Fast integral equation solvers on Graphics Processing Units for Electromagnetics
Shaojing Li, Ruinan Chang, and Vitaliy Lomakin,
Department of Electrical and Computer Engineering,
University of California, San Diego

In preparation for IEEE Antennas and Propagation Magazine

Abstract
This paper presents a comprehensive survey on current status of integral equation solvers implemented on parallel computing systems accelerated by graphics processing units (GPUs) and proposes several key points for efficiently utilizing this type of fundamentally different processors to accelerate several categories of algorithms used by today’s integral equation solvers. Three spatial interpolation-based algorithms, namely Non-uniform Grid Interpolation Method (NGIM), Box-level Adaptive Integral Method (B-AIM) and Fast Periodic Interpolation Method (FPIM) are described in details to show the basic principles for optimizing GPU-accelerated fast algorithms and guide the future designing process of GPU solvers. Due to intrinsic characteristics and careful implementation of these algorithms, speed-ups between 100 and 300 are achieved by a desktop GPU comparing to a desktop CPU at much lower memory consumption. Critical points to these achievements are the brand new programming patterns of GPU applications that trade excessive memory usage and transfer with increased amount of uniformly distributed arithmetic operations. GPU’s unique memory architecture also plays very important roles in deciding the final performance of a GPU code. The presented methods themselves and the designing principles behind them can find very wide applications to many fields inside and outside of computational electromagnetic society.
1 Introduction

The state of the art of integral equation (IE) electromagnetic solvers have grown tremendously since the development of fast methods, both hierarchical, e.g. fast multipole method (FMM) [1-3] and spatial interpolation based methods, and Fast Fourier Transform (FFT) methods [4, 5]. These methods amortize the computation and memory cost from $O(N^2)$ to $O(N)$ or $O(N \log N)$, where $N$ is the number of spatial degrees of freedom. The development of fast IE methods was followed by efforts of parallelization. These efforts are driven by the need of simulating realistic objects, under the situation of performance saturation of single core systems and the availability of massively parallel hardware architectures.

Recent advancements in the development of parallel hardware architectures open many exciting opportunities. Multi-core and multi Central Processing Unit (CPU) systems have replaced single-core single-CPU computer configurations as the mainstream computing devices. In addition, new Graphics Processing Unit (GPU) equipped computers have emerged and often outperforms conventional CPU-only computers. An important driving force of the GPU development has been its consumer market niche of graphics processing, which in many parts is very similar to scientific computing. This leads to higher transistor density and thus performance of GPUs. For example, NVIDIA GeForce GTX 580 GPU at a cost of about $500 has 512 stream processors with performance of over 1 TFLOPs and memory bandwidth of 140 GBps. This is much more powerful than any existing general-purpose CPU. Multiple GPUs can easily fit into an inexpensive desktop computing node, offering the performance of a small-to-medium size traditional CPU cluster. GPU supercomputers are also being built around the world and 3 of the current world’s top 10 supercomputers are GPU-based[6].

The high arithmetic throughput of GPUs makes them well suited for general purpose scientific computing. GPU computing has become especially attractive with the introduction of NVIDIA CUDA framework [7] and recently established OpenCL standard [8], which allow writing high-level codes in a C/C++-like language. GPUs have been used in many fields of scientific computing, such as computational biology, medical imagining, computational fluid dynamics, astrophysics, micromagnetics [9-15]. In various areas of study GPU implemented computational tools have made qualitative changes in the way problems are approached. For example, in molecular dynamics, famous open source packages such as GROMACS and LAMMPS both have GPU accelerator components that helps reduce the simulations time drastically. Another example is a recently presented micromagnetic simulator, FastMag. This simulator can solve problems of size and with speed never before possible, which allows considering problems not accessible to researcher in the past. Therefore, it is expected that GPU computing also has a high potential for Computational Electromagnetics.

Within the realm of Electromagnetics, GPUs have shown high efficiencies in several numerical tasks. Efficient differential equation solvers, such as Finite Difference Time Domain method and Finite Element Method, on GPUs have been developed and integrated with commercial solvers [16-18]. Efforts in the development of integral methods have mostly been focused in porting existing direct method-of-moment (MoM) codes from CPU systems to GPU systems [19-21]. Several works have also shown GPU implementations of acceleration schemes and accelerated IE solvers on GPUs including FFT-based approaches[22] and hierarchical multi-level approaches [23, Cwikla, 2010 #25]. Given the great potential of, and the rapidly growing interest in, GPU computing, a review of the current state and outline of future opportunities of GPU computing for fast IE methods in Electromagnetics is anticipated to facilitate the development of high-performance Computational Electromagnetics methods and simulators.

In this paper we will (i) review the current state of the art in GPU computing for Electromagnetics, (ii) describe recent advancements in the development of fast methods for IE solvers on GPUs, including hierarchical ($O(N)$ or $O(N \log N)$) methods and FFT-based ($O(N \log N)$ or $O(N^{3/2})$) methods, (iii) present a set of results comparing different methods, and (iv) outline future opportunities for GPU computing in Electromagnetics. Three categories of fast methods are addressed: the non-uniform grid interpolation method (NGIM), and box-
based adaptive integral method (B-AIM), fast periodic interpolation method (FPIM). NGIM belongs to the
cfamily of “tree-codes” such as Fast Multipole Method (FMM), so that the presented results have a wide
applicability in various areas of electromagnetics and computational physics in general. AIM is also a widely
used algorithm to accelerate IE solvers and is of high importance for many applications. FPIM is a new recently
introduced technique, which is highly efficient for accelerating IE solvers for general periodic unit problems. All
presented methods are kernel independent and thus allow addressing problems of several classes by simply
changing the integral kernel, i.e. changing the Green’s function.

The paper is organized as follows. Section 2 briefly summarizes electromagnetic integral equation approach,
outs lines its critical components and bottlenecks, and explores the potential of acceleration techniques. Section 3
discusses the GPU architecture and CUDA programming environment and gives guidelines for developing
efficient computational algorithms suited for GPUs. Section 4 shows how direct iterative IE solvers can be
efficiently accelerated on GPUs. The direct approaches demonstrated in Sec. 4 presents concepts of GPU
acceleration for field evaluations and serves as a baseline for comparing to fast IE solvers on GPUs. Section 5
shows an efficient implementation of NGIM on GPUs. Section 6 presents an efficient implementation of AIM
on GPUs. Section 7 discusses implementations of FPIM on GPUs. GPU-CPU accelerations of 150-400 with
very small absolute computational times for all the acceleration techniques are demonstrated. Section 7
demonstrates results of VIE solvers accelerated by the presented fast techniques and GPUs. Finally, Section 8
summarizes the findings of the presented work.

2 Integral equation solvers

The integral equation for a dielectric material reads [24]

$$\frac{D}{\varepsilon_r} - \nabla \left[ \nabla^\ast \left( k_e D \right) \right] \frac{e^{-j k_0 |r - r'|}}{4\pi |r - r'|} dV' - k_0^2 \nabla \left[ \nabla^\ast \left( k_e D \right) \right] \frac{e^{-j k_0 |r - r'|}}{4\pi |r - r'|} dV' = D^{inc} \quad (0)$$

where \( D \) is the total electric flux density, \( \varepsilon_r \) is the relative permittivity of the material, \( k_0 \) is the free space wave
number, and \( k_e = 1 - 1/\varepsilon_r \) is the contrast ratio.

The electric flux density is expanded over a set of basis functions, i.e. \( D = \sum_{n=1}^{N} D_n f_n(r) \), where \( N \) is the number
of basis functions and \( D_n \) are the unknown coefficients. Hence a discretized form of the VIE is obtained as

$$\sum_{n=1}^{N} \left[ f_n(r) \right] \frac{e^{-j k_0 |r - r'|}}{4\pi |r - r'|} dV' \nabla \left[ \nabla^\ast \left( k_e f_n(r) \right) \right] \frac{e^{-j k_0 |r - r'|}}{4\pi |r - r'|} dV' - k_0^2 \sum_{n=1}^{N} \left[ f_n(r) \frac{e^{-j k_0 |r - r'|}}{4\pi |r - r'|} dV' \right] D_n = D^{inc} \quad (0)$$

Equation (0) can be tested, i.e. integrated, with testing function to result in a system of algebraic equations

$$ZD = D^{inc} \quad (0)$$

where \( Z \) is the impedance matrix and \( D^{inc} \) is the projection of the incident field on the testing functions. For
example, choosing the testing functions the same as the basis function results in the following expressions for
the impedance matrix and incident field elements.
\[ Z_{mn} = \sum_{c} dV \mathbf{f}_m^{\prime}(\mathbf{r}) \cdot \sum_{c} \nabla \mathbf{f}_n^{\prime}(\mathbf{r}^\prime) + \sum_{c} dV \nabla \cdot \mathbf{f}_m^{\prime} \sum_{c} \nabla^{\prime} \cdot (k_e \mathbf{f}_n) \frac{e^{-j_0 |\mathbf{r} - \mathbf{r}^\prime|}}{4\pi |\mathbf{r} - \mathbf{r}^\prime|} \]

\[ -k_0^2 \sum_{c} dV \mathbf{f}_m^{\prime} \sum_{c} \nabla^{\prime} \cdot k_e \mathbf{f}_n (\mathbf{r}^\prime) e^{-j_0 |\mathbf{r} - \mathbf{r}^\prime|} \frac{1}{4\pi |\mathbf{r} - \mathbf{r}^\prime|} \]

\[ D_{inc}^{inc} = \sum_{c} \mathbf{D}^{inc} \cdot \mathbf{f}_m (\mathbf{r}) dV \]

Equations (0)-(0) can be solved iteratively and the left-hand side in Eq. (0) needs to be evaluated repeatedly until convergence is reached [ ].

The integrals in Eq. (0) can be evaluated using quadrature rules with a number of quadrature nodes \( N_q \), accompanied with a singularity extraction procedure [ ]. This results in the following representation of the algebraic equation

\[ P^T \left( Z' + Z^0 \right) P D = D^{inc} . \] (0)

Here, \( P \) is an \( N \times N_q \) matrix that maps the unknown coefficient \( D_n \) to the quadrature node field coefficients, determined via the product \( Q = P D \). The matrix \( Z^0 \) is an \( N \times N \) matrix that describes the local corrections (singularity extractions) in the potential. The matrix \( Z' \) is dense and it represents a mapping from \( N_q \) scalar charges to \( N \) scalar observers with its entries given by \( e^{-j_0 |\mathbf{r}_n - \mathbf{r}|} / |\mathbf{r}_n - \mathbf{r}| \). The matrix \( P^T \) is the transpose of \( P \) and it maps the quadrature node fields to the testing function coefficients. The matrices \( Q \), \( P \), and \( Z^0 \) are sparse and have non-vanishing entries only for elements corresponding to overlapping basis-testing function domains. The computation of the matrix-vector products corresponding to these matrices scales as \( O(N) \). The matrix \( Z' \) is dense and the corresponding transformation

\[ D'_m = \sum_{n=1,\neq m}^{N_q} e^{-j_0 |\mathbf{r}_n - \mathbf{r}|} \frac{1}{|\mathbf{r}_n - \mathbf{r}|} Q_n m = 1, 2,..., N_q \] (0)

typically is a major bottle neck of iterative electromagnetic integral equation solvers. The computational cost of the transformation in Eq. (0) scales as \( O(N_q^2) \) if the summation is evaluated directly. The computational cost can be reduced to \( O(N_q) \) or \( O(N_q \log N_q) \) via various acceleration algorithms[2, 4] and using parallelization.

The goal of this paper is to demonstrate how the transformation in Eq. (0) and the IE in Eq. (0) can be solved rapidly on GPUs using the direct summations in Eq. (0) with \( O(N_q^2) \) computational cost and using fast algorithms with \( O(N_q) \) or \( O(N_q \log N_q) \) computational cost.

3 GPU computing and CUDA programming environment

A GPU contains a certain number of stream multiprocessors (SM), each working as a Single Instruction Multiple Data (SIMD) processing unit, e.g. GeForce GTX 580 has 16 SMs. Each SM has 32 stream processors for the NVIDIA GPUs (and 64 for AMD GPUs). GPUs have several memory types, including global memory, shared memory, and registers. The global memory is accessible by all SMs and their stream processors. Each SM has a certain amount of shared memory, which is accessible simultaneously by all its stream processors. Each stream processor has a small amount of registers. In addition, there are special-purpose texture memory, which is designed to better accommodate 2-D or 3-D addressing and possess a certain amount of linear...
filtering/interpolation functionality. The total number of stream processors can be very large (e.g. up to 512 for NVIDIA GPUs and up to 1536 for AMD GPUs) and the memory access can be very fast (provided it is accomplished via proper approaches). This makes GPUs well suited for carrying out mathematically heavy operations often required in scientific simulations.

In 2007, NVIDIA released CUDA, the first high-level general purpose programming language for GPUs. [7] In the CUDA framework, GPUs use execution threads to populate data to stream processors and control the code execution. A separate thread cannot be executed alone. Instead, 32 threads are bundled together as a warp and distributed by scheduler to be executed on a SM at the same time. Warps can further be bundled into thread blocks, with 1 – 32 warps (32-1024 threads) per block. Threads within the same block may not be active at the same time but they share memory, which can be synchronized during their execution.

Figure 3.1. NVIDIA Fermi Architecture

This unique processor and memory architecture give GPUs certain advantages and disadvantages. One major advantage of GPUs is their fast shared memory. Shared memory in high-end GPUs has up to 2 Tbps bandwidth and can be accessed by threads within a block simultaneously. Shared memory, therefore, should be used when possible to reduce the global memory access. However, shared memory is only accessible when the specific block is active on the stream multiprocessor, so every time a block of threads is dispatched by the scheduler to multiprocessors, it has to be loaded with content from the global memory and has its content written back after the block finishes its job. This communication between shared and global memory usually suffers a noticeable
latency. This latency can be alleviated by coalesced access, which is triggered when threads in a warp access contiguous memory address space as well as when the L1 cache is used (this cache was recently added on Fermi-architecture GPUs). This requires the programmers to organize the data in global memory in such a way that threads in the same block access contiguous data addresses. Moreover, GPU algorithms usually have distinct optimization strategies comparing with their CPU version. For example, many CPU implementations of algorithms usually tabulate data used multiple times to save time in computations. GPUs generally have relatively less memory but are very efficient in arithmetic operations. As a result, it may be beneficial to perform arithmetic operations on GPUs on the fly. This may simultaneously reduce the memory consumption and alleviate the aforementioned impact of global memory access latency. Another property of current GPUs is that they lack complicated branch prediction and flow control circuits, which requires making tasks as uniform as possible across a single block. In addition, the bandwidth of GPU-CPU and GPU-GPU communications is much smaller than the bandwidth of global or shared memories within GPUs so GPU-CPU or GPU-GPU communications should be minimized and carefully examined and scheduled.

4 GPU implementations of $O(N^2)$ iterative IE solvers

The evaluation of the discrete spatial convolution in Eq. (0) is the major task of iterative IE solvers. When the summation in Eq. (0) is evaluated directly, the computational cost scales as $O(N^2)$. In this section we show how GPUs can be used to speed-up this summation dramatically. The methods used here will also be used in Secs.5-7 for more complex fast techniques for the evaluation of the spatial convolution.

One approach to compute the spatial convolution on GPUs is “explicit matrix” approach, in which impedance matrix is allocated, filled and transferred to the GPU device and matrix-vector multiplication is then performed using general purpose intrinsic or user-defined CUDA kernels. Literatures illustrate this approach include Ref. [19, 21, 25, 26] In these papers, this “explicit matrix” approach is shown to produce substantial speed-ups but storing the impedance matrix on GPU sets the upper limit of the problem size to about only $N = 10,000$ on most GeForce cards. As a matter of fact, even on CPUs with a much larger amount of available memory this approach would lead to inherent problem size limitations. To allow addressing larger problems, an approach in which the impedance matrix are transferred to the GPU device block by block and the matrix-vector products are executed with these blocks. This approach allows considering matrices of any size, but it suffers from major performance reductions due to the need for frequent CPU-GPU memory transfers. A simple solution to this problem is to compute the elements of the impedance matrix on-the-fly [27]. This approach allows considering problems of any size, avoids unnecessary memory operations, and leads to a very high performance. In the following paragraphs, this approach will be referred to as “GPU direct method”.

The basic principle of the GPU direct method is one-thread-per-observer and block-by-block accumulation of fields. One-thread-per-observer is chosen instead of an alternative one-thread-per-interaction option because the latter would require substantial amount of shared memory and the code would be memory bandwidth limited [27]. The block-by-block traverse ensures each global memory access is coalesced and the data loaded to the shared memory can be reused by all threads in the same block. Figure 4.1 shows the thread arrangement and memory loading scheme and Listing 4.1 is the source code of the device kernel.

The computational time of both CPU and GPU direct method is shown here in Figure 4.2. The comparison is between one core of Intel i7 950 CPU and NVIDIA GeForce GTX 570 GPU. It is evident that the GPU provides very large speed-ups for the direct method (up to 2500, comparing with CPU “on-the-fly” calculation). The GPU has the performance of around 500 GFLOPS in single precision in this case. Interestingly, the speed-ups are higher than the number of the GPU core (480 for the GPU). This large speed-up is obtained not only due to the large core count but also due to the larger GPU memory bandwidth. Importantly, this high performance is obtained starting from small values of $N$ (as small as 8k). The high performance stems from the simplicity of the problem, which leads to uniform execution path of parallelization and arithmetic-intensive nature of the
algorithm. Though the computational time increases as $O(N^2)$, the GPU version of the direct method is still useful for moderately large problems. Interestingly, as will be shown when analyzing fast methods in Secs. 5-7, a CPU fast (e.g. $O(N \log N)$) method on a single CPU core does not outperform the GPU direct code until $N$ is as large as one million.

Figure 4.1 Thread execution pattern of GPU direct method, as in Ref. [27].
Figure 4.2 Computational time of CPU and GPU direct method
Listing 4.1 GPU Direct Method kernel code

```c
__global__ void direct_onfly(float *SrcCoordGPU, Complex *SrcAmpGPU, Complex *FieldGPU, float *d_ftest, Complex WaveNumber, int n)
{
    unsigned int tidx = threadIdx.x;
    unsigned int bid = blockIdx.x;
    volatile Complex P;

    __shared__ float s_SrcCord[BLOCK_SIZE * 3];
    __shared__ float s_ObCord[BLOCK_SIZE * 3];
    __shared__ Complex s_SrcAmp[BLOCK_SIZE];

    for (int i = 0; i < 3; i++)
    {
        s_SrcCord[BLOCK_SIZE * i + tidx] = 0.0f;
        s_ObCord[BLOCK_SIZE * i + tidx] = 0.0f;
    }

    s_SrcAmp[tidx].x = 0.0f;
    s_SrcAmp[tidx].y = 0.0f;
    P.x = 0.0f;
    P.y = 0.0f;

    if (bid * BLOCK_SIZE + tidx < n)
    {
        for (int k = 0; k < 3; k++)
        {
            s_ObCord[BLOCK_SIZE * k + tidx] = SrcCoordGPU[bid * BLOCK_SIZE + n * k + tidx] - 0.0f;
        }
    }

    s_SrcAmp[tidx].x = 0.0f;
    s_SrcAmp[tidx].y = 0.0f;

    __syncthreads();

    for (int k = 0; k < BLOCK_SIZE; k++)
    {
        float r[3];
        Complex magn;
        for (int m = 0; m < 3; m++)
        {
            r[m] = s_SrcCord[k + BLOCK_SIZE * m] - s_ObCord[tidx + BLOCK_SIZE * m];
        }
        magn.x = s_SrcAmp[k].x;
        magn.y = s_SrcAmp[k].y;
        GetFldGPU(r, &magn, &P, WaveNumber);
    }

    __syncthreads();

    if (bid * blockDim.x + tidx < n)
    {
        FieldGPU[bid * blockDim.x + tidx].x = P.x;
        FieldGPU[bid * blockDim.x + tidx].y = P.y;
    }
}

__device__ inline void GetFldGPU(float *r, Complex *magn, volatile Complex *Q1, Complex k)
{
    // There are 25 floating point operations per interaction
    float r_amp1, r_amp2, inv_r_amp1;
    inv_r_amp1 = rsqrtf(r_amp2);
    r_amp1 = r_amp2 * inv_r_amp1;

    if (r_amp2 > 1.0e-7f)
    {
        float del_cos, del_sin, del_exp;
        del_cos = __cosf(-k.x * r_amp1);
        del_sin = __sinf(-k.x * r_amp1);
        del_exp = __expf(k.y * r_amp1);
        Q1[0].x += inv_r_amp1 * (magn[0].x * del_cos - magn[0].y * del_sin) * del_exp;
        Q1[0].y += inv_r_amp1 * (magn[0].x * del_sin + magn[0].y * del_cos) * del_exp;
    }
}
```

Listing 4.1 GPU Direct Method kernel code
5 GPU implementations of NGIM-based fast iterative IE solvers

Though GPUs provide thousands times of speed-ups for the direct method, the quadratic scaling of the computational time eventually makes the code unusable for solving complex large-scale structures. Multi-level tree-structured fast algorithms have been widely used to reduce the complexity from $O(N^2_q)$ to $O(N_q)$ or $O(N_q \log N_q)$ on CPUs. Early attempts to port FMM to GPU were devoted to cases of non-oscillatory kernels, e.g. static Green’s function with $k = 0$. [23, 28]. While showing significant overall speed-ups, these works had relatively insignificant speed-ups for the often dominant translation part of the method. A recent book chapter [29] showed more efficient implementations that can handle problems up to 1e7 on NVIDIA GTX 295 GPU with 1.8GBytes memory. Tree-code and FMMs on multiple GPU systems have also been investigated in Ref. [30], in which significant GPU-CPU speed-ups and absolute performance are demonstrated. However, the complex memory access pattern and the need of evaluating special functions of FMM itself may complicate their implementations on GPUs. Electromagnetic FMMs are even more complicated and only a few implementations have been presented [31, 32]. In this paper, we describe the basic principles and procedures of NGIM, which works equally well for electromagnetic problems in all-frequency regime without the need of significant changes. Comparing to other tree-code-like algorithms, NGIM has unique properties in geometric adaptivity, kernel independence, and high parallelizability for GPU systems.

The NGIM algorithm is implemented using a hierarchical domain decomposition method, which is similar to FMM. Computational domain is subdivided into a hierarchy of levels of boxes, which forms an octal tree. For each box, near-field and far-field boxes are identified for distances larger or smaller than a predefined distance (e.g., for distances twice the box length). The observers and sources belonging to a pair of near-field boxes are automatically near-field source-observer pair. The field potentials contributed by sources in the near-field boxes are evaluated directly via superposition. The field potential far from a source distribution is a function with a known asymptotic behavior. This allows smoothing the fast spatial variations of the potential, computing it on a sparse grid, and interpolating to the required observation points (Figure 5.1). Two sets of grids are constructed, namely the non-uniform grids (NG) and Cartesian grids (CG). The NGs are used to represent fields outside a source domain. The NGs are defined for all levels. The CGs are used to represent fields inside a domain free of sources. CGs are defined only for levels below a certain interface level [31]. The interface level is defined as the one for which boxes are sufficiently small compared to the wavelength. All boxes at lower levels have a small size and these levels are referred to as low-frequency levels. All levels above the interface level are referred to as high-frequency levels and their boxes are comparable to or larger than the wavelength.
The whole multilevel NGIM consists of several sequential 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 a NGs serving as sample points for all source boxes at 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): Field values at CGs of an observer box on a specific level come from two contributions. For the boxes at all low-frequency levels below the interface level, the first contribution is from same level interaction-list boxes via their NG samples. These boxes have their parent box as a neighbor of the observer box but exclude those that have already been accounted for in the near-field stage. For the interface level boxes, the first contribution comes from the same level interaction-list boxes as well as all high-frequency boxes, whose hierarchical parents are in their corresponding level interaction-list. The second contribution of field at CGs of all low-frequency levels below the interface level
comes from the parent box. The fields at the CGs of an observer box are found via interpolations from the NG samples in the former case and from CG samples in the latter case.

Stage 4 (CG to observers): The field values at the observation points are obtained by local interpolations from the CGs on the finest level of the domain subdivision.

The computational cost of \( O(N_q) \) in the low-frequency case, \( O(N_q \log N_q) \) in the high-frequency case, and between \( O(N_q) \) and \( O(N_q \log N_q) \) for the mixed-frequency case. The use of local interpolation guarantees the automatic adaptivity to geometrical features since the NGs and CGs are built and processed only around locations where sources and observers are present.

A source code for Stage 4 given in Listing 5.1 shows how GPU obtains field at observers by interpolating from CGs. All other stages have very similar thread and memory arrangements. For each stage, “one-thread-per-observer” type of parallelization is implemented, which is similar to the GPU direct approach described in Sec. 4. Here, the “observers” are the actual observers where the final field values are computed but they also can be intermediate observers such as grid samples, depending on the actual tasks done at each stage. 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. One or several thread blocks may be launched to handle a certain box if the number of observers is large but one block does not handle multiple boxes to avoid divergent branches within each block. Since the operations and data required by observers belonging to the same box are always shared the same threads within the same block can share data during the computation. This allows reducing the memory access count, using shared memory, and using the coalesced access when transferring data from the global to shared memory.

Field transformations on grid samples are done via interpolations. Linear and cubic Lagrange interpolations are chosen here due to their simplicity and high accuracy. All interpolations, including the computation of the interpolation coefficients are done on-the-fly. This reduces the memory consumption and the global memory read and write penalty. We also have tested an approach in which the interpolation coefficient are tabulated in the pre-processing step and used while the simulation. The on-the-fly approach on GPUs is faster than the pre-computation approach in some cases. On the other hand, in the CPU implementation of NGIM, it is critical to use the pre-computation approach as it is much faster than the on-the-fly approach. The need to pre-compute the interpolation coefficients results in a much larger memory consumption. Implementation details of each stage and stage-by-stage performance breakdown are described in Ref. [-]. Here only the overall performance of NGIM is shown.
Listing 5.1 Source code for Stage 4 of NGIM. It shows how fields on actual observers are obtained via Lagrange interpolation on GPU.
Figure 5.3 shows the computational time of CPU and GPU version of NGIM for low-frequency and high-frequency calculations are shown. We can see that the asymptotic complexity is $O(N)$ for low-frequency and $O(N \log N)$ for high-frequency as the density of sample points of NG does not decrease as the distance between source and observer increases. The speed-ups between the GeForce GTX 570 GPU and one core of Core i7 950 CPU are around 150-400 for the range of N from 2K to 16M. Due to memory limitation, the largest problem size that a single GeForce GTX 570 GPU with 1.25 GB memory can run is 28M. A single Tesla C1060 card with 4 GB memory can handle problems of 64M unknowns for low-frequency application and 160M for static applications.

![Graph showing computational time of CPU and GPU Direct Method](image)

Figure 5.3 Computational time of NGIM. Time of CPU and GPU Direct Method is shown for reference.

Memory consumption as a function of problem size is worth mentioning as well. Due to different implementation of interpolations between GPU and CPU versions of NGIM, CPU NGIM consumes much more memory than the GPU NGIM. For example, the memory required by NGIM CPU implementation for a problem of 8M is 18.1GBytes which is almost 50 times larger than that of the GPU code.

6 GPU implementations of AIM-based fast iterative IE solvers

Another category of the fast algorithm uses FFT to reduce the complexity of convolution operation between sources and Green’s function to $O(N \log N)$. The most popular FFT-based fast methods are the Adaptive Integral Method (AIM) and pre-corrected FFT (pFFT). [4, 33] They are widely used due to their simplicity, high accuracy, and parallelizability.

These two methods adopt similar philosophy to make the FFT-based convolution applicable to non-uniform source and observer distributions. They create relatively sparse uniform grids of samples, project the source excitations from actual sources to the grid samples, calculate interactions between source and observer grids and interpolate fields on actual observers from the observer grid samples. The projections and interpolations introduce errors and the errors are larger when sources and observer are closer to each other. Both AIM and FFT have a correction mechanism for this inaccuracy. For each observer, interactions from sources residing within a certain range of observer position are identified as the near-fields, and are supposed to be calculated directly. Other interactions from sources outside of this near-field domain will be identified as the far-fields, and are calculated via grid interactions. Since the interactions between grids are done through FFTs, inaccurate near-
field components are calculated again and removed from the total field. Detailed description of the algorithms can be found in Ref. [4, 33].

The Box-based Adaptive Integral Method (B-AIM) algorithm presented here follows a similar philosophy but has a different approach in arranging the data structure for projections and corrections. This data structure makes B-AIM well suited for parallelization on GPU and CPU platforms. Below we describe the stages of the algorithm.

The computational domain is divided into multiple subdomains, called boxes, as shown in Figure 6.1. The number of boxes can be set by the user or by other criteria, e.g. to result in a set average number of sources or observers per box. Empty boxes are excluded from the computations. Two grids are constructed with respect to the boxes, one for emulating the field generated by actual sources, referred to as the source grid and the other for estimating the field resulted on actual observers, referred to as the observer grid. In many cases (e.g. for free space problems) these two grids can be the same, but for some cases, e.g. periodic problems[34], these two grids are shifted to ensure fast convergence of the Green’s function. The grids can be expressed as two \( N_g \)-length vectors, \( \mathbf{I} \) and \( \mathbf{U} \), respectively. The algorithm proceeds with the following stages.

1) Projection: Lagrange interpolations are used for projecting actual sources to the source grid. The interpolation provides convergent results as long as the sources to be projected and observers to be interpolated do not belong to the same or nearby boxes. This condition holds as the interactions between sources and observers within the same and nearby boxes are calculated directly in the later “near-field correction stage.” The result of the projection operation can be expressed as an \( N \times N_g \) matrix \( \mathbf{V} \), so that \( \mathbf{I} = \mathbf{VQ} \).

2) Grids interaction calculation: field generated by source samples on the source grid are calculated at each observer via a convolution \( \mathbf{U} = \mathbf{G}_{\text{grid}} \otimes \mathbf{I} \). This is done by convolving the grid sample matrix with the Green’s function matrix via FFT. The FFTW library is used for the CPU version and NVIDIA’s CUFFT library is used for the GPU version.

3) Interpolation: In this stage, the fields at actual observers are found by interpolating from the field values of observer grid samples. The interpolation operation is the reciprocal operation of the projection so can be expressed as the transpose of \( \mathbf{V} \), so that \( \mathbf{F} = \mathbf{V}^T\mathbf{U} \), where \( \mathbf{F} \) is the approximation to \( \mathbf{F} \). Combining the above equations, this coarse estimation can be summarized as \( \mathbf{F} = \mathbf{V}^T \text{invFFT}[\text{FFT}[\mathbf{G}_{\text{grid}}] \cdot \text{FFT}[\mathbf{VQ}]] \).

4) Near field correction: The approximation near-field parts of \( \mathbf{F} \) are substituted by field values computed directly. This substitution requires a second pass of the previous stages and direct calculation of a portion of \( \mathbf{Z} \). This process has \( O(1) \) complexity for a single observer and \( O(N) \) complexity overall. This correction can be summarized as

\[
\mathbf{F} = \mathbf{F} - \mathbf{V}_{\text{near}}^T (\mathbf{G}_{\text{grid,near}} \otimes (\mathbf{V}_{\text{near}} \mathbf{Q})) + \mathbf{Z}_{\text{near}} \mathbf{Q}
\]
Figure 6.1 Far-field and near-field interactions. Inaccurate near-field subtraction not shown as they follow the same procedure as the far-field calculation.

For classical AIM/pFFT implementations, a list of near sources is usually maintained for each observer (or basis function) to accurately calculate near-fields. Projection and interpolation coefficients are also tabulated. This near-field list can be seen as a sparse matrix whose non-zero elements depend on the source geometrical distribution. In our CPU B-AIM codes we also tabulate such interpolation matrices. This is necessary for the reason stated in Sec. 5, but it requires a large amount of memory for large problem sizes. For GPUs this approach is not optimal for several reasons. First, preparing this sparse matrix usually contains multiple complex branches, which are not suitable for GPUs. Second, the large memory consumption limits the largest problem size GPU can solve, especially when the sparse-matrix is very unstructured. Indeed, relatively limited GPU-CPU speed-ups (of around 7 times) for conventional AIM implementations have been shown in Ref.[22] [Peng, paper].

One core features of our B-AIM presented here is that the grids are associated with the boxes and not with specific observers. The observers are associated with the boxes as well, so the mapping between grids and sources/observers are achieved through a double-stage mapping. This mapping makes sources and observers within the same box have the same near and far field source lists as well as the same projection and interpolation grid points. Box-level division of far- and near-field eliminates the need for keeping the near-field lists for each observer and near-field lists are generated in near-field correction stage from the position of boxes on-the-fly. This approach is similar to the finest level in tree-codes discussed in Sec. 5. In fact, due to these similarities our NGIM and B-AIM implementations share the same modules. This approach not only drastically reduces the memory requirements, but also allows reusing the data, directly associates the boxes with observers to CUDA blocks with threads, allows using coalesced access of global memory (to load the information about the entire box), and allows using the shared memory (for the loaded information for the box) and increases cache hit ratio during the global memory loading. In addition, the use of Lagrange interpolation is optimal for GPUs as they can be easily calculated on-the-fly with the computational cost lower than that of commonly used moment or
multipole matching algorithms[35]. Using interpolations also allows accounting for various complex Green’s functions, e.g. periodic and layered medium Green’s functions.

We note that the box subdivision requires using non-central interpolations (target observers situated not around the middle of interpolation range) for most of the sources but it does not lead to a noticeable performance reduction even for CPUs. Moreover, the use of symmetric projections and interpolations also reduces the CPU memory consumption for tabulating the interpolation coefficients.

In principle, we would choose the average number of sources per box to be a constant, which means the number of boxes and the number of grid samples are proportional to number of sources \( N_q \). This leads to \( O(N_q \log N_q) \) computational complexity of the stage 4. Stages 1 3, and 4 contain only local operations so they have an \( O(N_q) \) complexity. The overall asymptotical complexity of the algorithm is \( O(N_q \log N_q) \). The memory complexity of the algorithm is \( O(N_q) \).

The computational times of B-AIM are shown in the Figure 5.2 for the cubic Lagrange interpolations with 1e-4 average L1 error. A single Nvidia GeForce GTX 570 card is compared with one core of Intel i7-950 CPU. All computational times are obtained using optimal settings for CPU and GPU code, respectively. We can see that both the time of the CPU and GPU codes scales nearly linearly starting from very small problems sizes. The speed-ups between GPU and CPU implementations are around 100-200. For example, one field evaluation using GPU B-AIM costs 0.48 secs for a problem of 1 million unknowns, which is 110, 210 and 2.6e5 times faster than CPU B-AIM, GPU direct method, and CPU direct method, respectively. The largest problem can be solved on GTX 570 cards is 8 million. We also note that the memory consumption can be considerably further reduced by modifying the 3D FFTs to be performance through a set of 1D FFTs, thus eliminating the need for redundant zero-padding currently implemented.

![Figure 6.2 Computational time of CPU and GPU B-AIM method](image)

7 GPU implementations of FPIM-based fast iterative IE solvers

Interpolation based fast algorithms for integral equation solvers also extend to more complex problems such as those contain infinite periodic structures. Periodic structures are an important category of problems and they are usually exclusively solved via integral approach through convolution of sources within a principal unit cell with Periodic Green’s Functions (PGFs) that take account periodic boundary conditions. Most recent efforts on
accelerating the periodic solvers are focused on efficiently representing and evaluating PGFs, but they are still significantly more expansive to handle comparing with Green’s functions in free space. In Ref.[34], a fast algorithm, called Fast Periodic Interpolation Method (FPIM) and inspired by AIM, significantly reduces the required number of PGF evaluations, making the periodic solvers having the similar speed as free space solvers.

In periodic problems, without loss of generality, one unit cell is usually selected as the observer cell where the fields from sources other identical but shifted cells are calculated. Treating it as the origin, FPIM separate the whole computational domain into near region, which consists of cells within a certain distance and far region, which consists of infinite cells beyond that distance. Fields generated by source in the near region are called near-fields, which are interactions between finite number of sources and observers thus can be calculated by any fast methods applicable for free space problems. In our solver, we use either NGIM or B-AIM. Fields generated by sources in the far region are calculated via PGFs, but not directly from sources to observers. Similar to B-AIM, FPIM also builds grids for projecting amplitude sources and interpolating fields to observers. Two major differences comparing with B-AIM are that (i) FPIM usually uses non-coinciding source and observer grids as shown in Figure 7.1 and (ii) FPIM does not have the near-field correction stage as the near fields are separated beforehand and are taken care of by a free-space n-body algorithm.

![Figure 7.1 Projection and interpolation grids for FPIM](image)

The major reason for using non-overlapping grids for projection and interpolation is that it allows using simple Floquet expansion to evaluate PGFs between grid points, which are not allowed by other integral equation methods. The reason Floque expansion can work is that for separated source and observer grids the there is always a sufficiently large transverse separation making the Floquet series convergence exponentially fast. Moreover, any other method for the PGF evaluation can be used. For example, using Ewald approach [36] or other rapidly convergent PGF representations would not require making shifted source and observer grids. [37]

Projections from sources to the source grid and interpolations from the observer grid to observers are done via Lagrange interpolation and interactions between grids are done through FFT-accelerated convolutions, just as in NGIM and B-AIM. The GPU implementation of FPIM is very similar to that of NGIM and B-AIM. The computational times and scaling with the problem size are also very similar. Therefore such timing results are not shown. Instead periodic solver results using GPU implemented FPIM with NGIM are shown in Sec. 8

7 Simulation results

The NGIM, B-AIM, and PGIM have been coupled with a VIE solver. The VIE was implemented via standard procedures: SWG basis functions have been used for the field representations and four point quadrature rules for the tetrahedral elements and surface triangles have been used for the volume and surface integrals. The three
acceleration methods were implemented on GPUs and CPUs as described in Secs. 4-7. In addition, all the differential operators were represented in terms of sparse matrixes. Corresponding sparse matrix-vector multiplications were implemented on GPUs and CPUs as well. Such sparse products constituted a small portion of the total computational time. The matrix equation resulted from the VIE was solved iteratively via a GMRES solver.

We present two sets of examples, including scattering from free-space problems and infinitely periodic problems. Each set starts with a verification example, demonstrating the validity of the solver. This is followed by an example demonstrating scattering from a complex structure discretized over millions basis functions. In all examples, the simulations were run on a desktop with Intel i7-950 CPU with 24 GB of system memory and Nvidia GeForce GTX 570 GPU with 1.2 GB of global memory. It is noted that the largest global memory currently available of a GPU is 6 GB (e.g. for Tesla 2070). The largest problem size that one can handle with the presented solvers would scale correspondingly.

Figure 1 shows radar cross-section (RCS) of a free-standing sphere of diameter $D$ and permittivity as shown in Figure 7.1 at the wavelength $\lambda = D$. RCS obtained via the Mie scattering approach and via the VIE accelerated by GPU using B-AIM are shown and compared. The number of Mie series terms was chosen to achieve a full convergence. The number of SWG basis functions was ???. The VIE solution involved 63 iterations with $1e^{-3}$ convergence error. Note that at each iteration two vector matrix-vector multiplications are executed, which resulted in ??? total number of scalar matrix-vector multiplications. The preprocessing time was 9 secs and the solution time was 3.8 minutes.

Figure 7.1 RCS of a three-layer co-located dielectric spheres

Figure 7.2 demonstrates the solver performance for the problem of electromagnetic scattering from a highly detailed human body [Ref: the detailed meshes were provide Ali Yilmaz at UT Austin; line to website]. The permittivity of the tissue was $41.4 - 18j$. The mesh resolution was 4 mm. The total number of basis function was ???. The incident field was a plane wave coming from the top with a wavelength of 1.57m. The figure shows the RCS for this structure. The results were also verified via the PVIE approach. The simulation involved 100 iterations, corresponding to ??? the total of ??? scalar matrix-vector multiplicaitions. The preprocessing time was ??? and the total simulation time was ???.

19
Next we demonstrate the performance of VIE for periodic structures. The VIE was accelerated by FPIM for the far-field evaluation (as described in Sec. ???) supplemented by NGIM for the near-field calculations. All components were implemented on GPUs.

Figure ??? verifies the accuracy of the solver by comparing the scattering results for a doubly periodic square array with periodicity of ???. Each unit cell comprises a cube of size ???, which was made of material with the permittivity ???. The “exact” results were obtained via an RCWA solver [-], which was well convergent and fast in this case. The total number of basis functions for the VIE was ???. The results are shown for the normal reflection coefficient as a function of frequency. It is evident that the results via GPU and FPIM/NGIM accelerated VIE solver and the RCWA results match well. The resonant behavior is a typical Wood anomaly property of periodic gratings [-]. The total computation time of the VIE solver was ??? per a single frequency, including the preprocessing. Each simulation required around ??? iterations.

Finally, Fig. ??? shows a larger scale simulation of a doubly periodic structure with a complex unit cell comprising multiple split ring resonators; such structure can be used as an isotropic negative index metamaterial, which is a popular application nowadays. The computational time per frequency for this structure was ???, including preprocessing. The simulations required around ??? iterations. Again a resonant behavior was observed in agreement with anticipated behavior.
Figure 7.3

8 Summary

We demonstrated that GPU computing offers exciting opportunities for Computational Electromagnetics in general and for integral equation based methods in particular. We described several important points that should be kept in mind when developing GPU codes. In particular, GPUs have unique processing unit and memory arrangements. Efficient GPU implementations of the code require taking a special care of the data structure. Often performing operations on-the-fly may be more beneficial than using memory. Using memory should be done in a parallel and coalesced manner. When possible shared GPU memory should be used.

Keeping in mind these points, we presented GPU implementations of three fast $O(N_q)$ or $O(N_q \log N_q)$ methods for accelerating IE solvers. These methods include NGIM, which is a tree-type code, B-AIM, which is a modification of FFT-based techniques, and FPIM, which is a recently introduced new technique for accelerating general periodic unit cell problems. GPU implementations of all these techniques were shown to be very efficient with very high (100x-400x) GPU-CPU speed-up rates and very small absolute computational times.

We also coupled these GPU accelerated fast techniques with a conventional VIE solver to result in a powerful tool for the analysis of complex dielectric structure residing in free space or in a periodic unit cell of an infinitely periodic structure. On a simple desktop computer with a middle range GPU (at total cost of $2000) we can rapidly solve problems involving millions of degrees of freedom. Systems with more memory and higher-end GPUs can allow handling significantly larger problems with potentially higher speed. Moreover, we have initial implementations of multi-GPU codes, which demonstrate great opportunities for further scaling of the developed codes to GPU-based clusters. CPU clusters matching the performance of the presented GPU-accelerated fast solvers would be much more expensive, would consume much more power, and would require special facilities. Moreover, GPU systems have a great upgradability, as there is no need for replacing the whole system. Only the GPUs can be easily replaced as a next generation is released.


