USING GPUS TO ACCELERATE INSTALLED ANTENNA PERFORMANCE SIMULATIONS

Tod Courtney[1], John E. Stone[2], Bob Kipp[1]
[1]Delcross Technologies, Champaign, IL 61821
[2]University of Illinois at Urbana-Champaign, Urbana IL
tcourtney@delcross.com, john.stone@gmail.com, kipp@delcross.com

Abstract: Savant is a asymptotic ray-tracing CEM tool used to predict the performance of antennas installed on electrically large platforms, including far-field antenna patterns, near-field distributions, and antenna-to-antenna coupling. Savant is based on the shooting and bouncing rays (SBR) formulation. While asymptotic solvers like Savant have significantly smaller computational and memory requirements for electrically large problems than full-wave techniques, the computation costs still increase significantly with frequency and simulation fidelity, and such solvers benefit greatly from parallelization techniques. Graphics processing units (GPUs) are throughput-oriented processing devices that are well suited for the mathematically intensive workloads found in CEM solvers. Current GPUs contain hundreds of processing units, leverage thousands of threads, and can execute over one trillion floating-point operations per second. A hybrid CPU and GPU parallelization approach has been developed for Savant, providing significant speedups compared to CPU-only implementations. Results from the execution of GPU-accelerated Savant on multiple case studies will be presented.

1. Introduction

Over the last two decades, there has been a phenomenal increase in the computational power of microprocessors due to the dramatic decrease in transistor size and advances in computer architecture. Initially, the smaller transistor size translated directly into faster clock speeds. However, as the size of transistors continued to decrease, increased clock speeds became unsustainable due to excessive power consumption and heat dissipation. As a result, new methods have been developed to translate the benefits of smaller transistors into increased computational power. One approach is to increase the number of computational cores in the processor. It is now common to find four or six core processors in modern computer workstations, but the complex circuitry in modern CPUs makes it difficult to combine a large number of processor cores into an effective and power-efficient multi-core processor.

The trend toward increased transistor density has enabled the computer graphics industry to make dramatic advances in the power of graphics processing units (GPU). GPU cores are designed for throughput-oriented workloads. Instead of using large amounts of fast cache memory to hide memory latency as is done with CPUs, GPUs are heavily multithreaded and multiplex thousands of concurrently executing threads onto hundreds of processing units in order to keep arithmetic units busy while memory operations are being serviced. This design principle makes GPUs well suited to mathematically

intensive workloads on very large datasets. In order to achieve high performance on these workloads, GPUs contain high-bandwidth memory systems capable of transferring hundreds of gigabytes per second. These hardware differences enable modern GPUs to process significantly more floating-point arithmetic operations (FLOPS) than modern CPUs.

High-frequency asymptotic computational electromagnetics (CEM) solvers offer an approximate, yet computationally efficient, solution for many types of electromagnetics systems. Asymptotic solvers are preferred over full-wave solvers at high frequencies, where the electrical size of the platform becomes too large for the required matrix representation and solution. High-frequency codes do not depend on a large matrix representation and scale more efficiently with increasing frequency. Still, the computational cost of high-frequency algorithms increases with frequency; for many algorithms, such as the one in Savant, complexity grows quadratically with frequency. As a result, even high-frequency codes can require a significant amount of processing time, often requiring hours or days to complete a complex simulation.

In addition to the basic computational requirements of high-frequency simulations, many use-cases for high-frequency codes require the execution of multiple simulations using either different project configurations or different input conditions. Examples of such use-cases include optimization of the placement of multiple antennas on complex platforms, simulation of dynamic scenarios with one or more moving platforms/parts in the scene, and hybrid closed-loop asymptotic/full-wave solutions that repeatedly simulate different portions of the platform with full-wave and high-frequency solvers, exchanging intermediate results at each iteration until a convergence criterion is met. For all of these reasons, high-frequency CEM solvers will benefit greatly from the dramatic speedups offered by GPU-based parallelizations.

2. Asymptotic Solvers & Savant

Asymptotic methods, also known as high-frequency (HF) methods, are widely used to efficiently compute the scattering by objects whose overall size and features are electrically large. At high frequencies (or short wavelengths), propagation of electromagnetic (EM) waves can be approximated by ray bundles, and EM scattering is dominated by local conditions of the scattering body. Perhaps the oldest and most familiar theory is geometric optics (GO). The theory is not without drawbacks, one of which is its failure to model diffraction. Various asymptotic enhancements (*i.e.*, ones that are valid in the limit as wavelength goes to zero) have been proposed to overcome this limitation, including geometrical theory of diffraction (GTD) [1], physical optics (PO), and physical theory of diffraction (PTD) [2].

Savant is a tool for predicting the installed performance of antennas. In particular, Savant focuses on modeling the interaction of the antenna with the installation platform, using an asymptotic methodology known as shooting-and-bouncing rays (SBR) [3,4]. SBR was originally developed to efficiently model RCS for electrically large cavities [3] and later extended for radar signature modeling on realistic targets [5]. It was subsequently adapted to installed antenna applications [6-8].

In SBR, many GO rays are launched toward the scattering object (platform) using a general-purpose geometric ray tracer for complex 3-D CAD models. This determines which surfaces are lit by the antenna. The launched GO rays are vector-field-weighted

by the antenna and represent diverging volumetric ray tubes that "paint" surface currents on the CAD model according to the GO boundary condition (*i.e.*, PO currents). These induced currents are radiated to field observation points or receiving antennas. Next, a set of reflected rays is generated from the first-bounce hit points, with their vector fields updated according to GO and the material properties of the surfaces. Some of these reflected rays escape, while the remainder hit other surfaces of the CAD model, painting second-bounce currents. The process is continued, and in this way, SBR implements multi-bounce scattering mechanisms.

The algorithmic steps of SBR can be summarized as follows:

1. Launch many rays from the Tx antenna toward the platform. The rest of the steps are for each ray.
2. Assign each ray a vector field weight according to the Tx polarimetric antenna pattern.
3. Test each ray for intersection with the surfaces of the CAD model.
    a. If the ray escapes, ignore it.
    b. If the ray hits a surface, generate a reflected ray. Compute its fields according the incident ray fields, the material properties of the surface, and GO principles.
    c. If the ray hits a penetrable surface, also generate a transmitted ray in the same direction as the incident ray and then extend and process it as one would the reflected ray.
4. At the ray hit-point, compute the total GO field (incident + reflected + transmitted) projected on to the surface. Use these and the PO approximation to determine equivalent electric and magnetic surface currents.
5. Radiate the equivalent currents to far-field angles and near-field points by convolving them with the free-space Green's function: the radiation integral.
6. Continue tracing the reflected and transmitted rays generated in Steps 3b) and 3c) and repeat from there. Continue until either the ray escapes (Step 3a) or some maximum bounce limit is reached.

When curved surfaces are encountered, special care must be taken in computing the change in GO fields as they propagate from bounce to bounce, adjusting for the change in ray tube divergence rate [3,9,10]. Savant also implements an extension of SBR to model the surface wave (creeping wave mechanism) based on creeping-wave rays and the surface currents they deposit as they propagate along the surface. These details are not of concern in the current paper.

Among asymptotic ray tracing methods, it is important to bear in mind the joint role of GO and PO in SBR. As in GTD codes, SBR uses GO as an efficient means of accounting for the dominant mechanisms of interaction between surfaces of the scatterer. However, unlike GTD codes, the GO rays are not of direct interest in determining the scattered fields at observation angles and points. Instead, that role is played by radiation of the PO currents (Step 5) induced by the GO rays (Step 4). As such, the terminal condition of GO

rays is not of interest in SBR. Thus, a search for a few critical rays' paths is replaced by shooting many rays (thousands to millions) in order to sample the geometry in detail. This results in the ability to handle more varied and realistic geometric shapes.

From a computational perspective, the costs of performing the SBR algorithm can be broken into two categories: a) the cost of performing geometric ray tracing and associated updates of GO ray fields (Steps 1 – 4, 6) and b) the cost of radiating equivalent currents to observation angles/points (Step 5). In SBR, run time scales as $O(N_r)$, $O(N_o)$, and $O(N_f)$, where N_r, N_o, and N_f are the numbers of rays, observation angles/points, and frequencies, respectively. For a convergent result, N_r is generally proportional to the surface area of the scatterer measured in square wavelengths. When all of these quantities are large enough, run time scales as $O(N_r \cdot N_o \cdot N_f)$. However, as a practical matter, this condition is not often satisfied. For instance, the geometric ray-tracing burden is independent of N_o and N_f. Hence, when $N_o = 1$, one can often increase N_f from 1 to 50 - 100 before doubling run time. Eventually, however, the run time will scale with the number of frequencies. The same can be said for the number of observation angles/points, though its impact is felt sooner. For instance, we observe that for $N_f = 1$, run time doubles for N_o going from 1 to 20. Also, once N_o is large enough that ray processing (Step 5, evaluating the radiation integral) dominates over geometric ray tracing (Steps 1-4, 6), run time doubles when N_f increases from 1 to 4 or 5, at least in Savant's implementation. The memory footprint of SBR does not grow with any of these quantities, other than that needed to hold input and output, and is generally quite modest compared to the capacity of modern systems.

For many installed antenna problems, N_o is large and the computation of the radiation integral (Step 5) is the dominant cost of a simulation, the bottleneck. For example, in computing the installed antenna pattern, one frequently requires hundreds or thousands of observation angles to fill out a 1-D cut or a 2-D sector. Similarly, in computing the distribution of fields in the vicinity of the platform, one is typically interested in thousands of field observation points, sufficient to generate a 2-D image of the fields. Because of its form and the limited need to address contingency conditions in the algorithm, the radiation integral lends itself to efficient acceleration on GPUs, the subject of the present paper.

In its most general form, the radiation integral is a convolution of equivalent surface currents and charge densities with the free-space Green's function, given by

$$\bar{E}_s(\bar{r}) = \frac{1}{4\pi}\int_{S'} dS' \left[\left(\hat{n}' \bullet \bar{E}\right)\nabla' \frac{e^{-jkR}}{R} + \left(\hat{n}' \times \bar{E}\right) \times \nabla' \frac{e^{-jkR}}{R} - j\omega\mu_0 \left(\hat{n}' \times \bar{H}\right)\frac{e^{-jkR}}{R} \right], \qquad (1)$$

where $\bar{E}_s$ is the scattered field at the observation point, S' is the domain of the ray tube at the hit point when projected onto the surface, $\bar{E} = \bar{E}(\bar{r}')$ and $\bar{H} = \bar{H}(\bar{r}')$ are the total electric and magnetic fields at the surface when considering the incident ray and reflected ray (*i.e.*, the total GO field), $\bar{r}'$ is the position vector of the equivalent currents, $\hat{n}'$ is the surface normal, $\bar{r}$ is the position vector of the observer, $R = |\bar{r} - \bar{r}'|$ is the distance from the current sample point to the observer, and μ_0 is the free-space magnetic permeability.

The equivalent currents and charges mentioned in Steps 4 and 5 are those obtained by the cross-product or dot-product of $\hat{n}'$ with $\bar{E}$ or $\bar{H}$, as indicated in Eq. (1).

When *R* is sufficiently large that higher-order terms arising from the gradient operator are negligible, the equation simplifies considerably to

$$\bar{E}_s(\bar{r}) = \frac{-jk}{4\pi}\int_{S'} dS'\left[\hat{r}'\times\hat{n}'\times\bar{E}(\bar{r}') - \eta_0\hat{r}'\times\hat{r}'\times\hat{n}'\times\bar{H}(\bar{r}')\right]\frac{e^{-jkR}}{R}. \quad (2)$$

Still, Eq. (2) must be evaluated through numerical integration. When *R* is also sufficiently large such that $R \gg d'$, where *d*' is the largest dimension of *S*', then the vector dependence can be extracted from the surface integral as a scale factor. The scalar surface integral then accounts for linear phase progression across *S*' and can be evaluated analytically for the typical ray tube shape used in Savant, a triangle. In practical problems of interest, this criterion is often met. It is always satisfied for far-field observation angles and usually satisfied for near-field observation points. It is this analytic form of the integral that is accelerated using the GPU, as discussed in Section 4.

3. GPU hardware architecture and programming model

GPUs were originally designed as highly specialized fixed-function accelerators specifically for computer graphics workloads. As the complexity and computational demands of computer graphics algorithms have grown, GPUs have become progressively more flexible and have incorporated fully programmable processing units. Over the past five years GPUs have evolved further, becoming effective general purpose co-processors for throughput oriented computational workloads with significant data parallelism [11-13]. The latest generation of GPU hardware implements full IEEE double-precision floating point arithmetic, 64-bit addressing, incorporates caches, and uses error correcting codes to protect memory [14,15]. Collectively, these features have helped close gaps in ease-of-programming, reliability, and applicability of GPUs as compared to CPUs, making GPUs ideal accelerators for many arithmetic intensive data-parallel workloads found in science and engineering applications. Below we discuss the attributes of GPU hardware architecture and aspects of the CUDA GPU programming model that are most relevant for antenna modeling calculations. Detailed discussions of GPU hardware architecture and programming models are found in the literature [13-16,19].

3.1 Comparison of CPU and GPU hardware architecture

GPUs were originally designed for acceleration of computer graphics workloads that contain tremendous amounts of data-parallelism, and that by their nature are computationally intensive in terms of both floating-point arithmetic and memory bandwidth. These characteristics have led GPU hardware designs to employ massively parallel hardware architecture, with state-of-the-art GPUs containing over 512 processing units. Individual GPU processing units are organized into groups, so-called "multiprocessors" or "compute units" that share a single instruction decoder. Each of the processing units in the same group executes instructions in lock-step, following the single-instruction multiple-data (SIMD) model. By sharing instruction decoders among groups of processing units, GPUs use much less of the microprocessor die area for control logic, allowing the area to be used for arithmetic logic instead. This approach

enables GPUs to achieve arithmetic performance levels many times that of conventional multi-core CPUs while maintaining power consumption levels that are still roughly par with high-end CPUs.

Another characteristic that differentiates GPUs from CPUs is that they are designed for high overall execution throughput rather than low latency execution, following directly from the needs of graphics workloads. GPUs achieve high throughput at moderate clock rates through data parallelism rather than the approaches taken by CPUs, such as high clock rates, instruction level parallelism, and out-of-order execution. Conventional multi-core CPUs use large caches to mask the latency of read and write operations to off-chip DRAM. Instead of relying on caches, GPUs hide off-chip memory latency primarily by saturating processing units with a large number of threads, and context switching from threads stalled on memory accesses to threads that are immediately runnable. The need for runnable threads to be available for latency hiding increases the total concurrency required to allow a GPU to achieve peak execution throughput. Current generation GPUs require at least 10,000 independent threads of execution to fully utilize all processing units and hide latency for off-chip memory accesses. The extensive use of hardware multithreading reduces the amount of GPU die area spent on cache, thereby making this space available for arithmetic logic.

Since graphics workloads make frequent use of square roots and transcendental functions, GPUs incorporate dedicated machine instructions for fast evaluation of an unusually large number of special functions, as compared with CPUs. CPUs typically incorporate few machine instructions for special functions and often compute only low-precision "estimates" in hardware, requiring math libraries or application code to compute or refine special function results entirely in software. Dedicated GPU machine instructions for special functions give them a significant performance advantage versus CPUs, compounding with the performance benefits gained from massive parallelism. Applications that make heavy use of special functions have been observed to achieve GPU speedups over 100 times faster than a single CPU core [17-18].

The tremendous arithmetic capability of GPUs requires matching memory bandwidth. GPUs incorporate on-board high-bandwidth (over 140 GB/sec) multi-banked off-chip "global" DRAM memory systems to meet this demand. GPU global memory is accessible to both the host CPUs and to the GPU itself. Since current GPUs are not completely self-sufficient computers, input and output must be transferred between the host memory and the GPU over the PCI express bus. GPU global memory systems achieve peak performance when they are accessed in large, aligned, contiguous read/write operations known as "coalesced" accesses. Although current generation GPUs contain caches, they are much smaller than those found in CPUs and are primarily used to help reduce the performance impact of uncoalesced global memory accesses. Compute-oriented GPUs incorporate small, near-register-speed on-chip "shared" memory accessible to all threads executing on the same multiprocessor or compute unit. GPU shared memory provides a high performance mechanism for inter-thread communication needed for use as a software-managed cache. One of the key challenges involved in developing high-performance GPU-accelerated algorithms is making most effective use of fast on-chip memory systems on the GPU.

The future outlook for GPU hardware is bright. Vendor roadmaps project continuing increases in arithmetic throughput and memory bandwidth for at least three more processor generations, based on increased circuit density enabled by semiconductor fabrication process improvements leading to increasing numbers of cores, arithmetic units, and so on. If the evolution of future GPU designs follows past trends, arithmetic throughput could increase by a factor of three or more and memory bandwidth might increase by a factor of two over the next two GPU generations. These expected changes would provide the greatest benefit to applications that are arithmetic bound on current generation GPUs and that have a surplus of parallelism that can be exploited by future hardware. Even if future GPU hardware designs begin to diverge significantly from the GPUs of today, the algorithms designed for contemporary GPUs have been shown to be relatively easy to adapt to other highly parallel architectures and multi-core CPUs, with libraries, automatic translators, and even by hand. Recent CPU designs such as AMD Fusion and Intel Sandy Bridge have recently begun to incorporate GPUs on a small scale. Hybrid CPU/GPU processors cannot match the peak arithmetic performance and memory bandwidth of traditional discrete GPUs, but they have the potential to offer performance levels beyond those available in traditional CPUs and may be worth utilizing, specifically for algorithms that lack sufficient parallelism to fully utilize high-end GPUs.

3.2 GPU software interface and programming model

The two leading GPU software interfaces at the time of writing are NVIDIA's CUDA programming toolkit [19], and OpenCL [20,21], an industry standard heterogeneous computing API. The key programming abstractions provided by CUDA and OpenCL are very similar. Both APIs provide interfaces for allocating GPU memory, copying data between the host and GPU, defining functions (so-called "kernels") that can be executed on the GPU, launching GPU kernels, and checking for errors. CUDA and OpenCL both provide mechanisms for defining data-parallel computations in terms of multidimensional index spaces composed of tens of thousands of individual work items that can be computed in parallel. Work items are mapped onto GPU hardware threads that are executed in groups on the underlying multiprocessors or compute units, with smaller subsets of threads known as "warps" or "wavefronts" being executed in lock-step on the GPU's SIMD arithmetic hardware. Multiple warps or wavefronts executing on the same multiprocessor or compute unit are grouped together into so-called ``thread blocks'' or ``work groups'' that have access to on-chip shared memory and L1 cache, coordinating with each other through high-performance synchronization primitives. The mapping of work items to hardware threads and underlying arithmetic units has a significant effect on the resulting performance, particularly as it relates to GPU memory access patterns. The best choice of parallel decomposition and hardware mapping is inherently device specific. Today, applications must often be manually tuned for the target GPU architecture. Performance-portability of GPU-accelerated applications currently remains an ongoing research problem in the community.

4. Approach

In order to design the GPU-based version of the Savant SBR algorithm, the steps in the Savant SBR algorithm shown in Section 2 were analyzed to determine which steps were computational bottlenecks, and which steps were well-suited for throughput oriented

data-parallel implementation on the GPU. As discussed in Section 2, the SBR algorithm consists of two primary tasks: creation of multi-bounce ray tracks and computation of the radiation integral at each bounce. Of these, the radiation integral was observed to be the dominant cost of the algorithm, especially as the simulation parameters were scaled up to produce higher fidelity results. Increasing the simulation fidelity involves increasing the number of rays, N_r, or increasing the number of observation angles (or points), N_o, or increasing the number of frequency steps, N_f. Increasing any of these three values increases the computational cost of the radiation integral, either by increasing the cost at a single bounce (N_o or N_f) or increasing the number of ray bounces where the radiation integral must be performed (N_r). However, the ray tracing cost only increases with N_r. As a result, as a Savant simulation is scaled up to produce higher fidelity results, the cost of the radiation integral dominates the computational cost when generating high fidelity results.

For the second requirement for GPU computation, the computational structure of the ray tracing calculations and the radiation integral calculations were compared. Raytracing requires many unstructured memory accesses, as the rays propagate through the region around the platform and ray-surface intersection calculations are performed to determine the reflection and transmission points. Raytracing acceleration data structures, such as binary space partitioning (BSP) trees and k-d trees, are commonly used to reduce the number of ray-surface intersection calculations. But these structures increase the number of unstructured memory accesses and the tree traversal algorithms tend to be implemented using recursive functions, which are only supported by the latest GPU hardware. In addition, it is difficult to predict, a priori, how many times a given ray will reflect or transmit off the platform, leading to a wide variation in the computational cost of each ray. The unstructured memory accesses, variable length multi-bounce rays, and recursive nature of many acceleration algorithms make ray tracing a challenging algorithm to develop on the GPU.

The radiation integral for each ray bounce has a uniform computational structure. The radiation integral calculation can be structured so that the same calculations are performed at each observation angle (or point), at each frequency step, for each bounce. The memory access pattern resulting from this design is regularly-structured, involving iteration over lists of observations, frequencies, and ray bounce data. In other words, the radiation integral is a data-parallel algorithm with the right computational structure to be considered for the GPU.

Based on this analysis, a hybrid parallelization was developed for Savant, where the ray tracing calculations would be performed using Savant's existing task-based parallelization for multi-core CPUs and the radiation integral would be performed using data-parallel computation on the GPU. With this approach, each processor type (CPU or GPU) is given the task that is well suited for its architecture, and the entire Savant simulation is parallelized across all available hardware in the system.

After identifying the radiation integral as the appropriate candidate for GPU parallelization, we began to design the GPU version of the algorithm. Development began with the radiation integral derivation and the CPU implementation in C++ and led to the GPU implementation in CUDA C. The main focus of the effort was in redesigning the layout of the data structures used by the code. The object-oriented structure of C++,

where data and methods are associated together for each object, is well suited for a large-scale software project. However, this is not the best organization for data-parallel algorithms on the GPU. In C++, an object typically contains multiple fields of primitive data (floating point numbers, integers, *etc.*), and multiple objects are stored in arrays, in a so-called *array of structures* organization. For the GPU, arrays of C++ objects are flattened into many separate arrays of primitive data, known as a *structure of arrays* organization. These arrays are then passed to the GPU kernel functions. In CUDA, the kernel contains the operations and calculations to be performed by all threads on the GPU, but it is written in terms of the operations performed by a single thread.

Three distinct GPU kernels were developed to implement the radiation integral on the GPU. The first kernel computes the projection of the ray tube footprint in the direction of the reflected ray, a required input for the second kernel. The second kernel computes the surface currents at the ray footprint, as described in Step 4 of the SBR algorithm. The third kernel calculates the radiation integral for all observations, as described in Step 5. The first and last kernels have separate implementations for far-field angles and near-field points. The second kernel is common to both.

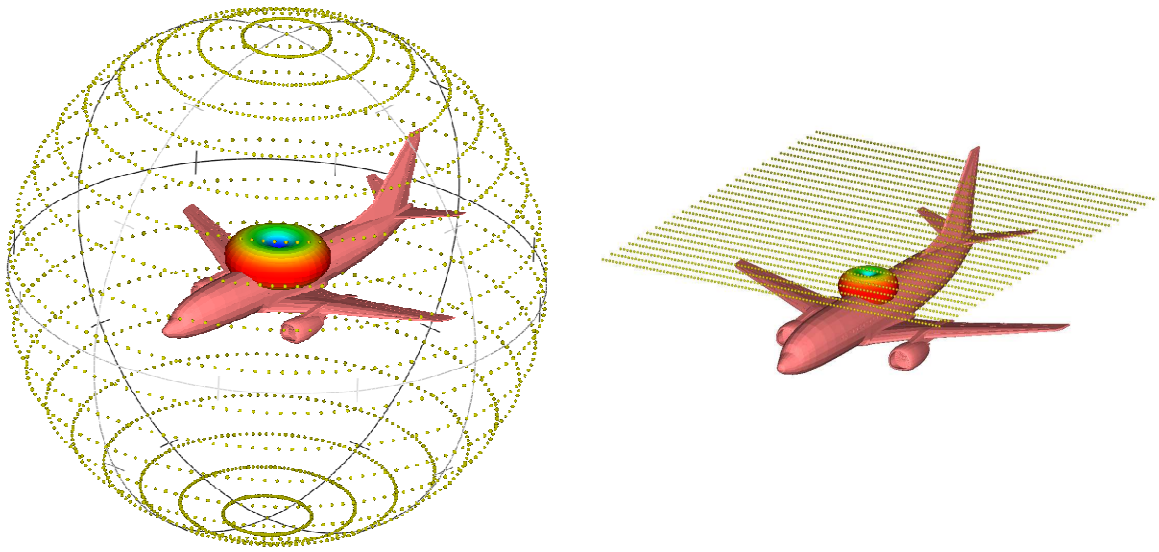
Initial development of the GPU kernels was completed in the early stages of the project. At that point the focus of the development effort shifted from implementation correctness to performance tuning. Many experiments were conducted with different thread grid layouts in order to find a GPU thread grid design that performed well across a wide range of problem sizes. Data transfer costs between the CPU and GPU were measured and the communication over the PCI express bus was redesigned in order to minimize the costs. The GPU kernel functions were modified to reduce GPU register usage, improve memory access patterns and utilize GPU hardware mathematic functions. The software interface between the CPU and GPU was redesigned to improve the load balancing of the hybrid parallelization across the multiple CPU cores and multiple GPUs. These tuning efforts led to significant improvements in computational performance, as discussed in the next section.

5. Results

We conducted a series of experiments to demonstrate the acceleration potential of Savant GPU for the computation of both far-field radiation patterns and near-field field distributions in a variety of problem sizes. The base configuration is the same for all experiments: a short dipole antenna mounted on top of a Boeing 737 aircraft as shown in Figure 1. The antenna is simulated in a frequency range of 3 GHz to 4 GHz, with the single-frequency runs at 4 GHz. The aircraft is modeled by a triangle mesh containing 18,370 triangles and 55,110 vertices. At 4 GHz, the aircraft is 413 wavelengths long and has a wingspan of 391 wavelengths.

For both the far-field and the near-field cases, analysis is performed at five different levels of observations angles (or points), from 2500 to 40000, with the number of observations doubled in each level. Three different numbers of frequency steps are tested for each observation level: one, five and 25. This created 15 different experiments for far-field radiation patterns, and 15 different experiments for near-field field distribution.

Each set of experiments is solved using four different computational configurations: one CPU core, four CPU cores, one CPU core plus one GPU, and finally four CPU cores and two GPUs. The CPU was an AMD Phenom II x4 965 running at 3.4 GHz. The GPUs are NVIDIA GTX 480 based on the NVIDIA Fermi GPU running at 1.4 GHz. Wall clock times are provided for each compute configuration and CPU utilization is provided. The wall clock and CPU utilization are measured using the system tool *time* based on the entire run of the simulation, including initialization of the simulation and file input and output. Speedups for the three parallel processing configurations are provided relative to the configurations that precede them and are all based on total simulation run time. The baseline is the one-CPU-core case on the left, and the fastest is the four-CPU-core, two-GPU case on the right.



(a) (b)

Figure 1: Test case showing the Boeing 737 platform with a short electric dipole antenna mounted on top of the fuselage. The far-field observation domain is shown in a). Near-field observation domain is shown in b).

Performance data for the far-field radiation pattern and near-field field distribution experiments are shown in Tables 1 and 2. While the specific performance numbers are not exactly the same for the two sets of experiments, many of the trends are present in both. These trends are discussed in the remainder of this section.

Far Field Boeing 737 Short Electric Dipole 3-4GHz # Rays: 747820			1 CPU core (a)	4 CPU cores (b)			1 CPU core, 1 GPU (c)			4 CPU cores, 2 GPUs (d)				
Run #	# Obv. Angles	# Freq Steps	Wall Time (sec)	CPU Util. %	Wall Time (sec)	Speedup vs (a)	CPU Util. %	Wall Time (sec)	Speedup vs (b)	CPU Util. %	Wall Time (sec)	Speedup vs (a)	Speedup vs (b)	Speedup vs (c)
1	2500	1	455.9	393%	118.6	**3.8**	100%	19.3	**23.6**	342%	8.4	**54.3**	**14.1**	**2.3**
2	5000	1	897.8	394%	230.4	**3.9**	99%	19.5	**46.1**	335%	9.2	**97.8**	**25.1**	**2.1**
3	10000	1	1777.3	395%	453.9	**3.9**	99%	20.9	**85.0**	346%	9.8	**182.1**	**46.5**	**2.1**
4	20000	1	3536.9	394%	914.9	**3.9**	99%	26.6	**133.1**	344%	12.0	**295.5**	**76.4**	**2.2**
5	40000	1	7062.0	394%	1860.0	**3.8**	99%	41.1	**171.7**	322%	18.6	**379.3**	**99.9**	**2.2**
6	2500	5	1080.1	395%	279.6	**3.9**	96%	25.2	**42.9**	346%	11.3	**95.6**	**24.7**	**2.2**
7	5000	5	2141.2	395%	551.3	**3.9**	98%	32.8	**65.3**	323%	15.1	**141.4**	**36.4**	**2.2**
8	10000	5	4271.0	395%	1119.3	**3.8**	99%	73.4	**58.2**	288%	34.6	**123.3**	**32.3**	**2.1**
9	20000	5	8545.0	394%	2232.3	**3.8**	98%	117.4	**72.8**	250%	59.8	**142.9**	**37.3**	**2.0**
10	40000	5	17295.0	395%	4474.0	**3.9**	99%	217.3	**79.6**	229%	120.0	**144.2**	**37.3**	**1.8**
11	2500	25	2738.8	392%	728.3	**3.8**	98%	56.1	**48.8**	307%	26.2	**104.6**	**27.8**	**2.1**
12	5000	25	5446.0	393%	1460.0	**3.7**	99%	96.1	**56.7**	268%	47.4	**114.9**	**30.8**	**2.0**
13	10000	25	11053.0	394%	2899.6	**3.8**	98%	181.2	**61.0**	236%	91.6	**120.7**	**31.7**	**2.0**
14	20000	25	22055.0	394%	5952.0	**3.7**	98%	329.2	**67.0**	223%	172.3	**128.0**	**34.5**	**1.9**
15	40000	25	44279.0	394%	11554.0	**3.8**	99%	639.9	**69.2**	209%	374.2	**118.3**	**30.9**	**1.7**

Table 1: Comparison of Savant run-time performance when computing a far-field radiation pattern using four different computational configurations on 15 different problem sizes.

Near Field Boeing 737 3-4GHz # Rays: 747820			1 CPU core (a)	4 CPU cores (b)			1 CPU core, 1 GPU (c)			4 CPU cores, 2 GPUs (d)				
Run #	# Obv. Points	# Freq Steps	Wall Time (sec)	CPU Util. %	Wall Time (sec)	Speedup vs (a)	CPU Util. %	Wall Time (sec)	Speedup vs (b)	CPU Util. %	Wall Time (sec)	Speedup vs (a)	Speedup vs (b)	Speedup vs (c)
1	2500	1	652.5	393%	167.5	**3.9**	97%	20.1	**32.5**	341%	8.8	**74.2**	**19.1**	**2.3**
2	5000	1	1284.9	394%	329.3	**3.9**	99%	19.6	**65.5**	347%	9.3	**138.2**	**35.4**	**2.1**
3	10000	1	2559.6	394%	650.4	**3.9**	99%	24.4	**104.9**	327%	12.1	**212.1**	**53.9**	**2.0**
4	20000	1	5067.0	394%	1320.8	**3.8**	99%	35.3	**143.6**	330%	16.1	**315.5**	**82.2**	**2.2**
5	40000	1	10165.0	395%	2627.5	**3.9**	99%	58.6	**173.4**	297%	27.7	**366.7**	**94.8**	**2.1**
6	2500	5	1521.0	393%	392.0	**3.9**	99%	30.8	**49.4**	339%	14.0	**108.3**	**27.9**	**2.2**
7	5000	5	3027.4	392%	825.3	**3.7**	99%	44.8	**67.6**	314%	20.8	**145.5**	**39.7**	**2.2**
8	10000	5	6048.0	393%	1658.2	**3.6**	99%	73.9	**81.8**	282%	35.5	**170.4**	**46.7**	**2.1**
9	20000	5	12440.0	393%	3386.0	**3.7**	99%	133.6	**93.1**	250%	66.7	**186.6**	**50.8**	**2.0**
10	40000	5	24997.0	394%	6617.0	**3.8**	99%	253.0	**98.8**	223%	132.3	**189.0**	**50.0**	**1.9**
11	2500	25	4458.0	391%	1337.8	**3.3**	99%	108.3	**41.2**	264%	53.1	**84.0**	**25.2**	**2.0**
12	5000	25	9240.0	391%	2673.5	**3.5**	99%	202.4	**45.7**	234%	102.6	**90.1**	**26.1**	**2.0**
13	10000	25	18830.0	393%	5476.0	**3.4**	99%	386.7	**48.7**	215%	198.6	**94.8**	**27.6**	**1.9**
14	20000	25	36705.0	392%	11214.0	**3.3**	99%	769.0	**47.7**	207%	416.7	**88.1**	**26.9**	**1.8**
15	40000	25	73396.0	391%	21382.0	**3.4**	99%	1532.8	**47.9**	201%	861.9	**85.2**	**24.8**	**1.8**

Table 2: Comparison of Savant run-time performance when computing a near-field field distribution using four different computational configurations on 15 different problem sizes.

The four-core-CPU configuration (column set (b) in both of the tables) consistently shows a 3.8 to 3.9 times speedup over the one-core-CPU case. This demonstrates that the multi-core parallelization within Savant scales efficiently for a variety of problem sizes on a four-core CPU. It also serves as a baseline for the final configuration to demonstrate the benefits of adding multiple GPUs to a four-core CPU. The speedup is less than the ideal of 4.0 because the initialization and file output is not parallelized and there is a small amount of thread synchronization and coordination overhead when running on multiple cores.

Performance data for the one-core-CPU, one-GPU case is shown in column (c). For each experiment the GPU provides a dramatic speedup, from 24x to 171x for far-field and 32x to 173x for near-field. The benefit of the GPU increases as the size of the problem increases because, as the problem size increases, the radiation integral becomes the more dominant computational cost. For the smallest problem sizes, such as those found in runs 1, 2 and 3, the run time is constant, even though the problem size increases by a factor of four. Recall that the GPU and CPU are computing different parts of the problem in parallel. For each of these cases, the CPU performs the same amount of work to shoot rays at the platform, but the GPU workload is doubling. For the small problem sizes, the GPU completes the radiation integral before the CPU generates the next set of ray bounces, resulting in idle time for the GPU. As the cost of the radiation integral increases, the GPU workload increases and eventually, when the GPU utilization has been maximized, the run times double when the problem size doubles.

The multi-frequency cases also benefit from GPU acceleration, but the degree of impact is less than that of the single-frequency case. The main reason for this is that the CPU is able to exploit a more efficient iterative formulation for the frequency-dependent aspect of the radiation integral that requires fewer sin() and cos() function evaluations per frequency step. Because the GPU computes all the frequency steps in parallel, it is not able to take advantage of the iterative algorithm. Because the CPU is roughly two to three times faster per frequency step when computing a linear sequence of frequency steps than when the CPU computes individual frequencies, and the GPU speedup is constant in either case, the relative improvement of the GPU verses the CPU is reduced for the multi-frequency cases.

The final configuration is the four-CPU-core, two-GPU case shown in column (d). This is the optimal configuration for this machine where Savant utilizes all computational processors available in the system. The results show that adding the second GPU doubles the speedup in almost all of the cases. Some of the smaller cases actually see a speedup that is more than double. These are the cases that are CPU-bound in the single-GPU tests and see a benefit from the extra CPU cores. The CPU utilization for all of the two-GPU runs is less than 400%, typically 200% to 300%. This is an indication that the simulation is GPU-bound, and that, for this simulation configuration on this hardware, only two or three CPU cores are required to reach a high bounce-generation rate to supply bounces to the two GPUs.

Overall, these results show that there is a significant benefit for each level of parallelization in Savant, with significant performance gains through the use of one or more GPUs. At the same time, the data show that the degree of speedup is variable and depends on many factors, such as number of observations, number of frequency steps and

the relative computational cost of ray tracing on the CPU verses the cost of the radiation integral on the GPU. For these reasons, it is very difficult to predict *a priori* the exact level of performance improvement that will be achieved for a particular Savant simulation.

6. Conclusion

In this paper we discussed the architectural differences between CPU and GPU designs, and demonstrated how those differences could be exploited to produce a hybrid CPU-GPU parallelization of the Savant asymptotic CEM solver. Using a conventional desktop workstation with two GPUs, we were able to show 50x to 380x speedups over a single CPU core. While the level of speedup shown in the previous section is significant in its own right, it becomes even more significant when the cost of the hardware is taken into account. Adding the two GPU cards to the system roughly doubled the total cost of the hardware and roughly doubled the power consumption and heat dissipation. Yet the computational capacity of this GPU-based machine is equivalent to a cluster of a minimum of 14 quad-core machines and, for certain problems, a maximum of at least 100 quad-core machines. Clearly GPU parallelization offers a significant savings in computation time as well as a significant reduction in capital and operational costs for computing hardware, power and cooling budgets.

The work presented in this paper represents the progress made during the first 18 months of a SBIR project with Dr. John Asvestas in the Radar and Antenna Systems Division of the Naval Air Systems Command (NAVAIR). The project will continue for another 18 months. For the next phase in the project, we plan to support additional computation algorithms on the GPU, such as the creeping wave radiation integral and GPU-based ray tracing for triangle mesh and NURBS surfaces. We also plan to develop a dynamic load-balancing MPI-based cluster algorithm to enable execution of Savant on clusters of GPU-enabled nodes.

Acknowledgement

The authors gratefully acknowledge the support of U.S. Naval Air Systems Command (NAVAIR) in funding the development of Savant and its acceleration with GPUs.

References:

[1] R. G. Kouyoumjian, “The geometrical theory of diffraction and its application,” in *Numerical and Asymptotic Techniques in Electromagnetics*, R. Mittra, Ed., Springer-Verlag, 1975, Ch. 6.

[2] P. Y. Ufimtsev, *Theory of Edge Diffraction in Electromagnetics*, Tech. Science Press, Encino, CA, 2003.

[3] H. Ling, R. C. Chou, and S. W. Lee, “Shooting and bouncing rays: calculating the RCS of an arbitrarily shaped cavity,” *IEEE Trans. Antennas Propagat.*, vol. 37, pp. 194-205, Feb. 1989.

[4] J. Baldauf, S. W. Lee, L. Lin, S. K. Jeng, S. M. Scarborough and C. L. Yu, “High-frequency scattering from trihedral corner reflectors and other benchmark targets: SBR vs. experiments,” *IEEE Trans. Antennas Propagat.*, vol. 39, pp. 1345-1351, Sep. 1991.

[5] D. J. Andersh, M. Hazlett, S. W. Lee, D. D. Reeves, D. P. Sullivan, and Y. Chu, "XPATCH: A high frequency electromagnetic-scattering prediction code and environment for complex 3D objects," *IEEE Antennas Propagat. Mag.*, vol. 36, pp. 65-69, Feb. 1994.

[6] S. W. Lee and R. Chou, "A versatile reflector antenna pattern computation method: shooting and bouncing rays," *Microwave and Optical Tech. Letters*, vol. 1, no. 3, pp. 81-87, May 1988.

[7] T. K. Wu, R. A. Kipp, and S. W. Lee, "Field of view of a spacecraft antenna: analysis and software," *NASA Tech Briefs Journal*, vol. 19, no. 11, Nov. 1995.

[8] T. Ozdemir, M. W. Nurnberger, J. L. Volakis, R. Kipp, and J. Berrier, "A hybridization of finite-element and high-frequency methods of pattern prediction for antennas on aircraft structures," *IEEE. Antennas Propagat. Mag.*, vol. 38, pp. 28 – 38, June 1996.

[9] G, A. Deschamps, "Ray techniques in electromagnetics," *Proc. IEEE*, vol. 60. no. 9, Sep. 1972, pp. 1022-1035.

[10] R. A. Kipp, "Curved surface scattering geometry in the shooting and bouncing rays method," *2010 IEEE Antennas Propagat. Intl. Symp.*, Toronto, ON.

[11] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A.E. Lefohn, and T. Purcell, "A survey of general-purpose computation on graphics hardware", *Computer Graphics Forum*, vol. 26, no. 1, pp. 80-113, 2007.

[12] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips, "GPU computing", *Proceedings of the IEEE*, vol. 96, pp. 879-899, 2008.

[13] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture", *IEEE Micro*, vol. 28, no. 2, pp. 39-55, 2008.

[14] NVIDIA, Whitepaper: "NVIDIA's next generation CUDA Compute Architecture: Fermi", 2009.

[15] D.A. Patterson, "The top 10 innovations in the new NVIDIA Fermi architecture, and the top 3 next challenges", Sept. 2009.

[16] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel computing experiences with CUDA", *IEEE Micro*, vol. 28, no. 4, pp. 13-27, 2008.

[17] J.E. Stone, J. Saam, D.J. Hardy, K.L. Vandivort, W.W. Hwu, and K. Schulten, "High Performance Computation and Interactive Display of Molecular Orbitals on GPUs and Multi-core CPUs", *ACM International Conference Proceeding Series*, vol. 383, pp. 9-18, 2009.

[18] J.E. Stone, D.J. Hardy, I.S. Ufimtsev, and K. Schulten, "GPU-Accelerated Molecular Modeling Coming Of Age", *J. Molecular Graphics and Modelling*, vol. 29, no. 2, pp. 116-125, 2010.

[19] J. Nickolls, I. Buck, M. Garland, K. Skadron, "Scalable parallel programming with CUDA", *ACM Queue*, vol. 6, pp. 40-53, 2008.

[20] A. Munshi, "OpenCL Specification Version 1.0", 2008.

[21] J.E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems", *Computing in Science and Engineering*, vol. 12, no. 3, pp. 66-73, 2010.