	Ratio
16	3	2.11e–1	1.02e–3	206.9
32	3	8.63e–1	2.22e–3	388.7
64	3	3.42e0	5.90e–3	579.7
N_p	L	CPU	GPU	Ratio
16	4	1.97e0	7.49e–3	263.0
32	4	7.84e0	1.74e–2	450.6
64	4	3.13e1	4.84e–2	646.7
N_p	L	CPU	GPU	Ratio
16	5	1.85e1	7.76e–2	238.4
32	5	6.75e1	1.45e–1	465.5
64	5	2.66e2	5.30e–1	501.9
N_p	L	CPU	GPU	Ratio
16	6	1.43e2	6.38e–1	224.1
32	6	5.66e2	1.19e0	475.6
64	6	2.22e3	4.37e0	508.0

* *All timing results shown in this table are in seconds.*

Compiler v10 with –O3 optimization (there was around 20-fold speed-up of a -O3 optimized CPU code over a nonoptimized one). At the GPU end, an NVIDIA GTX480 running at 700 MHz with 1.5 GB of memory was used. The GPU implementation was written and compiled using CUDA Toolkit v3.0 from NVIDIA. Both CPU and GPU versions of the code used the *on-fly* approach, and the positions of source and observer were random with a uniform probability distribution function.

It is evident that the speed-up ratios of the GPU code compared with that of the CPU one are very high, varying between 200 and 650. The speed-ups are higher for larger N_p, that is, for a larger number of sources and observers per box, when the massive parallelization is fully exploited. Taking into account the fact that the number of the GPU cores in the considered case is 480 and they are run at the clock rate around 4.5 times lower than that of the CPU, achieving the acceleration rates higher than 600 is impressive. Such high rates are obtained because of not only massive floating-point computing power and memory bandwidth of GPUs but also the memory sharing between threads and the coalesced memory access of the global memory.

A comment should be made on the timing results shown Table 19.1. For a fixed number of levels, the complexity of the near-field calculation stage scales approximately as $O(N^2)$ with increasing N as the number of near-field evaluations is proportional to N_p^2. The complexity of $O(N)$ of the near-field stage is achieved because the number of levels L increases with an increase of N. Indeed, the computational time behaves as $O(N)$ when the number of level L is properly chosen, balancing the near- and far-field computation time.

19.4.3 Outward Computation from Sources to NG Samples (Stage 1)

The NG construction stage computes the field values at NGs, which is the first step of the upward pass of the algorithm. The core operations in this stage are (a) the construction of NGs of each nonempty box at the finest level L and (b) the direct calculation of the field values at these NGs via Eq. (5).

The sequential version of code consists of two nested loops to deal with all pairs of sources and NG samples for individual boxes and another loop to account for all boxes at level L (Eq. (5)). Because the relative positions of NG samples and sources do not change during the whole process of IE solver, their interaction coefficients can be calculated beforehand in the preprocessing stage and stored for later use (i.e., the interacting matrix filling). However, this task is only done in our CPU version of code. For GPU, no coefficient matrices are used for the reasons mentioned in previous sections.

Before the fields on NG samples can be calculated, the positions of the NG samples have to be calculated first. The construction process follows the "one-thread-per-observer" approach described in the section titled "Algorithms Introduction." However, here the "observers" are in their broader definition, referring to NG samples. One or several blocks of threads are allocated for each observation box. For calculation in the low-frequency regime without very high accuracy, the number of NG samples per box is not large, and one block of threads would be enough to achieve the maximal parallel efficiency. In the high- and mixed-frequency regimes, however, the boxes at high-frequency levels can have a large number of NG samples. This requires assigning multiple blocks for each box. Regardless of the number of blocks assigned for each box, the "coalesced" memory-reading technique is always employed to accelerate the loading of source coordinates and amplitudes.

The computational times of Stage 1 are presented in Table 19.2 (these results are frequency regime independent for the same N, L, and the number of NG samples per smallest box). It is evident that the speed-up ratio increases significantly with an increase of the number of sources per box.

It should be mentioned that generally, the computation time of Stage 1 is about 1–5% of the total time. Therefore, the influence of this stage is relatively insignificant provided other stages are implemented efficiently.

19.4.4 NG Upward Aggregation (Stage 2)

In this stage, the field values at the NG samples of the parent boxes at levels from $L-1$ to 2 are computed by interpolating from the NG samples of the corresponding nonempty child boxes. In this chapter, we use spatial tri-linear interpolations for phase- and amplitude-compensate fields.

Similar to other stages, the GPU implementation follows the "one-thread-per-observer" parallelization, in which one thread handles one observer, and threads handling observers of the same box are bundled to form one or more blocks. The interpolation process includes calculating coordinates of the NG samples of parent boxes, transforming them into the coordinate system of their child boxes, extracting coordinates and amplitudes of the nearest grid samples around the observers, calculating the interpolation coefficients, and finally evaluating the fields. All these operations are done by a single thread for a single observer. This process seems to be burdensome, but the vast computational power of even a single-stream processor on a GPU handles the jobs with ease.

Table 19.2 The Computational Times and Speed-Up Ratios of the Source-to-NG Stage (Stage 1) on CPU (Xeon X5248) and GPU (GeForce GTX 480) for Different N_p, L, and Oversampling Rates. N_p is the Average Number of Sources per Box on the Level L. The Relation Between N_p and N is $N_p = N/8^L$

Accuracy Requirement	N_p	L	CPU*	GPU	Ratio
L_1 error $= 1 \times 10^{-3}$ for domain size $D = \lambda/2$	16	3	5.89e–3	1.33e–4	44
	32	3	1.55e–2	1.00e–4	155
	64	3	3.16e–2	1.33e–4	238
	N_p	**L**	**CPU**	**GPU**	**Ratio**
	16	4	5.01e–2	6.31e–4	79
	32	4	1.21e–1	9.28e–4	130
	64	4	2.45e–1	1.66e–3	148
	N_p	**L**	**CPU**	**GPU**	**Ratio**
	16	5	4.74e–1	3.99e–3	119
	32	5	1.16e0	6.73e–3	172
	64	5	2.44e0	1.24e–2	197
	N_p	**L**	**CPU**	**GPU**	**Ratio**
	16	6	5.49e0	2.99e–2	184
	32	6	1.07e1	5.23e–2	205
	64	6	2.14e1	9.78e–2	219

* *All timing results shown in this table are in seconds.*

Table 19.3 shows the computational time results of Stage 2 for the low-frequency case. Note that the results are not shown for different N because this stage is solely grid operations that do not depend on N for a fixed L. (We assume all boxes are active, which means at least one source/observer is presenting in any boxes.) The speed-up ratios are in the same range as those of Stage 1 for most problem sizes. It is noted that in the low-frequency regime, Stage 2 takes only less than 2% of the total computational time.

Table 19.4 shows the computational time for the high-frequency regime for the GPU code. The absolute computational time is noticeably larger than that for the low-frequency case in Table 19.3, but it still shows a gain taking into account the number of more operations required for high-frequency calculations. Note that no CPU results are shown for this case because the increased grid density for high-frequency calculation makes the "precomputation approach" used in the CPU not able to produce results for problems in the comparable range. To allow for larger problem sizes would require implementing the on-fly approach also in the CPU code, making it significantly slower. The rest of the results in this chapter for the high-frequency regime are also presented only for the GPU code. The GPU code efficiency is accessed by comparing with the GPU code for the same N for low-frequency problems.

Table 19.3 The Computational Times and Speed-Up Ratios for the NG Aggregation Stage (Stage 2) on CPU (Xeon X5248) and GPU (GeForce GTX 480) for Different L and Oversampling Rates. The Number of NG Samples per Box is Chosen as 64 Due to the Accuracy Requirement

Accuracy Requirement	L	CPU	GPU	Ratio
L_1 error $= 1 \times 10^{-3}$ for domain size $D = \lambda/2$	3	2.70e–3	1.33e–4	20
	4	2.74e–2	3.96e–4	69
	5	2.25e–1	1.77e–3	127
	6	1.81e0	8.33e–3	217

Table 19.4 The Computational Times and Speed-Up Ratios for the NG Aggregation Stage (Stage 2) in High-Frequency Regime for Different L on GPU (GeForce GTX 480). No CGs are Constructed. The Number of NG Samples per Box at Level l is $64 \times 8^{L-l}$, and the Frequency for a Given is Chosen as $D = \lambda(L-2)$

Accuracy Requirement	L	GPU
L_1 error $= 5 \times 10^{-2}$	3	4.14e–4
	4	2.92e–3
	5	3.59e–2
	6	3.89e–1

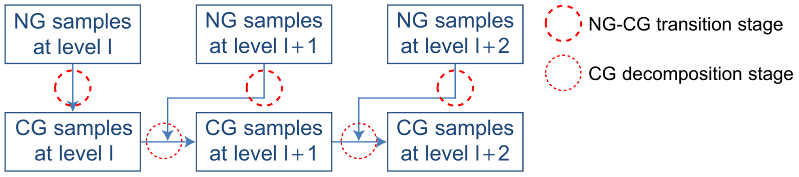


FIGURE 19.4

The relation of two sub-stages: NG-CG transition stage and CG decomposition stage in calculating the field values on CG samples of boxes at each computational level in the low-frequency regime.

19.4.5 Evaluation of Field Values on CG Samples (Stage 3)

In Stage 3, the field values at the samples of CGs are calculated. This stage can conceptually be treated as a two-step process in low-frequency regime as shown in Figure 19.4, but a one-step process in high-frequency regime with some variation. In the two sub-stages in low-frequency regime, each of the two origins of fields, as mentioned in Section 3, on CG samples are computed. In the high-frequency regime, fields on observers are directly obtained via similar interpolation as in low-frequency regime.

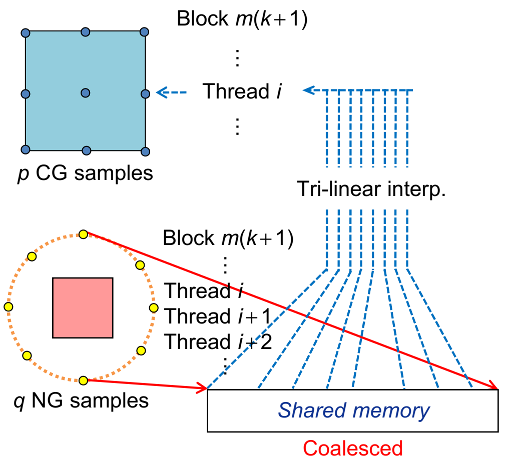


FIGURE 19.5

Memory access scheme of the NG-CG substage. Eight NG samples are required by the trilinear interpolation through which the field values on CG samples are obtained.

In mixed-frequency regime, the process is a hybrid of those two as described in [32]. The "one-thread-per-observer" approach is still adopted as shown in Figure 19.5, where "observers" now are CG samples for in low-frequency regime and actual observers in high-frequency regime.

This substage, evaluating fields of the CG samples from the NG samples in the interacting far-field boxes, is the most time-consuming part of the far-field calculation. The major reason for this larger time, besides a large number of operations needed to be executed, is that the source information required by observers in each observation box is not situated in the contiguous region of the global memory. This random memory access pattern is the result from the fact that "interaction-list" boxes of an observer box belong to a number of different parent boxes or even great parents. In addition to that, one box usually belongs to several "interaction-list" of different observer boxes, making the memory accessing sometimes conflict each other. This "random" relation between source and observer boxes and possible memory loading request of same information from different running blocks may lead to the reduced efficiency of memory handling.

The CG decomposition step is executed for CGs in all observation boxes at the low-frequency levels after the NG-CG transition step. At each such level the CG samples in the child boxes are obtained via trilinear interpolations from the CGs samples in the parent boxes. This step is the NG-NG aggregation stage (Stage 2) in opposite direction, but even simpler as no spherical-Cartesian coordinate transformations are required to account for the relative shift of source (parent) and observer (child) boxes. The total contribution of this substage to the total computational time is low, accounting for only about 1–2%.

Table 19.5 shows the computational time of Stage 3 (NG-CG transition stage and CG decomposition combined) for different L in the low-frequency regime. Similar to the results in Table 19.3 (for Stage 2), for a fixed number of levels L, the speed of Stage 3 is independent of the number of source and observer points N and hence no dependence of N is shown. The obtained computational times

Table 19.5 The Computational Times and Speed-Up Ratios of Stage 3 in the Low-Frequency Regime, on CPU (Xeon X5248) and GPU (GeForce GTX 480) for Different L and Oversampling Rates

Accuracy Requirement	L	CPU	GPU	Ratio
L_1 error $= 1 \times 10^{-3}$ for domain size $D = \lambda/2$	3	2.68e–1	2.44e–3	110
	4	3.02e0	2.32e–2	130
	5	2.95e1	2.11e–1	140
	6	2.43e2	1.82e0	134
	L	**CPU**	**GPU**	**Ratio**
L_1 error $= 2.5 \times 10^{-4}$ for domain size $D = \lambda/2$	3	3.23e0	2.10e–2	154
	4	4.16e1	2.32e–1	179
	5	3.38e2	2.18e0	155

depend on the oversampling ratios of both NGs and CGs, particularly CG. The smallest speed-up ratio list in the table is at the 100+ level, which is in the same order to other far-field stages (comparing with significantly lower speed-up for similar stages in GPU implementation of MLFMA [29, 31]). The speed-up ratios increase for increasing oversampling rates as GPU is less vulnerable to the computational burden owing to the larger number of interpolations. In addition, from our tests on different generations of GPUs, we found that the new Fermi generation (GeForce GTX 480) GPU better handles kernels with access to relatively "random" sets of data in the memory, which results in about threefold computational time reduction on GeForce GTX 480 as compared with the Tesla C1060. This time reduction is interesting, taking into account that the number of stream processors in GTX 480 is only twice as large (480 vs. 240).

Table 19.6 shows the computational time of Stage 3 in the high-frequency regime. The time increases compared with the low-frequency case in Table 19.5, but this increase is relatively insignificant (on the order of the number of levels L), which demonstrates the efficiency of the code in the high-frequency regime.

19.4.6 CG Grids to Observers (Stage 4)

In this stage, the fields at actual observers are interpolated from the CG samples of the finest level L boxes to which the observers belong. This stage is only valid in low- and mixed-frequency regime. This stage is conceptually the Stage 1 in opposite direction. All critical programming strategies in Stage 1 are adopted, including "one-thread-per-observer," coalesced loading of source information, and so on. The computational time of this stage is smaller than that of Stage 1 because only interpolations from the CGs are involved, and no direct evaluations shown in Eq. (5) are needed.

Timing results of Stage 4 are presented in Table 19.7. The computational time behavior is similar to that of Stage 1 (Table 19.2). The GPU computational times are constant for smaller problem sizes. For larger problems sizes, after all stream processor are utilized, the speed-up ratio increases up to a saturation point of around 150. We have also tested more cases with an increase in CG oversampling rates. We found that the increase of the CG oversampling rates barely affects the computational time

Table 19.6 The Computational Times and Speed-Up Ratios of Stage 3 in the High-Frequency Regime on GPU (GeForce GTX 480) for Different L and Oversampling Rates. The Number of NG Samples per Box at Level l is $64 \times 8^{L-l}$, and the Frequency for a Given L is Chosen as $D = \lambda(L-2)$

Accuracy Requirement	L	GPU
L_1 error $= 5 \times 10^{-2}$	3	3.33e–3
	4	4.79e–2
	5	6.16e–1
	6	6.97e0
	L	**GPU**
L_1 error $= 1.5 \times 10^{-2}$	3	2.84e–2
	4	4.39e–1
	5	5.33e0

Table 19.7 The Computational Times and Speed-Up Ratios of CG-to-Receiver Stage (Stage 4) on CPU (Xeon X5248) and GPU (GeForce GTX 480) for Different N_p and L. There are 64 CG Samples per Box. N_p is the Average Number of Sources per Box on the Level L. The Relation Between N_p and N is $N_p = N/8^L$

N_p	L	CPU	GPU	Ratio
16	3	1.18e–3	1.50e–4	8
32	3	1.99e–3	1.50e–4	13
64	3	3.62e–3	1.70e–4	21
N_p	**L**	**CPU**	**GPU**	**Ratio**
16	4	9.77e–3	3.30e–4	30
32	4	1.68e–2	3.30e–4	51
64	4	3.53e–2	5.06e–4	70
N_p	**L**	**CPU**	**GPU**	**Ratio**
16	5	1.03e–1	1.50e–3	69
32	5	1.84e–1	1.50e–3	123
64	5	3.51e–1	2.60e–3	135
N_p	**L**	**CPU**	**GPU**	**Ratio**
16	6	9.01e–1	9.90e–3	91
32	6	1.58e0	1.01e–2	156
64	6	2.95e0	1.79e–2	165

on CPU or GPU, even though with more CG samples per box, more data has to be loaded before doing the interpolations. For the CPU version, the memory-loading time is negligible compared with calculations, whereas for the GPU, the memory-loading time is small owing to coalescent access.

It should be mentioned that, generally, the computational time of Stage 4 is below 1% of the total time. Therefore, the influence of this stage is insignificant provided other stages are implemented efficiently.

19.5 RESULTS

19.5.1 Computational Time

First, we study the computations time in the low-frequency regime. The overall performance of CPU and GPU implementations of NGIM in this regime is shown in Figure 19.6 and Table 19.8. The GPU implementations have been tested on two generations of NVIDIA's GPUs: older generation Tesla C1060 with 4 GB of memory and new-generation Geforce GTX 480 with 1.5 GB of memory. Because the acceleration provided by GPUs varies across stages and is closely related to the problem size, optimal performance of the CPU or GPU is achieved under different parameters. Therefore, similar to [29], we define "effective" speed-up ratio as the ratio between computational times of the CPU and GPU codes when each implementations uses its optimal settings.

In Figure 19.6, the computational time of the direct calculation (i.e., the evaluation of each source-observer pair on CPU and GPU) are provided as a reference. For the direct calculations, the

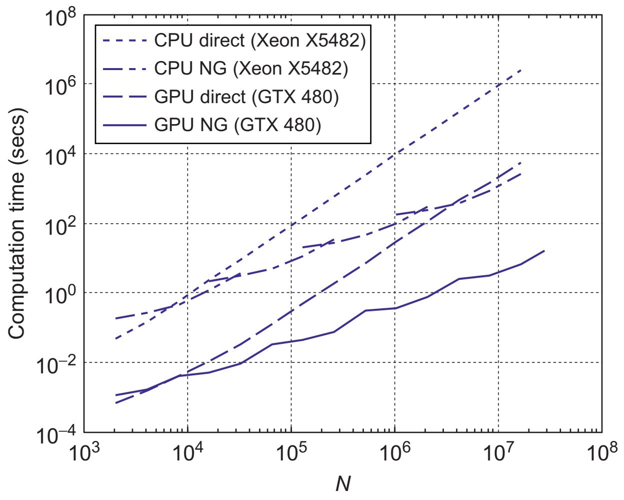


FIGURE 19.6

Computational time of the direct method and multilevel NGIM on CPU and GPU as a function of N. The time of all necessary memory transfer between the hosts and the GPU devices are included, as will be the case for all other timing results in this section. The size of the computational domain is $D = \lambda/2$. The relative L_1 error is approximately 5×10^{-3}.

Table 19.8 Computational Times and Speed-Up Ratios of the GPU and CPU Implementations of the NGIM and the Direct Method for $\Omega_r = \Omega_a = \Omega_x = 2$. The Size of the Computational Domain is $D = \lambda/2$. The Relative L_1 Error is Approximately 5×10^{-3}

# of Unknowns		16K	64K	256K	1M	4M	16M	64M
CPU	Time	1.15e0	4.84e0	2.69e1	9.66e1	3.67e2	2.49e3	N/A
GPU	Time	5.19e–3	3.09e–2	7.49e–2	3.69e–1	2.33e0	6.36e0	N/A
(GTX480)	Speed-up	222	157	359	262	152	392	N/A
GPU	Time	1.15e–2	5.29e–2	1.53e–1	8.14e–1	3.85e0	1.18e1	8.49e1
(C1060)	Speed-up	100	90	176	119	95	211	N/A

computational time scales as $O(N^2)$. The computational time of the NGIM scales as $O(N)$ for both CPU and GPU implementations when optimal L is chosen, respectively, but the GPU code is significantly faster and can handle larger N. The largest problem size N that GeForce GTX 480 can handle is 28 million (1.5-GB memory), while Tesla C1060 can handle $N = 64$ million (4-GB memory). As a comparison, the CPU code can run up to 16 million with 32 GB of RAM. As the acceleration ratio of the near-field components increases within each curve while that of the far-field component remains almost the same, the cross point of curves with different L shifts toward larger N on GPU. It is remarkable that the computational time break-even point between the GPU direct code and the NGIM CPU code is around $N = 4$M. The break-even point between the GPU direct code and the NGIM GPU code is only $N = 4000$.

A detailed list of the computational time is shown in Table 19.8. For example, for a problem with $N = 16,777,072$, the computational time is only 6.36 seconds, which is 392 times faster than the CPU version of NGIM, 862 times faster than the GPU direct version, and 7 million times faster than the CPU direct version (estimated). The comparison between the GPU NGIM code running on a Tesla C1060 and GeForce GTX 480 shows around a twofold speed increase of the latter, which is consistent with the twofold increase of the number of stream processor in GeForce GTX 480 (480 vs. 240). It should be mentioned that the speed-up of GTX 480 compared with the Tesla C1060 of the far-field regime (about three times) is more significant than that of the near-field regime (slightly below two times). We attribute this difference to the fact that GeForce GTX has an improved architecture allowing easier handling of more complex memory loading and thread arrangement required for the far-field calculation.

Table 19.9 lists the computational time when the NG and CG grids are further oversampled to improve the accuracy. Clearly, the computational time increases as a result of more operations being done in far-field stages. As the problem size increases, with a certain L, the near-field component gradually dominates the computational time, and the curves merge. The cross points between different L moves for both GPU and CPU cases. Qualitatively, the performance of the CPU and GPU versions is similar to the low oversampling case, but the adverse effect of oversampling in computational time is much lesser for the GPU code. In fact, the "L_1 error $= 2 \times 10^{-2}$ case" runs at the same speed as "L_1 error $= 5 \times 10^{-3}$" case because our current version of the GPU code launches at least one warp to handle one observer box, as explained in Section 17.4. Therefore, further decreasing of NG/CG samples per box below 32 would not reduce the computational resources consumption on GPUs.

Table 19.9 Computational Times and Speed-Up Ratios of the GPU and CPU Implementations of NGIM with Different Oversampling Rates. The Size of the Computational Domain is $D = \lambda/2$

# of Unknowns			1,048,576	8,388,608
L_1 error $= 2 \times 10^{-2}$	CPU NG	Time (sec)	5.29e1	4.46e2
	GPU NG	Time (sec)	3.69e–1	3.11e0
		Speed-up	143	143
L_1 error $= 5 \times 10^{-3}$	CPU NG	Time (sec)	9.66e1	8.14e2
	GPU NG	Time (sec)	3.69e–1	3.11e0
		Speed-up	262	262
L_1 error $= 1 \times 10^{-3}$	CPU NG	Time (sec)	3.43e2	N/A**
	GPU NG	Time (sec)	1.02e0	8.87e0
		Speed-up	336	N/A

Table 19.10 Computational Times and Speed-Up Ratios of the GPU and CPU Implementations of NGIM for the Surface Source-Observer Distribution of the "inverse-T" Structure in Figure 19.7

# of Unknowns		8,192	32,768	131,072	524,288
CPU NG	Time (sec)	4.35e–1	2.03e0	8.34e0	4.41e1
GPU NG	Time (sec)	2.22e–3	6.33e–3	2.16e–2	1.12e–1
	Speed-up	196	321	386	394

*The size of the computational domain is $D = \lambda/2$, L_1 error $= 5 \times 10^{-3}$.

Finally, the performance of the GPU and CPU implementations of the NGIM in the low-frequency regime is illustrated for a surface problem in Figure 19.2. All sources are placed on the surface of an "inverse T-structure." For this problem, FFT-based methods would require sampling the empty volume using excessive zero padding, resulting in a significant increase of the computational time and memory consumption. As evident from Table 19.10, the CPU and GPU implementations of the NGIM have performance similar to (and even better than) that obtained for the source/observer distribution in a box. The GPU-CPU speed-up ratios are high as well.

Next, we show the computational time results for the GPU NGIM code in the high- and mixed-frequency regimes. The general behavior of the code performance is similar to that for the low-frequency regime and hence many conclusions and discussions for the low-frequency regime apply here as well. In Table 19.11 and 19.12 we present a quantitative summary of the results in the format similar to that in Table 19.1. Table 19.11 shows the computational time in the high-frequency regime for the number of sources up to $N = 14$ million and domain sizes up to $D = 12\lambda$. The computational time is consistent with the time of the low-frequency regime taking into account the anticipated increase related to L. As in the low-frequency case, the speed increase for the newer-generation GPU is around twofold.

Table 19.12 shows the computational time in the mixed-frequency regime with the hybrid NG-CG transformation scheme. It is evident that the code is efficient in handling this case. The computational

Table 19.11 Computational Time of the NGIM on GPUs in the High-Frequency Regime. The Number of Sources N is Taken such that there are Around 20 Sources per Linear Wavelength. The Density of NGs are to Keep the L_1 Error Less than 5.0e-2 for all Problem Sizes

# of Unknowns	8K	64K	216K	512K	1M	4M	8M	14M
Domain size (D/λ)	1.0	2.0	3.0	4.0	5.0	8.0	10.0	12.0
Level L	3	3	3	4	4	5	5	5
GPU Time GTX 480	5.0e–3	2.8e–2	2.0e–1	4.1e–1	9.6e–1	4.9e0	1.0e1	2.1e1
GPU Time TESLA C1060	1.4e–2	6.1e–2	3.8e–1	8.9e–1	1.8e0	1.0e1	2.0e1	4.4e1

Table 19.12 Computational Time in the Mixed-Frequency Regime. The Domain Size is Set to be $D = 2\lambda$. The L_1 Error is 6.0e-2 for All Cases. For the Case $N = 64$M, the Optimal Level L should be 7, but Owing to the Memory Limitations, the Results are Shown for $L = 6$

# of Unknowns	64K	256K	512K	1M	4M	8M	32M	64M*
Level L	3	4	4	5	5	6	6	6
GPU Time GTX 480	2.6e–2	9.2e–2	2.1e–1	3.7e–1	1.8e0	3.1e0	N/A	N/A
GPU Time TESLA C1060	4.4e–2	1.7e–1	3.7e–1	8.6e–1	3.1e0	7.0e0	1.2e1	8.0e1

time for the same N is noticeably smaller for the results in Table 19.12 as compared with those in Table 19.11 because the grid density for the low-frequency levels is a constant of $O(1)$, and the number of high-frequency levels is reduced.

19.5.2 Memory Usage

The memory usage is a very important factor affecting the applicability of a GPU-based algorithm because GPUs typically have smaller amounts of memory than CPU-based systems. For example, in the Dell Precision T7400 workstation used to test our algorithm, two Xeon processors share 32 GB of RAM, but our GPU, an NVIDIA GeForce GTX 480, has 1.5 GB of global memory. In our current version of the code, the memory usage of GPU is determined by both L and N.

With 1.5 GB of memory, the NVIDIA GeForce GTX 480 can handle up to $L = 6$ and up to $N = 28,000,000$ using NGIM code presented in this chapter. The memory consumption of our CPU code used for comparison is significantly larger (while at the same time, much slower). For example, the memory required by NGIM CPU implementation for a problem of $N = 8,388,632$ is 18.1 GB for the same accuracy requirements. This is almost 50 times more than that of the GPU code. Most of the CPU memory is used for storing the coefficient matrices, which if eliminated would cause the computation time of the CPU code to be significantly longer.

19.6 INTEGRATING THE GPU NGIM ALGORITHMS WITH ITERATIVE IE SOLVERS

We have coupled the CUDA NGIM accelerator with our volumetric and surface electric field IE solver. In these solvers the NGIM implemented on GPUs is run four times for each one of the three vector

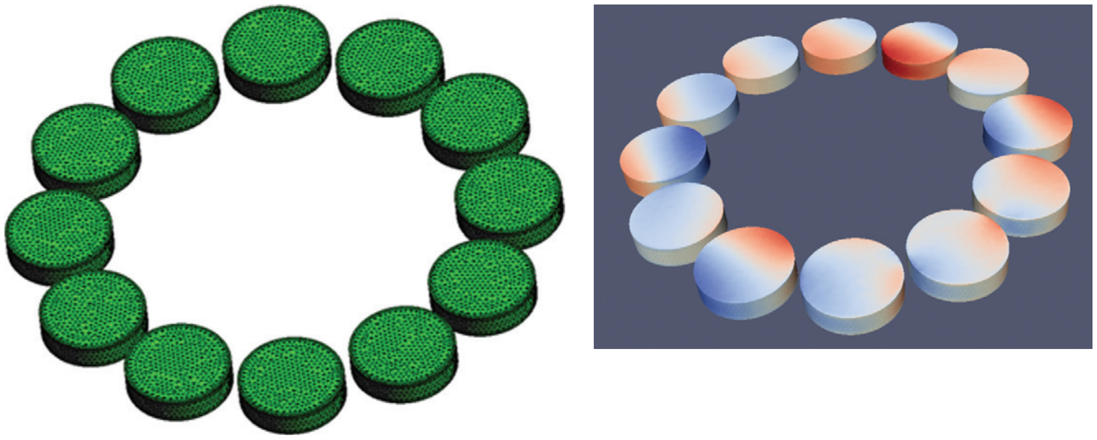


FIGURE 19.7

The mesh figure and solved flux density in 12 dielectric resonators.

components of the vector potential and one scalar potential in Eq. (2). The solution was iterative via tfqmr iterative solver, which evaluates a single iteration with three field evaluations, totaling 12 scalar matrix-vector products per iteration. All other operations, which have a complexity of $O(N)$, are implemented in a Fortran CPU code. The solver can handle a large scattering problem very efficiently. All results in this section are obtained on an inexpensive desktop with an Intel i7-920 CPU operating at 2.66 GHz and an NVIDIA GTX 280.

Figure 19.7 demonstrates the performance of our general-purpose volume integral equation solver accelerated with NGIM and GPUs. The figure shows the color map of the electric flux density distribution in an array of single-mode optical resonators. Each resonator is a cylinder of a diameter of 200 nm and a height of 50 nm. The number of resonators is 12. The resonators are arranged in a circle with the smallest distance between the resonators of approximately 25 nm. The whole structure was meshed with a tetrahedral mesh of 201,958 elements. The structure was excited by a y-polarized plane wave propagating in the x-direction with the free-space wavelength of 1 μm. The total computational time of the simulation was 81 sec. The simulation was completed in 17 iterations with 4.7 sec per iteration. The preprocessing time was 4.7 sec.

Figure 19.8 demonstrates the performance of our surface integral equation solver. The figure shows the structure and the equivalent current distribution on a structure comprising 4 perforated films, which is excited by a plane wave propagating in the z-direction with the free-space wavelength of 1 μm. Each film is of size of $1 \times 1\,\mu m$, thickness of 20 nm, and is made of gold with assumed Drude model for permittivity. The films are modeled via their corresponding surface impedance, which is taken into account in the IE. This structure supports plasmonic resonances due to the interactions between the holes and between the plates. Respectively, the current exhibits bright spots. Because of the plasmonic resonances, the structure needs to be discretized densely. The number of unknowns in this example was above $N = 500{,}000$. There were 300 iterations to converge. The total time for a single iteration on a GTX GeForce 280 was 5 secs. This time was dominated by the local operations running on the CPU, whereas the time of the N to N superposition evaluated via NGIM on GPUs was only 0.5 second, i.e., 10% of the total time.

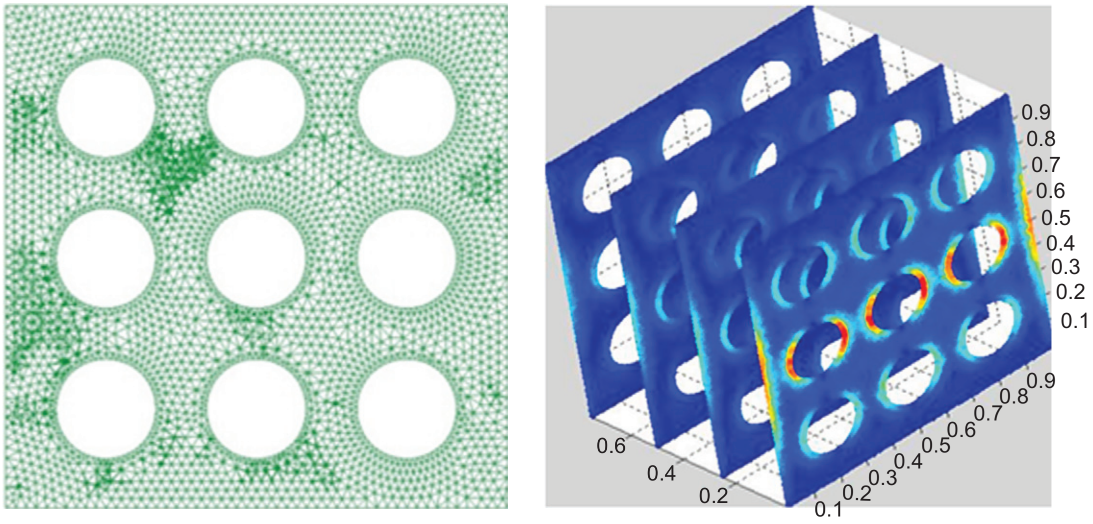


FIGURE 19.8

The mesh figure and solved current distributions on 4 parallel plates.

The resulted computational performance of the volume and surface solvers on GPUs is much smaller than that corresponding to the same solvers running on a CPU. It is also compares favorably to many other results reported for similar structures.

19.7 FUTURE DIRECTIONS

Our future work will focus on the following directions. First, we will port the entire electromagnetic solver to the GPU platform. This step will result in a faster and more robust code. Second, we will improve and extend the NGIM implementations. In particular, we will incorporate higher order interpolations to result in better error control. Because of the massive parallelization, the computational time of the code with higher order interpolations will not be significantly different as compared with the times quoted here. However, the GPU-CPU speed-ups will be more significant as the CPU time for higher order interpolations is noticeably larger. Second, we will improve the way the grids are constructed. In the current version of the code, the grids are constructed for all directions in space. In the future codes, the grids will be constructed only around the structure of interest, thus reducing the computational time. Third, we will implement the code in heterogeneous systems with multiple nodes comprising multiple GPUs. This will involve using MPI and Open-MP to parallelize across multiple GPUs. It should be mentioned that we also use the NGIM (assuming a static kernel with $k_0 = 0$) to accelerate micromagnetic solvers and achieve a very high overall performance of the micromagnetic codes [8, 9].

References

[1] A.F. Peterson, S.L. Ray, R. Mittra, Computational Methods for Electromagnetics, IEEE Press, New York, 1998.

[2] R.G. Belleman, J. Bdorf, S.F. Portegies Zwart, High performance direct gravitational N-body simulations on graphics processing units II: an implementation in CUDA, New Astron. 13 (2008) 103–112.

[3] J.E. Stone, J.C. Phillips, P.L. Freddolino, D.J. Hardy, L.G. Trabuco, K. Schulten, Accelerating molecular modeling applications with graphics processors, J. Comput. Chem. 28 (2007) 2618–2640.

[4] S. Peng, Z. Nie, Acceleration of the method of moments calculations by using graphics processing units, IEEE Transa. Antennas Propag. 56 (2008) 2130–2133.

[5] L. Greengard, V. Rokhlin, A fast algorithm for particle simulations, J. Comput. Phys. 73 (1987) 325–348.

[6] V. Rokhlin, Rapid solution of integral equations of scattering theory in two dimensions, J. Comput. Phys. 86 (1990) 414–439.

[7] L. Greengard, J. Huang, V. Rokhlin, S. Wandzura, Accelerating fast multipole methods for the Helmholtz equation at low frequencies, IEEE Comput. Sci. Eng. 5 (1998) 32–38.

[8] S. Li, B. Livshitz, V. Lomakin, Graphics processing unit accelerated O(N) micromagnetic solver, IEEE Trans. Magn. 46 (2010) 2373–2375.

[9] R. Chang, S. Li, M. Lubarda, B. Livshitz, V. Lomakin, FastMag: Fast micromagnetic solver for large-scale simulations, J. Appl. Phys. 109 (2011) 07D358.

[10] V. Jandhyala, E. Michielssen, B. Shanker, W.C. Chew, A combined steepest descent-fast multipole algorithm for the analysis of three-dimensional scattering by rough surfaces, IEEE Trans. Geosci. Remote Sens. 36 (1998) 738–748.

[11] H. Cheng, L. Greengard, V. Rokhlin, A fast adaptive multipole algorithm in three dimensions, J. Comput. Phys. 155 (1999) 468–498.

[12] G. Brown, T.C. Schulthess, D.M. Apalkov, P.B. Visscher, Flexible fast multipole method for magnetic simulations, IEEE Trans. Magn. 40 (2004) 2146–2148.

[13] H.W. Cheng, W.Y. Crutchfield, Z. Gimbutas, L.F. Greengard, J.F. Ethridge, J.F. Huang, et al., A wideband fast multipole method for the Helmholtz equation in three dimensions, J. Comput. Phys. 216 (2006) 300–325.

[14] B. Shanker, H. Huang, Accelerated Cartesian expansions — A fast method for computing of potentials of the form R-[nu] for all real [nu], J. Comput. Phys. 226 (2007) 732–753.

[15] E. Bleszynski, M. Bleszynski, T. Jaroszewicz, AIM: adaptive integral method for solving large-scale electromagnetic scattering and radiation problems, Radio Sci. 31 (1996) 1225–1251.

[16] J.R. Phillips, J.K. White, A precorrected-FFT method for electrostatic analysis of complicated 3-D structures, IEEE Trans. Comput. Aided Des. Integr. Circuits Sys. 16 (1997) 1059–1072.

[17] A.E. Yilmaz, J. Jian-Ming, E. Michielssen, Time domain adaptive integral method for surface integral equations, IEEE Trans. Antennas Propag. 52 (2004) 2692–2708.

[18] H. Bagci, A. Yilmaz, V. Lomakin, E. Michielssen, Fast solution of mixed-potential time-domain integral equations for half-space environments, IEEE Trans. Geosci. Remote Sens. 43 (2005) 269–279.

[19] W. Hackbusch, B. Khoromskij, A sparse matrix arithmetic based on H-matrices. Part I: Introduction to H-matrices, Computing 62 (1999) 89–108.

[20] W. Hackbusch, B. Khoromskij, A sparse H-matrix arithmetic. Part II: Application to multi-dimensional problems, Computing 64 (2000) 203–225.

[21] W. Chai, D. Jian, An H-Matrix-based method for reducing the complexity of integral-equation-based solutions of electromagnetic problems, in: Presented at the IEEE International Symposium on Antennas and Propagation, San Diego, CA, 2008.

[22] W. Chai, D. Jian, An H2-Matrix-based integral-equation solver of reduced complexity and controlled accuracy for solving electrodynamic problems, IEEE Trans. Antennas Propag. 57 (2009) 3147–3159.

[23] A. Boag, E. Michielssen, A. Brandt, Nonuniform polar grid algorithm for fast field evaluation, IEEE Antennas Wirel. Propag. Lett. 1 (2002) 142–145.
[24] A. Boag, U. Shemer, R. Kastner, Hybrid absorbing boundary conditions based on fast non-uniform grid integration for non-convex scatterers, Microw. Opt. Technol. Lett. 43 (2004) 102–106.
[25] A. Boag, B. Livshitz, Adaptive nonuniform-grid (NG) algorithm for fast capacitance extraction, IEEE Trans. Microw. Theory Tech. 54 (2006) 3565–3570.
[26] A. Boag, V. Lomakin, E. Michielssen, Nonuniform grid time domain (NGTD) algorithm for fast evaluation of transient wave fields, IEEE Trans. Antennas Propag. 54 (2006) 1943–1951.
[27] B. Livshitz, A. Boag, H.N. Bertram, V. Lomakin, Nonuniform grid algorithm for fast calculation of magnetostatic interactions in micromagnetics, J. Appl. Phys. 105 (7) (2009) 07D541.
[28] J. Meng, A. Boag, V. Lomakin, E. Michielssen, A multilevel Cartesian non-uniform grid time domain algorithm, J. Comput. Phys. 229 (22) (2010) 8430–8444, ISSN 0021-9991, doi: 10.1016/j.jcp.2010.07.026.
[29] N.A. Gumerov, R. Duraiswami, Fast multipole methods on graphics processors, J. Comput. Phys. 227 (2008) 8290–8313.
[30] J. Mahaffey, K. Sertel, J.L. Volakis, On the implementation of the fast-iterative solvers on graphic processor units, in: Presented at the 2010 National Radio Science Meeting, Boulder, CO, 2010.
[31] M. Cwikla, J. Aronsson, V. Okhmatovski, Low-frequency MLFMA on graphics processors, IEEE Antennas Wirel. Propag. Lett. 9 (2010) 8–11.
[32] S. Li, B. Livshitz, V. Lomakin, Fast evaluation of Helmholtz potential on graphics processing units (GPUs), J. Comput. Phys. 229 (2010) 8430–8444.
[33] NVIDIA, CUDA Compute Unified Device Architecture Programming Guide, V2.3, Santa Clara, CA, 2009.
[34] A. Brodtkorb, E.C. Dyken, T.R. Hagen, J.M. Hjelmervik, O.O. Storaasli, State-of-the-art in heterogeneous computing, Sci. Program. 18 (2010) 1–33.