Electromagnetic Modeling with GPUs

Ian Will

November 27, 2010

Abstract

Modeling electromagnetic (EM) fields and their interaction with a complex environment is important for developing safe and effective wireless systems. Methods for electromagnetic modeling are computationally intensive, and even small scenario can have long run times. Developments in graphics processing units (GPUs) over the past decade have created an attractive architecture for scientific computing, and many computational methods for EM apply. This paper will give an overview of GPU architecture and history, will briefly describe the field of EM modeling, will summarize past efforts to model electromagnetics using GPUs, and will conclude by considering Finite-Difference Time-Domain (FDTD) and the Parabolic Wave Equation (PWE) on GPUs.

Introduction

Electromagnetic (EM) components are an integral part of many civilian and military systems. Cell phones use EM to communicate with base towers, doctors and transit security workers use EM to see through clothes and skin. Many people depend on radio and television broadcasts over EM for daily news, weather, and entertainment. Military systems depend EM for targeting, communication, and radars. Potential adversaries are equally reliant, making it critical to understand the physical behavior of electromagnetic fields both to design effective systems capable of operating in a variety of environments.

EM interacts with the environment in complex ways, and its important to evaluate the interaction and potential interference (intentional or otherwise)

between these systems. Some of the important research areas that require accurate modeling are studying the impact of EM radiation on the human body; designing efficient cell phone antennas; understanding the complex interaction between a radar pulse and sea, land, and atmosphere environments it travels through; designing military equipment with low radar cross section (RCS) to avoid detection by adversarial radar; and predicting attenuation in urban settings for wireless network planning.

There are many computational methods that model EM behavior. The most accurate are based on direct solutions to Maxwell's Equations, however these are typically also the slowest and most memory intensive. Alternate EM models trade accuracy for speed, but in that trade-off its necessary to ignore particularly complex (and particularly interesting) EM behavior, such as three-dimensional multipath or atmospheric super-refraction. In some cases, those complexities can be safely ignored and system performance can be approximated quickly. But in others complex phenomena must be accounted for, requiring low speed computational models. Improving the speed of these models would allow more dynamic planning with models that better represent dynamic and complex EM behavior. This is especially critical in fast-tempo military planning cycles, which demand high accuracy computational models and present possibly the most complex EM environments.

Graphics Processor Units (GPUs), designed to be powerful and efficient parallel processing architectures for the high-end gaming market, have gained popularity in scientific computing over the last decade as an inexpensive parallel hardware architecture. Graphics hardware is designed for large pixel throughput, and has a lot in common with single instruction, multiple data (SIMD) architectures. GPUs are widely available, inexpensive, and rapidly improving, which makes them an attractive alternative to expensive special-purpose super computing architectures. The number of GPU processor cores has scaled linearly with transistor counts, doubling every 18 months [1]. This is largely because GPU processor hardware is simpler, allowing it to dedicate a larger portion of transistors to arithmetic than CPUs, which dedicate a significant amount of the transistor budget to branch prediction and out-of-order execution [2].

As general purpose use of GPU hardware has increased over the past decade, there has been interest in harnessing it for EM simulation. This paper will discuss applications of GPU to electromagnetic computation. It will begin by presenting an overview of the most prominent computation methods for EM simulation. It will then present a brief history of GPUs, and continue

by discussing the Tesla and Fermi architectures in more depth. It will end by discussing application of the Finite Difference Time Domain (FDTD) method for EM, and presenting some considerations regarding application of GPUs to the parabolic wave equation (PWE).

Computational Electromagnetics

Many modern computing devices use electro-magnetic (EM) waves in the radio frequency (RF) spectrum to communicate. Cell phones use frequencies between 900 and 1900 MHz to communicate with base stations and connect to telephone networks. Laptops use the 2.4 GHz spectrum to communicate over 802.11 protocols with wireless access points, making the Internet easily available in many places. Over the past decade, wireless Internet and phone access has become common place. The RF spectrum is of immense importance to military applications, as many systems rely on wireless links for networks and communication, and radar.

Because of the ubiquity of electromagnetic components in modern electronics, its important for system designers and planners to have accurate tools to predict effective operating distances. There are two basic approaches to estimating RF attenuation due to propagation, one involves solving Maxwells equations, and the other involves empirical formulas designed to match observed phenomenon in particular circumstances. The former is more rigorous and theoretically accurate, but also requires great precision in modeling the environment and a great deal of computation time. The latter are less rigorous, but much faster.

Propagation analysis began with geometric analysis based on ray-tracing. A ray-tracing approach does not provide accurate field-strength predictions at long distances because it fails to account for diffraction or refraction. However, it does provide a qualitative picture of propagation conditions by modeling reflection and basic geometric behavior of waves. It is also less computationally expensive than solving parabolic wave equations, and is used in hybrid propagation modeling in regions where refraction is not severe and diffraction does not occur.

Normal-mode waveguide theory was developed to account for diffraction effects because of ray-tracing's deficiencies. It decomposes solutions of the wave equation into eigenfunctions in terms of normal modes. With a few dominant modes, this provides efficient solutions, but with many significant

nodes, (such as when elevated ducts are present), numerical difficulties arise. It is also difficult to model range-dependent environments using mode theory. These difficulties led to the wide adoption of parabolic equations over mode theory [3].

Parabolic Wave Equation

The parabolic wave equation (PWE) is an alternative approach that fits between the ray-tracing and mode-theory methods in both accuracy and computation time. It is a forward-scatter, narrow-angle approximation the Helmholtz wave equation [4]. It can account for such diverse phenomena as spherical-earth diffraction, atmospheric refraction, surface reflections, and object scattering. There are two primary algorithms for solving parabolic equations: finite-difference (Crank-Nicolson [5]), or split-step/Fourier sine transform. The finite-difference method is O(n^2) and the split-step method is O($nlogn$), but the finite-difference method lends itself more naturally to parallelization.

The parabolic wave equation was first described in the 1940s by Fock [6], but the only practical solutions were constrained to very limited conditions. In 1972 Hardin and Tappert [7] introduced a Fourier Sine Series (FSS) solution for the PWE, but application was focused in underwater acoustics for a decade. In the mean time, Popov [8] applied an Implicit Finite Difference (IFD) solution for PWE to the EM domain. Ko et. al. [9] from Johns Hopkins Applied Physics Lab (APL) applied PWE to tropospheric propagation in 1983, developing a model called Electromagnetic Parabolic Equation (EMPE). The U.S. Navy used this model extensively during the 1980s to analyze test results including tests for the Aegis program. Development of PWE models continued over the next two decades. The Navy's Space Warfare (SPAWAR) System Center San Diego (SSC-SD) developed another PWE model called Radio Parabolic Equation (RPE) [10], later replaced by a hybrid model Radio Physical Optics (RPO) [11]. APL continued to develop EMPE, and renamed it to Tropospheric Electromagnetic Parabolic Equation Routine (TEMPER) in 1988 to differentiate it from similarly named commercial products. These models focused on EM propagation characteristics over open ocean, largely because they were developed for Navy application. However, changing Naval mission drove a need for accurate prediction over land, which led to SPAWAR's development of the Terrain Parabolic Equation

Model (TPEM) [12]. In 1998, RPO and TPEM were combined into a model that handled both over-land and over-water propagation called Advanced Propagation Model (APM) [13] [14]. Like RPO, APM is a hybrid model which uses ray optics and other approximations in regions where propagation is less complex and uses PWE to solve complex low-altitude conditions. F. Ryan developed a full PWE solution model based on APM and RPO but without using hybrid approximations called Variable Terrain Radio wave Parabolic Equation (VTRPE) [15], [16]. VTRPE, APM, and TEMPER are the most widely used models for computing long-range EM propagation in current U.S. Navy applications. APM is the fastest due to its hybrid approximation. VTRPE is the slowest, but arguably the most accurate.

Computation Methods for Solving Maxwell's Equations

Electromagnetic field behavior is most accurately predicted by solving Maxwell's equations directly. Direct solution is much more computationally demanding than aforementioned models, and is typically restricted to small scenarios. Computationally viable methods for solving the equations have been around for decades, but computational power to solve the equations for large scenarios has not been available until recently. Yee proposed one of the favored methods (FDTD) in 1966 [17]. Direct solution methods have typically been used to solve relatively small scale problems such as complex antenna characteristics, radio cross section*s (RCS), co-site interference (many transmitting antennas located in close proximity), or propagation loss across small areas. Larger scale problems, such as the behavior of radar emissions over tens or hundreds of kilometers, are computationally out of reach of direct solutions with current single processors. Many of the algorithms for direct solution lend themselves well to parallelization, resulting in much interest in GPU applications in recent years. There are three popular computational approaches to solving Maxwells equations directly: Finite-difference Time-domain (FDTD), Finite Element Method (FEM), and Method of Moments (MoM).

Finite Difference Time Domain

The FDTD method of solving Maxwells equations was introduced by K.S. Yee in 1966 [17]. This method solves Maxwell's equations using differential equations in the time domain. The problem space is decomposed into a discrete grid of cells (the Yee grid) [17], each of which contains representations of an electric field (E) and a magnetic field (H) offset from each other by half a grid step. The field in each cell interacts with its neighboring cells according to the characteristics of E and H, and the dielectric properties of each cell. Dielectric properties allow each cell to represent a particular type of material, such as air, concrete, dry, or moist soil, etc. Time is discretized into small steps, and Maxwells equations compute E and H for all neighboring cells at each time step, for each cell. E and H are computed at alternating half-steps, so E is computed at $t = 1$, and H at $t = 1.5$. The full solution for a large grid can take many thousands of iterations.

FDTD solutions are typically more memory efficient and faster than other direct solution methods because they only require solving one equation at a given time step, whereas MoM and FEM require managing systems of multiple equations. Because of its efficiency and ease of programming, it is a favored method for computational electromagnetics when it applies. It was also the first method implemented on a GPU [18]. One of the drawbacks of FDTD stems from its grid decomposition. When obstacle boundaries don't align naturally with a grid structure (for instance if they are curved, or at oblique angles to the grid lines), aliasing error is introduced in a stair-step pattern. Conveniently for one major modern application, most cities are organized in grid patterns and decompose well into FDTD problems. Another limitation of FDTD is that its time step is limited to $\Delta t < \frac{h}{c\sqrt{3}}$, this prevents FDTD from being used to solve eddy currents [19]. However, when wavelength and geometry characteristics are roughly comparable (such as for microwaves), FDTD is a very efficient computation method.

There have been many approaches to solving FDTD using GPUs beginning in 2004 with Krakiwski, Turner, and Okoniewski [18], continuing to the present [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] and [31]. Table 1 below presents that work in summary.

Researcher	Year	Speedup	Mcells†	Card
Krakiwsky [18]	2004	10.4	4	GeForce FX 5900 Ultra
Inman [21]	2005	40	2.5	ATI Radeon x800
Adams [22]	2007	429.2	10	GeForce 8800 GTX
Adams [22]	2007	12.4	2.7	GeForce Go 7400
Valcarce [23]	2008	100	2.1	GeForce 8600M GT
Valcarce [23]	2008	540	2.1	Tesla C870
Unno [24]	2009	30	2.8	GeForce GTX 295
Sypek [28]	2009	43	432	Quadro FX 5600
Knappil [26]	2009	15	MW‡	Quadro 5500
Mattes [25]	2010	72	30	GeForce 9600 GT
Zunoubi [29]	2010	99	4.2	GeForce 9800 GT
Donno [32]	2010	42	26	GeForce 9500 GT
Ong [30]	2010	28	440	Tesla S1070
Remcom [31]	2010	19	MW‡	Quadro FX 5600

†Million cells ‡Moving Window (MW) FDTD uses a variable grid which follows a pulse along a two-dimensional plane.

Table 1: Comparison of FDTD GPU work

Finite Element Method

For simulations which require a large number of curved or irregular geometries, the Finite Element Method (FEM) is the preferred method of solving Maxwell's equations. It is a popular method of solving differential equations in many disciplines of physics modeling. However, solving the equations at a given time is not a straightforward as in FDTD. Instead of solving a single equation, a system of linear equations must be solved for each field update at each time step. This requires more CPU operations and more memory to stores the equation matrix.

The first effort to implement FEM programs using GPUs was made by Goddeke, Strzodka, and Turek [33] in 2005, but Liu, Wang, Zhang, and Liao [34] were the first to so FEM on a GPU applied to electromagnetics in 2007. Liu et. al. produced a rudimentary implementation of time-domain FEM using an NVIDIA GeForce FX 5700, but they were unable to achieve interesting speedups. Their initial implementation actually slowed computations down, but with some optimization they were able to produce just under 2x speedups. More work has been done in the past two years with FEM for EM

applications on GPUs, the interested reader is referred to [33] [35] [34] [36] [37] [38] and [39].

Method of Moments

The Method of Moments solution is based on the integral formulation of Maxwells equations. It is good at modeling open ended problems, such a scattering. Like FEM, it also allows complex geometries. Peng and Nie [40] (2008) were the first to implement EM simulations using MoM with a GPU. They observed speedup of 20x running their implementation on an NVIDIA GeForce 7600 GT (675 MHz) over an AMD Athlon 3000+ (1.81 GHz). In an exciting demonstration of the potential performance gains resulting from parallelized algorithms and the rapid rate of GPU hardware performance gains, they found an acceleration ratio of 100x when running the same code on an NVIDIA 8800 GTX (525 MHz, 128 SPs) shortly before publication. E. Lezar and D. B. Davidson have recently published similar efforts [41] [42] to implement MoM using CUDA, resulting in maximum speedups of 65x when running on an NVIDIA GeForce GTX 280 versus a 2.2 GHz single-core AMD Opteron. Other interesting MoM research can be found in [43] and [44].

Having covered some background on computational EM in broad strokes, we will proceed to take a closer look at the details of modern GPU architecture.

GPU History

The term GPU was coined by NVIDIA in 1999 to differentiate the GeForce 256 (the first GPU) from prior graphics hardware. To qualify as a GPU, a card must have both vertex and fragment processing on the same chip. And the GeForce 256 was much more capable than previous graphics hardware, being close to professional computer aided design (CAD) hardware in its ability to process vertexes and fragments, but much less expensive. Prior to GPUs, graphics hardware was divided into two classes: professional 3D workstations capable of CAD and other high-performance graphics applications, and consumer cards were not programmable and were design for 3D acceleration of games. The GeForce 256 bridged that gap, providing similar flexibility and power as specialized architectures at consumer prices.

The next advance in GPU architectures GeForce 256 came in 2001 with

NVIDIA's GeForce 3, which included programmable vertex shaders. ATI introduced the Radeon 9700 in 2002, which included the first programmable 24-bit floating-point pixel-fragment processor, with 24-bit precision [2]. Later that year, NVIDIA's GeForce FX extended floating-point pixel-fragment processing to 32-bits. In 2004 NVIDIA release the GeForce 6800, which had a scalable architecture for balancing vertex and pixel shader loads. Both the FX and the 6800 shaders could be programmed using Cg, an extension to the C language with syntax for defining a shader program to operate on one vertex or one pixel, and interact with GPU hardware. Brook [45], another high-level language for programming GPUs, was also introduced in 2004.

GPU technology continued to push forward in 2005 with the Xbox 360s inclusion of a unified GPU programming environment–executing vertex and pixel-fragment programs on the same hardware. NVIDIA introduced their Tesla architecture in 2006 with the GeForce 8800. Tesla adopted a unified shader model, using the same processing hardware to execute pixel and fragment shader programs. It also introduced the CUDA programming language, which provides an alternative API for programming GPU hardware, as a provision for high-performance application development. The GeForce 8800 had 128 cores, and could concurrently process 12,288 threads. Nickolls [1] provides a helpful table listing the advances in GPU architecture over the past decade in Table 2.

The unification of vertex, and fragment processor hardware was perhaps the single most important advance in GPU technology for general purpose use. On earlier graphics hardware, vertex processing and pixel-fragment processing happened on separate processors. The vertex processing hardware supported low-latency, high-precision math, as was required for transforming comparatively smaller sets of vertexes. Fragment processing hardware was optimized for high-latency and low-precision, as was necessary for reading textures from higher latency texture memory and applying it to pixels. As requirements for more programming flexibility emerged, the design of vertex and pixel hardware has converged. Typically there are more pixels than vertexes, since pixels represent the filled shapes and vertexes only define the corners. Because of this, non-unified graphics hardware contained more pixel processors than vertex processors by a factor of three. However, the average case of 3 pixels per vertex was rarely encountered in practice, so there were usually idle processors. In some cases, such as very large or very small triangles, the processing imbalance leading to under-utilization of much of the hardware. As graphics APIs became more complex, the

Date	Product	Transistors	CUDA cores	Technology
1997	RIVA 128	3 million	-	3D graphics accelerator
1999	GeForce 256	25 million	-	First GPU, programmed with DX7 and OpenGL
2001	GeForce 3	60 million	-	First programmable shader GPU, programmed with DX8 and OpenGL
2002	GeForce FX	125 million	-	32-bit floating-point (FP) programmable GPU with Cg programs, DX9, and OpenGL
2004	GeForce 6800	222 million	-	32-bit FP programmable scalable GPU. GPGPU Cg programs, DX9, and OpenGL
2006	GeForce 8800	681 million	128	First unified graphics and computing GPU, programmed in C with CUDA
2008	GeForce GTX 280	1.4 billion	240	Unified graphics and computing GPU, IEEE FP, CUDA C, OpenCL, and DirectCompute
2009	Fermi	3.0 billion	512	GPU computing architecture, IEEE 754-2008 FP, 64-bit unified addressing, caching, ECC memory

Table 2: Timeline of NVIDIA GPUs [1]

imbalance between vertex and pixel processor loads became more dynamic. The problem was even more exaggerated for general purpose applications; researchers developing general purpose applications for GPUs had to choose whether to implement their algorithm using the high-precision but fewer vertex processors, or on low-precision but numerous fragment processors. Using general purpose processor cores to execute both vertex and fragment processors makes all cores available for processing at all stages of the graphics pipeline. This is helpful for graphics programs in certain cases, but is critical for general purpose programming.

In addition to unifying the processor hardware in Tesla, NVIDIA also developed a new C-based API for programming GPU hardware called CUDA.

CUDA brought the increasingly powerful and generic SIMD architectures on GPUs within reach of a much broader scientific computing audience. Prior to CUDA, a few ambitious researchers undertook the herculean effort of mapping their scientific algorithms into graphics APIs, but this was not a task that could readily be applied *en masse* to production code. In making the hardware more accessible to non-graphics programmers, CUDA allowed programmers to focus efforts on crafting code for better utilization of the hardware rather than spending time finding circuitous routes through graphics-oriented APIs. Figure 1 shows a comparison of speedups achieved from mapping a scan algorithm to CUDA as opposed to implementations run directly on a traditional CPU, and mapped to GPU hardware using the OpenGL graphics API, done by Harris et al. [46]. The CUDA implementation outperforms OpenGL by as much as seven times, and is up to twenty times faster than the CPU implementation. A figure depicting the performance gains from CUDA from [46] is reproduced in Figure 1.

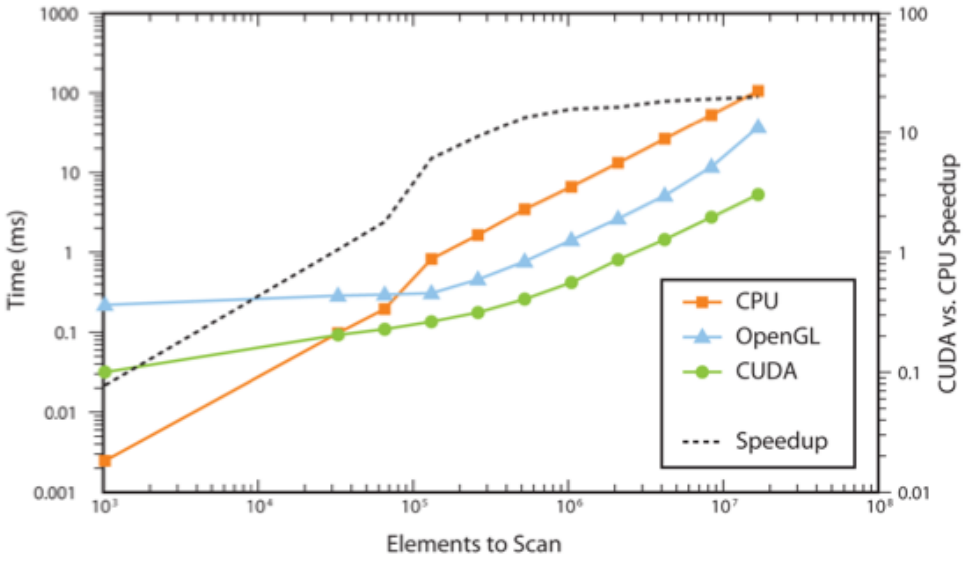

Figure 1: Comparison of CPU, OpenGL, and CUDA speedups [46]

After the introduction of Tesla, general purpose use of GPUs became much more accessible and more widely adopted. Some researchers even built inexpensive massively parallel super-computers by connecting many GPU

nodes. To meet this market, NVIDIA produced GPUs in form factors that were more conducive to building clusters of large numbers of GPUs for high performance computing (HPC) (e.g. rack-mounts with multiple GPUs). This line of products began with the Tesla C870 in 2007. They contained two, four, or eight T8 GPUs (architecturally equivalent to the GeForce 8800).

NVIDIA continued producing GPUs in different form factors for desktop graphics and scientific use with the GeForce GTX 280 / Tesla C1060 / T10 in 2008. Each T10 GPU consists of 240 cores, offering 1-teraflop-per-second peak single-precision floating point performance. It also included support for double-precision 64-bit floating point arithmetic.

In 2009, NVIDIA introduced the Fermi architecture, which adds error correcting codes (ECC) for memory protection, improves double-precision floating-point performance, and adds a larger and more flexible caching system. The GeForce GTX 400 [47] was the first consumer-packaged graphics card based on the Fermi architecture.

The Tesla Architecture

Fundamentally, all GPU architectures are organized around numerous small processing cores called streaming-processors (SPs). Tesla organizes SPs into three layers: at the top level are texture/processor clusters (TPCs), the second level is streaming-multiprocessors (SMs), and the third level is composed of SPs. The number of TPCs in a Tesla implementation can vary, but the number of SMs with a TPC and the number of SPs within an SM is fixed, for example the GeForce 8800 has 8 TPCs, and the GTX has 15. Each TPC is composed of sixteen SPs, divided evenly between two SMs. Because of this, the number of cores supported by any Tesla architecture must be divisible by sixteen.

Tesla Multithreading Model

Before continuing to discuss the Tesla architectural organization in further depth, it will be helpful to cover the multithreading model it supports. Thread parallelism is centered around SPs. Each SP executes a small program called a shader (in graphics parlance) or kernel (in general-purpose parlance). These small programs process a single pixel, vertex, or compute a single value.

Threads are executing the same shader program are grouped together into a warp. Each warp can contain up to 32 threads. All threads within a warp execute the same program, in fact all threads execute the same instruction in unison. Warps are mapped to SMs by hardware in the TPC. SM circuitry manages scheduling of up to 24 warps simultaneously, so each SM can concurrently manage up to 768 threads. Threads in a warp execute the same instruction at the same time, but instruction address and register state are maintained for each thread independently allowing threads to take divergent execution paths. When threads take different branches, every instruction from each branch is executed, but threads which have not taken the current branch are suspended while those that have taken the branch execute the branch instructions. Thus branching is allowed, bringing significant flexibility to the programmer, but at the cost of multithreaded performance, as some threads must block while others execute. NVIDIA calls this architecture single-instruction, multi-thread (SIMT).

There is no synchronization between warps, only within them, so if two warps are executing the same code and take divergent paths they will continue executing independently and will not be suspended or otherwise hindered. This improves flexibility over previous SIMD GPU architectures by allowing branching with shader programs while maintaining very efficient parallelism. The flexibility of the architecture allows programmers to use either thread or data-level parallelism. Programmers can ignore the warp construct and produce valid code, but organizing programs to minimize branching yields the most performance.

In CUDA there is no language level specifier for warps. Instead, kernels can be grouped into thread blocks, and thread blocks can be grouped into grids. Each thread block can contain up to 512 threads, each is identified by a unique thread ID. A thread ID can be composed of up to three dimensions, and threads use these IDs like loop indices. Thread blocks are assigned to a single SM. Although SMs execute up to 24 warps, each SM can only execute eight thread blocks at a time. Thus each thread block will be executed by an SM as multiple warps. Thread blocks can be grouped into grids, allowing the GPU to dynamically load-balance blocks from the same grid amongst SMs. The grid construct allows specifying dependence between groups of thread blocks. One grid may be dependent on another, and the GPU work distribution hardware will prevent execution until dependent grids finish [48]. All threads within a grid execute the same kernel.

Tesla Hardware

A single host interface on the GPU controls communication with the host computer, responding to commands from the CPU, interfacing with system memory, and performing context switching. An input assembler collects geometries and their attributes and sends them to a work distribution unit, which allocates work amongst TPCs using a round-robin distribution scheme. The results are collected in internal buffers and sent to the viewport/clip/setup/raster/zcull unit, which rasterizes the geometries into pixel fragments. Pixel fragments are sent to the pixel work distribution unit, which does not use round-robin distribution and instead delivers work based on pixel location. Alternately, general purpose computation code can be sent directly to the compute work distribution unit, which uses the same round-robin scheme that is used for vertex and geometry shaders. A digram from [48] depicting the Tesla architecture is reproduced in Figure 2.

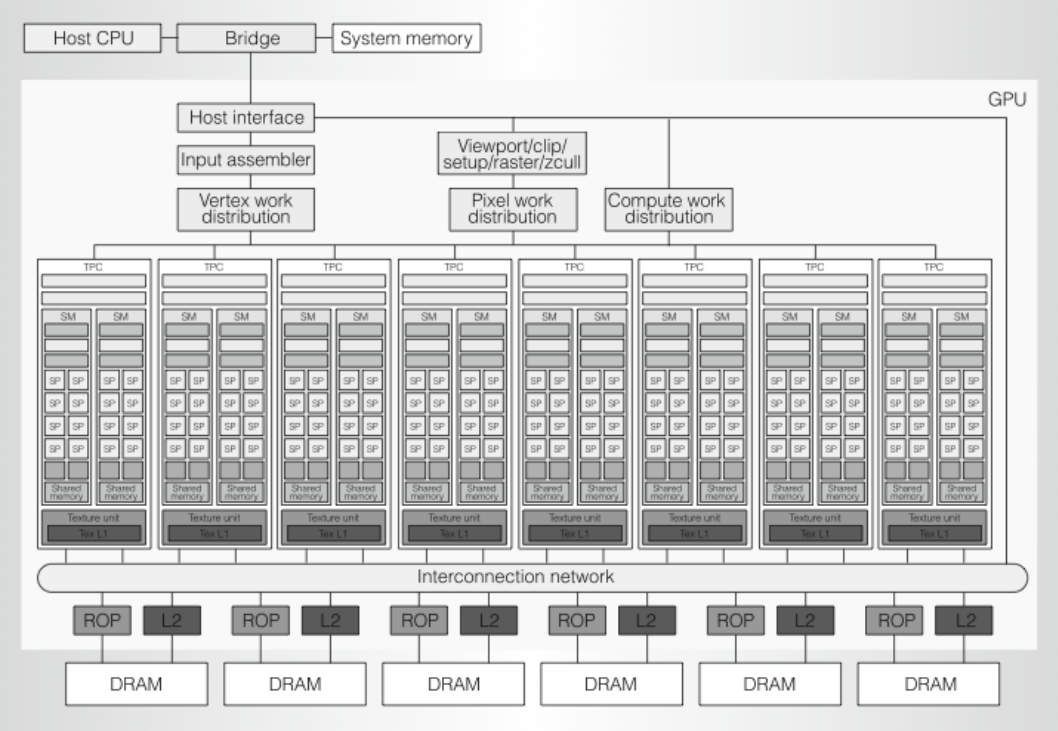

Figure 2: Tesla Architecture [48]

DRAM is organized into six partitions, each containing 1/6 of the address

space and 64 pins, resulting in a total bus width of 384 pins. Memory access is arbitrated to maximize total transfer efficiency by favoring grouping related requests by bank while minimizing latency. Each TPC communicates with the cards DRAM through a higher level interconnect network which carries computed results to DRAM and texture reads back from DRAM. All writes to DRAM through the network pass through raster operation processors (ROPs), which execute color and depth frame buffer operations directly on memory. TPC reads are all directly through an L2 cache before being sent to a TPC.

Each TPC manages scheduling work between its two SMs, and a texture cache that is shared by the two SMs. Work is scheduled to SMs by an SM controller (SMC) which balances vertex, geometry, and pixel workloads; breaks work into threads and warps; assigns threads to SMs, processes results of SM execution; and arbitrates access to the texture unit, load/store path, and I/O path. Each type of workload (pixel, geometry, vertex) has a separate I/O path, but the SMC is responsible for load balancing. The SMC contains the logic to break a thread block (up to 512 threads) into 32-thread warps and allocating those warps to the SMs based on resource availability. The texture unit can execute four threads per cycle, one of each workload type (vertex, geometry, pixel, or compute). Because SPs are pipelined, generally texture unit reads dont interrupt SM thread execution unless data dependencies result in a stall. Texture units themselves are deeply pipelined, and contain a cache to aid locality. Because of the deep pipelining, if a request stream contains some cache hits and some misses, it may not result result in a stall.

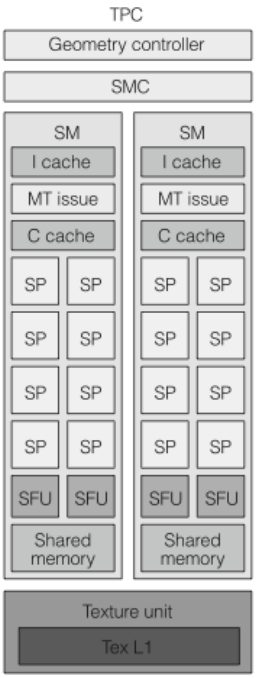

Figure 3: Texture/Processor Cluster (TPC) [48]

An SM is responsible for managing multi-threaded instruction fetch and issue (MT issue), an instruction cache (I cache), a constant cache (C cache), two special-function units (SFUs). 16 KB of fast-access memory is shared amongst the 8 SPs. The SFUs are used for attribute interpolation and transcendental functions (including sine, cosine, binary exponent, log-

arithm, reciprocal, and reciprocal square root). Each of the 8 SPs contain one multiply-add unit, and can execute add, multiple, multiply-add, integer, comparison, and conversion operations. The SPs within an SM are connected via a low-latency network.

A streaming multiprocessor can execute heterogeneous warps concurrently. Warp scheduling is done entirely in hardware by the multithreaded issue unit (MT). The MT saves execution state for each running thread, allowing threads to execute code paths independently. Implementing thread scheduling in hardware provides much more efficient parallelism than software based scheduling, at the cost of hardware complexity.

The MT unit schedules warps based on priority, which is reevaluated at each instruction issue. Priority is scored by a combination of warp type, instruction type, and prior execution time. The scheduler clock operates at half the rate of the cores. The 32 threads within a warp are executed as two sets of 16 threads, over four clock cycles. Because executing 32 threads on the 8 available stream processing cores takes four core cycles, and the scheduler operates at half the core cycle rate, two scheduler cycles will pass before a warp finishes executing its instruction. The scheduler uses these extra two cycles to issue instructions to the special function units, which execute independently of the stream cores. This allows the scheduler to keep all available computation hardware fully occupied.

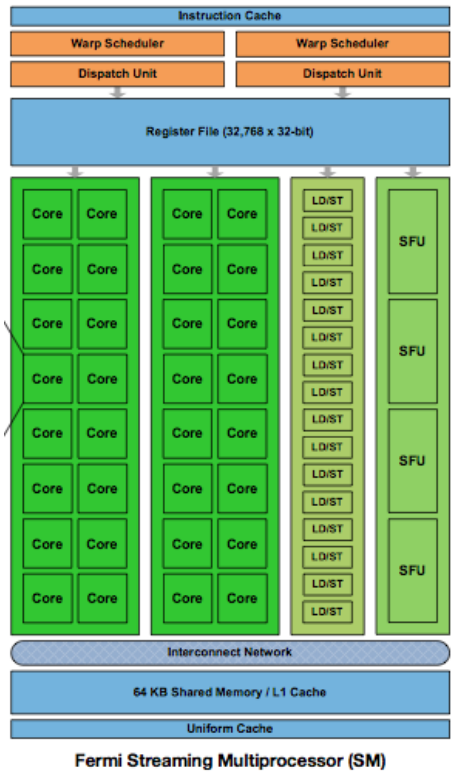

Figure 4: Fermi SM [49]

Fermi Architecture

The Fermi architecture expands Tesla by increasing the number of cores in each SM, adding ECC, and expanding cache sizes. Perhaps one of the most obvious departures from Tesla when looking at architecture diagrams is the

removal of the TPC layer. Instead, texture units were moved inside of SMs, and warp scheduling is done by a new functional unit called a GigaThread scheduler [47]. The streaming multiprocessor organizational concept is maintained, but in Fermi each SM is responsible for 32 cores, up from the 8 in Tesla. Overall, Fermi can support up to 512 cores with 16 SMs [50]. Figure 4 depicts the internals of a Fermi SM.

In Fermi, NVIDIA implements the IEEE 754-2008 floating-point standard, as opposed to the IEEE 754-1985 standard supported by previous GPUs. The 2008 standard introduces a fused multiply-add (FMA) instruction, which is more efficient than a multiply-add (MAD) instruction and equally precise. Overall, Fermi performs floating point operations at about four times the rate of previous GPUs [47].

Fermi also introduces larger caches closer to the processing cores. Tesla architectures had 16 KB of shared memory in each SM. Fermi uses 64 KB that can be configured as 16 KB of shared memory and 48 KB of L1 cache or vice versa, allowing different types of programs (cache intensive or shared-memory intensive) to operate efficiently. Fermi also adds a new 768 KB L2 cache, support for 6GB of on-card DRAM, and adds Error Correcting Code (ECC) protection for all data in memory.

Case Study: Applying FDTD to a GPU

The fundamental premise of FDTD is to divide the simulated region into a two or three dimensional grid called the Yee grid, depicted in Figure 5 [17]. An electric field E is stored at each grid intersection, and a magnetic field H is stored at the center of each grid plane. The grid is excited from one or more cells, and the fields propagate to neighboring cells over a sequence of discrete time steps. After sufficient iterations, the field will propagate through the entire grid. By monitoring the field at each cell, EM properties can be determined at points of interest. The E and H fields are staggered by a half-step in both space and time. This is achieved spatially by positioning the E field components at

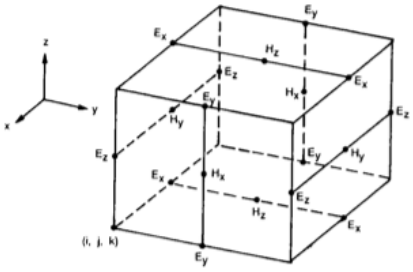

Figure 5: Yee Grid [51]

grid intersections and the H field components at cell plan centers. In time, the staggering is achieved by updating E at whole step intervals and H at half-steps. So at time t, E is computed from the states of previous neighbor cells, and H is updated at $t + \frac{1}{7}2$.

The equations to update the x component of the E field at a grid position i, j, k is given by

$$E_x^t(i,j,k) = E_x^{t-1}(i,j,k) + \frac{Dt}{\epsilon_0} * [\frac{H_z^{t-.5}(i,j,k) - H_z^{t-.5}(i,j-1,k)}{Dy} - \frac{H_y^{t-.5}(i,j,k) - H_y^{t-.5}(i,j,k-1)}{Dz}]$$

, and the equations for x and y components of E, and for H are similar.

The E field updates and H field updates can be represented by two CUDA kernels. The updates for all cells at a particular time step be executed concurrently, which leads to very effective utilization of the GPU. The memory access required by each kernel during its updates is also fairly localized. The update equations for each component at a cell are dependent on the orthogonal components (e.g. y and z for x) of the opposite field, given by the equations below.

Sample code for the H kernel from [28] is

```
__global__ void calcE( float *ex, float* ey, float* ez,
float* hx, float* hy, float* hz, float* ce){

  const int fid = threadIdx.x + blockIdx.x*NUMTH;

  ex[fid] = ce[fid] * (-hy[fid] + hy[fid-IJ]
    + hz[fid] - hz[fid-I]);
  ey[fid] = ce[fid] * (hx[fid] + hx[fid-IJ]
    + hy[fid]-hy[fid-I]);
  ez[fid] = ce[fid] * (-hx[fid] + hx[fid-I]
    + hy[fid]-hy[fid-I]);
}
```

where I is the number of cells in the x direction, and J is the number of cells in the y direction. Such a kernel would be launched in CUDA using the following syntax:

```
int main(int argc, char* argv){
  float* ex = float[I*J*K];
```

```
  float* ey = float[I*J*K];
  float* ez = float[I*J*K];

  float* ex = float[I*J*K];
  float* ey = float[I*J*K];
  float* ez = float[I*J*K];

  float* ce = float[I*J*K];
  float* ch = float[I*J*K];

  //initialize ex, ey, ez, hx, hy, hz, ce, ch
  dim3 threadsPerBlock(I, J, K);
  calcE<<<NUM_TH, threadsPerBlock>>>(ex, ey, ez, hx, hy, hz, ce);
  calcE<<<NUM_TH, threadsPerBlock>>>(ex, ey, ez, hx, hy, hz, ch);
}
```

which means that NUM_TH thread blocks will be created (each having a unique $blockIdx.x$ and $blockIdx.y = blockIdx.z = 1$), and each block will contain I*J*K threads, each with unique threadIdx.x, threadIdx.y, and threadIdx.z triples. Similar code would set up a calcH kernel.

In fact, the kernel decomposition can be done to allow for even further parallelism by creating six kernels–one to update each of the x, y, z components of the E and H fields. This results in kernels like

```
__global__ void calcEx( float *ex, float* hy, float* hz, float* ce){
  const int fid = threadIdx.x + blockIdx.x*NUM_TH;

  ex[fid] = ce[fid] * (-hy[fid] + hy[fid-IJ] + hz[fid] - hz[fid-I]);
}
```

this is also preferable, as it allows for more focused use of shared memory within an SM [28]. With this decomposition, each kernel executes four floating point operations. There are a maximum of $3*I*J*K$ threads operating in parallel per cycle, and each iteration update requires two cycles to update both E and H fields.

At this point we can see that the total memory requirements as a function of grid dimensions I, J, and K is $sizeof(float)*8*I*J*K$. Previous GPU FDTD work varied widely in size, from 2 million cells (Mcells) to 440 Mcells. Using eight floats to represent each grid cell, 440 Mcells requires nearly 12 GB of storage. The Fermi architecture currently only supports up to 6 GB on-card RAM [47], so problems of this size inevitably require host to card

transfers, which reduces potential speed ups. The work by Mattes et. al. [25] sacrifices some computational efficiency to mask this latency by decomposing kernels with some overlap in thread block sub-grids, but maintains speedups of 72x even with card to host transfers.

There are constraints on the size of cells and the length of time step required for numerical stability and validity of results, which prevents solutions from reducing cell counts by making relatively large cells and settling for coarser results. For valid results, the dimensions of the grid must be significantly smaller than the wavelength. Yee uses

$$\Delta = \frac{\lambda}{8} \tag{1}$$

Typically solutions use same Δ for Δx, Δy, and Δz, both to keep code simple and to keep deviation due to numerical approximation and discretization consistent in all dimensions. This makes simulations infeasibly large for moderately sized regions like a house or a city block at the short wavelengths that modern wireless phone and internet networks use. There are ways to avoid this problem, such as running the simulation at relatively lower frequencies and adjusting the results with post-processing. This is the approach used by Valcarce et. al. [23] when modeling a 2.4 km by 3.4 km section of Munich. They used a divider of 10 instead of 8, and brought the 3.3 cm required to model 900 MHz up to 2 m, which equates to 15 MHz. This allowed them to use a grid size of 1204x1704 (2 Mcells) instead of 727,272x1,030,303 (749 Bcells).

Additionally, the time step must be less than the time in which a wave moves the distance of a cell. That is

$$\sqrt{\Delta x^2 + \Delta y^2 + \Delta z^2} > c\Delta t \tag{2}$$

where c is the speed of light in the modeled region [17]. The speed of light in a given material is dependent on its permittivity (ϵ) and its permeability (μ), defined by

$$c = \sqrt{\frac{1}{\epsilon\mu}} \tag{3}$$

.

Furthermore, ϵ and μ can have different values at each cell, meaning c will also vary at each cell. Thus we must computed the minimum and maximum

c over all cells to determine the required size of Δt. The maximum time step Δt can be determined by

$$\begin{aligned} \Delta t &< \frac{\sqrt{\Delta x^2 + \Delta y^2 + \Delta z^2}}{c_{max}} \\ &< \frac{\lambda\sqrt{3}}{8c_{max}} \end{aligned}$$

The number of required iterations i required for a field to traverse the grid can then be determined by

$$\begin{aligned} i &> \frac{dur}{\Delta t} \\ &> \frac{dim}{c_{min}\Delta t} \end{aligned}$$

.

A simulation for the wireless coverage in an average United States household, for example, would require dimensions of about 25 meters in each direction (based on 2,171 ft^2 average household floorspace from a DOE study on energy consumption [52]). The minimum wavelength for 802.11n wireless networks, based on support for both 2.4 GHz and 5 GHz (.125m and .06m respectively), is .06m. Using Yee's cell size divider results in $\Delta = 75$ millimeters. To model the average house for this wavelength, the grid will have approximately 3,333 cells in each dimension, totalling 37 Billion cells in three dimensions. This requires roughly 1.08 TB for grid storage if four byte floats are used. The time step must be $\Delta t < 433.3$ picoseconds to insure each wave is properly captured. This would require a minimum of 1925 iterations. With 6 GB of on-card memory supported by Fermi architectures [47] this would take 184 trips between host and card to transfer the grid with no overlap.

The major features in a house that will impact propagation are roughly on the order of six inches, so if we increase the cell size to capture those features and allow some error as Valcarce et. al. do in [23] we can use grid dimensions of $\Delta = 4.6$ centimeters (about 2 inches) and total cells per dimension of 544 (a multiple of 32, to better match Fermi's 32 core SMs) for a more reasonable grid of 161 Mcells. Using single-precision floating point representation, the grid requires 4.78 GB, which can fit entirely on high-end GPUs. The time step is restricted to $\Delta t < 7.51$ nanoseconds. Only 314 iterations are required for the wave to travel through all dimensions.

Fermi architecture can share up to 48 KB in fast shared memory with an SM, each CUDA thread block is restricted to executing on a single SM, so an optimal allocation of memory to threads and thread blocks will group threads that require access to the same cells. The calcEx kernels described previously could be grouped into the same thread block, allowing adjacent E cells to use the Hy and Hz components that previous cells have already brough in to shared memory.

Assuming a kernel decomposition similar to the one discussed in calcEx, each kernel executes four floating point operations. Each iteration requires three kernels for every cell for the E field and three kernels for the H field computation. Further, because E and H field updates are staggered, all E field kernels must complete before corresponding H field kernels can run. For each iteration, nearly one billion kernel instances will run, totalling 3.8 billion floating point operations or 3,863 Gflops. Fermi can support up to 512 cores, meaning those billion kernel instances could be run as 628,864 batches of 512 cores to compute the E fields and the same number subsequently to compute the H fields. If each floating point operation requires 2 cycles, and the clock speed is 1.5 GHz [47], each field would update in about 1.7 milliseconds, or each iteration (computing both E and H fields) would take about 3.4 milliseconds. A full simulation (314 iterations) would require about 1 second to complete.

In comparison, consider a Nehelem architecture with four cores running at 3 GHz. With four cores, the kernels could run as 241 million cycles. This would take 160 seconds per iteration running at 3 GHz assuming no memory stalls. A full simulation of 314 iterations would take about 14 hours. This is almost a 50,000x speedup, although it assumes optimal parallelization and no memory stalls. Real implementations would not be able to achieve either assumption, but the potential for massive speed ups for FDTD problems is significant. Much of the speedup comes from the increasingly fast GPU cores. Previous generations ran at 500 MHz, but Fermi's support for up to 1.5 GHz for core clocks makes it even more competitive with conventional processors.

Parabolic Wave Equation on GPUs

I was recently able to speak with Dr. Frank Ryan [10], [15], [16], one of the leading experts on electromagnetic propagation modeling using parabolic wave equations (PWE), about the potential for implementing PWE code on

GPUs. He had put a lot of thought and effort into solving PWE using array processors in past years. Basically there are two algorithms to solve PWE, one uses finite differences and operates in O(n^2), the other uses logarithmic convolution and Fourier transforms and operates in O($nlogn$). The finite differences method features more localized computations and lends itself better to massive parallelization, whereas the Fourier transform method is significantly more efficient but requires relatively sparse memory access, which is difficult to execute in parallel (because of data dependencies from far-away memory locations) and difficult to effectively cache the working set without large caches.

In the 1980s, Dr. Ryan attempted to achieve performance gains for his PWE code by implementing his PWE code on an array processor. His experience was that the performance gains he was able to achieve (2x to 10x) did not outweigh the required development time (about a year). By the time the port was finished, similar gains had been achieved in general purpose single-core processor hardware. The array processor PWE code (assembly language) was specialized to a specific array processor architecture which was not increasing in performance as rapidly as other architectures. The array processor code was also difficult to maintain, which presented significant problems as the PWE code was undergoing frequent revisions. Because of this experience with SIMD style parallelization and the hardware of the time, Dr. Ryan focused his attention on powerful, deeply pipelined single-processors with multiple functional units and large on-chip caches.

However, GPUs now offer faster performance increases in hardware than CPUs, and run high-level code which is completely compatible with future generations of GPUs. GPUs also have much improved hardware support for scientific applications, especially notable for PWE solvers are native support for double-precision floating point and transcendentals. The problem that of being left behind by hardware performance gains has now reversed; GPU hardware is increasing at faster rates than traditional MIMD processors. And GPUs can be programmed with higher languages (C++/CUDA), making it more easy to maintain and insuring compatibility with future (faster) generations of GPU hardware. Two of the problems Dr. Ryan encountered when attempting to parallelize PWE in the 80's, namely the slow development time and inability to leverage hardware advances, have become strengths of modern GPUs. A C++ based parallel programming environment is much easier to maintain that assembly code, and GPU hardware is improving more rapidly than CPUs. The Tesla/Fermi architectures, coupled with CUDA, al-

low GPU programs compatibility between generations of GPUs. Because these drawbacks are no longer issues, should implementation of PWE on GPUs be reconsidered? To consider this, we will take a closer look at Dr. Ryan's description of the structure of PWE solutions.

It may be tempting to consider whether the performance gain from an easily parallelized solution such as finite-difference could make up for its algorithmic complexity, but even optimistic performance improvements to the O(n^2) finite difference method are quickly overtaken by O($nlogn$) Fourier convolution method for realistic simulation sizes. The break even point for the two algorithms, optimistically assuming that one could achieve a speedup of 100x for the finite difference (O(n^2)) algorithm, is about $n = 1000$. This is too small to be of any interest, so we must focus efforts to parallelize PWE on the logarithmic convolution algorithm.

The logarithmic convolution method consists of thousands of recursions of a two step function. The first step A is the Fourier transform, which features a large number of non-adjacent memory accesses. The seconds step B accesses a local segment of memory, and includes application of environmental factors to the results of step A. If the environment is homogeneous, B can be executed once, instead of after every recursion of A, however applications that require a PWE solution over faster executing methods typically require non-homogeneous environment representations. Both A and B are operations on large vectors (large, for interesting problems that is, small formulations can be made but are uninteresting practically) of typically 2^{23} elements, each requiring 16 bytes (a double-length complex number, or 8 bytes for the real and 8 for the imaginary components), which equates to 8 MB for each vector. Figure 6 reproduces one of Ryan's sketches on my whiteboard, depicting the recursive nature of the two step function, and the memory access patterns of A and B.

Step A (the Fourier transform step) typically occupies 60% of the run time, and step be the other 40%. But step B is more naturally applied to parallel architectures. Assume we are able to achieve very good speedups for B (say 100x), and are able to mask all memory accesses with computations. Given that B only occupies 40% of the overall run time, Amdahls law states that the net gain from parallelization can not exceed 1.66x. To be useful, a GPU implementation of PWE will need to parallelize both the Fourier transform step A and the linear step B.

One of the major barriers to parallelizing the Fourier transform step, according to Ryan, is its memory access pattern. The Fourier transform

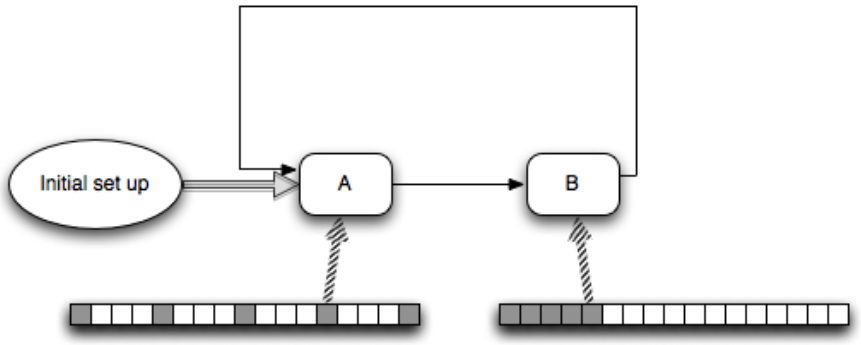

Figure 6: Structure of Fourier Solution to PWE

accesses elements from all over a data vector of complex numbers, which typically has as many as 2^{23} elements. This equates to 8MB of memory per vector, and the Fourier transform step must access blocks spread throughout. To make this efficient, its important that much of the large input vector can be stored in on-chip caches. Because of this, Ryan has focused on architectures with large on-chip caches, especially the Intel Nehalem architecture, which can support 8 MB of L3 cache [47]. Current GPUs have up to 768 KB of L2 cache (no L3 cache). This is actually larger than the 256 KB L2 cache featured in Nehalem. It would be interesting to compare memory access times from GPU DRAM versus Nehalem's L3 cache, but space and time constraints make this out of the scope of this paper.

Although the cache issues may make PWE on GPUs infeasible, many of the other obstacles to Ryan's previous work have been removed by developments in GPUs over the past decade. Cache size is a remaining issue, but caches on GPUs are getting larger [49], and future generations may feature caches large enough to store significant portions of a PWE data vector. Perhaps most convincingly, effort spent implementing PWE in C++ with CUDA will continue to pay off in performance gains by increasingly powerful GPU technology.

Conclusion

Many methods for computational electromagnetics have been demonstrated to work well with GPU style parallelization. The methods that directly solve Maxwell's equations seem to apply best, and there has been significant interest and work in the past few years. Finite Difference Time Domain (FDTD) is perhaps the best match for GPU architecture because its grid decomposition maps well to CUDA's thread block and thread grid programming constructs and memory access patterns are fairly local. Other methods, including Finite Element Method (FEM) and Method of Moments (MoM), have also been successfully implemented with notable performance gains. The parabolic wave equation (PWE), typically used as a bridge between the very accurate direct-solution methods and less accurate high-speed models, has not been implemented on a GPU. Frank Ryan, one of the leading experts on PWE, implemented his PWE code (VTRPE) [15] on a SIMD architecture in the 1980's, but found the performance gains to be small and not worth the high effort require to implement and maintain. GPUs have become an attractive alternative to specialized array-processing hardware, being easier to program and offering architectural compatibility between generations. Many of the obstacles that prevented efficient implementation of PWE on SIMD architecture in the 1980's are not issues with NVIDIA's Fermi architecture, and developing GPU based PWE models may be worth a closer look.

References

[1] J. Nickolls and W. Dally, "The GPU computing era," *Micro, IEEE*, vol. 30, no. 2, pp. 56–69, 2010.

[2] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[3] M. Levy, *Parabolic equation methods for electromagnetic wave propagation*, ser. IEE electromagnetic waves. Institution of Electrical Engineers, 2000.

[4] G. D. Dockery, "Development and use of electromagnetic parabolic equation propagation models for u.s. navy applications," *Johns Hopkins APL Technical Digest*, vol. 19, no. 3, pp. 283–292, Jul 1998.

[5] J. Crank and P. Nicolson, "A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type," *Advances in Computational Mathematics*, vol. 6, pp. 207–226, 1996.

[6] V. A. Fock, "Solution of the problem of propagation of electromagnetic waves along the earths surface by method of parabolic equations," *J. Phys. USSR*, vol. 10, no. 5, pp. 399–420, 1946.

[7] R. H. Hardin and F. D. Tappert, "Applications of the split-step fourier method to the numerical solution of nonlinear and variable coefficient wave equations," *SIAM Rev.*, vol. 15, p. 423, 1973.

[8] A. Popov, "Solution of a parabolic equation of diffraction theory by a finite difference method," *USSR Computational Mathematics and Mathematical Physics*, Jan 1968.

[9] H. W. Ko, J. W. Sari, and J. P. Skura, "Anomalous microwave propagation through atmospheric ducts," *Johns Hopkins APL Technical Digest*, Jan 1983.

[10] F. J. Ryan, "RPE: A parabolic equation radio assessment model," *AGARD Conf. Proc. Operational Decision Aids for Exploiting or Mitigating Electromagnetic Propagation Effects*, no. 453, pp. 1–10, 1989.

[11] H. Hitney, "Hybrid ray optics and parabolic equation methods for radar propagation modeling," *Radar International Conference*, pp. 58 – 61, 1992.

[12] A. Barrios, "A terrain parabolic equation model for propagation in the troposphere," *Antennas and Propagation, IEEE Transactions on*, vol. 42, no. 1, pp. 90 – 98, 1994.

[13] ——, "Modeling surface layer turbulence effects at microwave frequencies," *Radar Conference, IEEE*, pp. 1–6, 2008.

[14] ——, "Considerations in the development of the advanced propagation model (apm) for u.s. navy applications," *Radar Conference, Proceedings of the International*, pp. 77– 82, 2003.

[15] F. J. Ryan, "Users guide for vtrpe," 2005, distributed with the VTRPE program.

[16] P. Ransom and F. Ryan, "Exploration of lloyd mirror rotation effect and radar look down angle," *Antennas and Propagation Society International Symposium, IEEE*, vol. 1A, pp. 400– 403, 2005.

[17] K. Yee, "Numerical solution of initial boundary value problems involving maxwell's equations in isotropic media," *Antennas and Propagation, IEEE Transactions on*, vol. 14, no. 3, pp. 302–307, 1966.

[18] S. Krakiwsky, L. Turner, and M. Okoniewski, "Graphics processor unit (GPU) acceleration of finite-difference time-domain (FDTD) algorithm," *Circuits and Systems, Proceedings of the 2004 International Symposium on*, vol. 5, pp. 265–268, 2004.

[19] A. Bondeson, T. Rylander, and P. Ingelström, *Computational electromagnetics*, ser. Texts in Applied Mathematics. New York, NY: Springer Science+Business Media, Inc., Jan 2005, vol. 51.

[20] S. Krakiwsky, L. Turner, and M. Okoniewski, "Acceleration of finite-difference time-domain (FDTD) using graphics processor units (GPU)," *Microwave Symposium Digest, IEEE MTT-S International*, vol. 2, pp. 1033–1036, 2004.

[21] M. Inman, A. Elsherbeni, and C. Smith, "FDTD calculations using graphical processing units," *Wireless Communications and Applied Computational Electromagnetics, IEEE/ACES International Conference on*, pp. 728– 731, 2005.

[22] S. Adams, J. Payne, and R. Boppana, "Finite difference time domain (FDTD) simulations using graphics processors," *DoD High Performance Computing Modernization Program Users Group Conference*, pp. 334–338, 2007.

[23] A. Valcarce, G. D. L. Roche, and J. Zhang, "A GPU approach to FDTD for radio coverage prediction," *Communication Systems, 11th IEEE Singapore International Conference on*, pp. 1585–1590, 2008.

[24] M. Unno, Y. Inoue, and H. Asar, "GPGPU-FDTD method for 2-dimensional electromagnetic field simulation and its estimation," *Electrical Performance of Electronic Packaging and Systems, IEEE 18th Conference on*, pp. 239–242, 2009.

[25] L. Mattes and S. Kofuji, "Overcoming the GPU memory limitation on FDTD through the use of overlapping subgrids," *Microwave and Millimeter Wave Technology (ICMMT), 2010 International Conference on*, pp. 1536–1539, 2010.

[26] J. Knapil, J. Stack, J. Schuster, and S. Fast, "Moving window finite difference time domain on a graphic processor unit," Apr 2009, technical report by Remcom, Inc.

[27] G. Baron, E. Fiume, and C. Sarris, "Accelerated implementation of the s-mrtd technique using graphics processor units," *Microwave Symposium Digest, IEEE MTT-S International*, pp. 1073 – 1076, 2006.

[28] P. Sypek, A. Dziekonski, and M. Mrozowski, "How to render FDTD computations more effective using a graphics accelerator," *Magnetics, IEEE Transactions on*, vol. 45, no. 3, pp. 1324–1327, 2009.

[29] M. Zunoubi, J. Payne, and W. Roach, "CUDA implementation of TE^z-FDTD solution of maxwell's equations in dispersive media," *Antennas and Wireless Propagation Letters, IEEE*, vol. 9, pp. 756–759, 2010.

[30] C. Ong, M. Weldon, S. Quiring, L. Maxwell, M. Hughes, C. Whelan, and M. Okoniewski, "Speed it up," *Microwave Magazine, IEEE*, vol. 11, no. 2, pp. 70–78, 2010.

[31] R. Belmonte, "JCREW program: Advanced computational techniques for near earth EM propagation including urban areas," Nov 2010, technical report of work done under contract N00173-08-C-2118 with the Naval Research Lab.

[32] D. D. Donno, A. Esposito, L. Tarricone, and L. Catarinucci, "Introduction to GPU computing and CUDA programming: A case study on FDTD [EM programmer's notebook]," *Antennas and Propagation Magazine, IEEE*, vol. 52, no. 3, pp. 116–122, 2010.

[33] D. Göddeke, R. Strzodka, and S. Turek, "Accelerating double precision FEM simulations with GPUs," *18th Symposium Simulations Technique*, pp. 139—144, 2005.

[34] K. Liu, X. Wang, Y. Zhang, and C. Liao, "Acceleration of time-domain finite element method (TD-FEM) using graphics processor units

(GPU)," *Antennas, Propagation & EM Theory, 7th International Symposium on*, pp. 1–4, 2007.

[35] L. Kun, "Graphics processor unit (GPU) acceleration of time-domain finite element method (TD-FEM) algorithm," *Microwave, Antenna, Propagation and EMC Technologies for Wireless Communications, 3rd IEEE International Symposium on*, pp. 928–931, 2009.

[36] N. Goedel, T. Warburton, and M. Clemens, "GPU accelerated discontinuous galerkin FEM for electromagnetic radio frequency problems," *Antennas and Propagation Society International Symposium, IEEE*, pp. 1–4, 2009.

[37] N. Goedel, S. Schomann, T. Warburton, and M. Clemens, "GPU accelerated Adams-Bashforth multirate discontinuous galerkin FEM simulation of high-frequency electromagnetic fields," *Magnetics, IEEE Transactions on*, vol. 46, no. 8, pp. 2735–2738, 2010.

[38] A. Kakay, E. Westphal, and R. Hertel, "Speedup of FEM micromagnetic simulations with graphical processing units," *Magnetics, IEEE Transactions on*, vol. 46, no. 6, pp. 2303–2306, 2010.

[39] A. Dziekonski, A. Lamecki, and M. Mrozowski, "Jacobi and gauss-seidel preconditioned complex conjugate gradient method with GPU acceleration for finite element method," *Microwave Conference (EuMC), 2010 European*, pp. 1305–1308, 2010.

[40] S. Peng and Z. Nie, "Acceleration of the method of moments calculations by using graphics processing units," *Antennas and Propagation, IEEE Transactions on*, vol. 56, no. 7, pp. 2130–2133, 2008.

[41] E. Lezar and D. B. Davidson, "GPU-based lu decomposition for large method of moments problems," *Electronics Letters*, vol. 46, no. 17, pp. 1194–1196, 2010.

[42] E. Lezar and D. Davidson, "GPU acceleration of method of moments matrix assembly using Rao-Wilton-Glisson basis functions," *Electronics and Information Engineering (ICEIE), 2010 International Conference on*, vol. 1, p. 1, 2010.

[43] T. Takahashi and T. Hamada, "GPU-accelerated boundary element method for helmholtz'equation in three dimensions," *International Journal for Numerical Methods in Engineering*, Jan 2009.

[44] R. shan Chen, K. Xu, and J. jun Ding, "Acceleration of mom solver for scattering using graphics processing units (GPUs)," *Wireless Technology Conference*, pp. 1–4, Jul 2008.

[45] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream computing on graphics hardware," *ACM Transactions on Graphics*, vol. 23, no. 3, pp. 777—786, May 2004.

[46] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with CUDA," in *GPU Gems 3*, H. Nguyen, Ed. Adison-Wesley, Jan 2008, pp. 851–876.

[47] P. N. Glaskowsky, "NVIDIA's Fermi: The first complete GPU computing architecture," Oct 2009, produced for NVIDIA, Inc.

[48] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *Micro, IEEE*, vol. 28, no. 2, pp. 39–55, 2008.

[49] NVIDIA, "NVIDIA's next generation CUDA compute architecture: Fermi," Oct 2009, produced by NVIDIA, Inc.

[50] J. Wang, "NVIDIA GF100 world's fastest GPU delivering great gaming performance with true geometric realism," Aug 2010, produced for NVIDIA, Inc.

[51] K. Yee, J. Chen, and A. Chang, "Conformal finite difference time domain (FDTD) with overlapping grids," *Antennas and Propagation Society International Symposium, IEEE*, pp. 1949–1952 vol.4, 1992.

[52] D. of Energy, "Part 1: Housing unit characteristics and energy usage indicators," 2005. [Online]. Available: http://www.eia.doe.gov/emeu/recs/recs2005/c&e/summary/pdf/tableus1part1.pdf