Using GPUs for Accelerating Electromagnetic Simulations

Manuel Ujaldon

Department of Computer Architecture
University of Malaga, Malaga 29071, Spain
ujaldon@uma.es

Abstract- The computational power and memory bandwidth of graphics processing units (GPUs) have turned them into attractive platforms for general-purpose applications at significant speed gains versus their CPU counterparts [1]. In addition, an increasing number of today's state-of-the-art supercomputers include commodity GPUs to bring us unprecedented levels of performance in terms of raw GFLOPS and GFLOPS/cost. Inspired by the latest trends and developments in GPUs, we propose a new paradigm for implementing on GPUs some of the major aspects of electromagnetic simulations, a domain traditionally used as a benchmark to run codes in some of the most expensive and powerful supercomputers worldwide. After reviewing related achievements and ongoing projects, we provide a guideline to exploit SIMD parallelism and high memory bandwidth using the CUDA programming model and hardware architecture offered by Nvidia graphics cards at an affordable cost. As a result, performance gains of several orders of magnitude can be attained versus thread-level methods like pthreads used to run those simulations on emerging multicore architectures

Index Terms - Graphics processors, electro-magnetic simulations, CUDA, GPGPU.

I. INTRODUCTION

Graphics processors are usually characterized by parallelism, pipelining and bandwidth. After completing a steady transition from mainframes to workstations to PC cards, Graphics Processing Units (GPUs) emerge nowadays like a solid and compelling alternative to traditional computing, delivering extremely high floating point performance for those applications which can be arranged to fit and exploit the inherent parallelism and high memory bandwidth [2]. The newest versions of programmable graphics processing units (GPUs) have consistently demonstrated an outstanding performance in many applications beyond graphics, including data mining [3,4], computer vision [5], signal and image processing and segmentation [6,7,8], numerical methods [9], and assorted simulations [10,11,12].

This fact has attracted many other researchers and encouraged the use of GPUs in a broader range of applications, where developers will need to leverage this technology with new programming models which ease the developer's task of writing programs to run efficiently on GPUs. Nvidia and ATI/AMD, manufacturers of the popular GeForce and Radeon sagas of graphics cards, have released software components which provide simpler access to GPU computing power than that realized by treating the GPU as a traditional graphics processor. CUDA (Compute Unified Device Architecture) [13] is Nvidia's solution as a simple block-based API for programming; AMD's alternative is called Stream Computing and includes technologies such as the Brook+ compiler [14] and the Compute Abstraction Layer, both of which allow the developer to work in a high-level language which abstracts away GPUs' specifics. Those companies have also developed hardware products aimed specifically at the General Purpose GPU (GPGPU) computing market: The Tesla products [15] are from Nvidia, and Firestream [16] is AMD's product line.

Between Stream Computing and CUDA, we chose the latter to program the GPU for being more popular and providing more mechanisms to optimize general-purpose applications which do not entirely fit into the more traditional graphics processing paradigm. More recently, Apple's OpenCL framework [17] emerges as an attempt to unify those two models with a superset of features, but since it is closer to CUDA and inherits most of its mechanisms, we are confident on an eventual portability for the methods described throughout this paper without loss of generality.

Novel scientific applications are good candidates to take the opportunity offered by CUDA and counterparts (see Fig. 1), and electromagnetic simulations is clearly one of them for three primary reasons:

1. This field has traditionally proven to be of great success for GPUs during its evolution towards high-performance general-purpose computing.
2. The increasing complexity of recent electromagnetic algorithms has made simulation part of the workflow in both academia and industry to be very computationally demanding.
3. Traditional architectures reveal themselves as inefficient solutions for this class of applications.

Electromagnetic simulations are memory intensive applications containing assorted access patterns where memory optimizations play a primary role. Fortunately, CUDA provides a set of powerful low-level mechanisms for controlling the use of memory and the behavior of its hierarchy. This affects performance severely at the expense of a considerable programming effort, which we describe throughout this paper.

The rest of the paper is organized as follows. Section II reviews the most recent results obtained by GPUs on electromagnetic simulations. Section III focuses on the specifics of the GPU programming with CUDA, and Section IV describes optimization strategies particularly oriented to simulation codes. Section V concludes.

II. THE GPU ON ELECTROMAGNETIC SIMULATIONS

A. Related Work

Over the past few decades, the increase of overall computing power coupled with the maturation of many electromagnetic algorithms has produced a blooming on the simulation side. Many explorations focused on 2D first, were later extended to 3D, and even were modeled as so-called 2.5D problems.

In response to that evolution, a number of approaches to hardware acceleration of electromagnetic simulations have been investigated in the past five years. Those approaches can be classified into two main categories:

1. Stand-alone computing devices like ASICS, which represent the highest achievable acceleration but quickly becomes too expensive due to the massive hardware required.
2. Co-processors with their own memory and connected to a host PC via an input/output bus or socket interface. Within this category, we may find Field Programmable Gate Arrays (FPGAs) [18] and Graphics Processing Units (GPUs) [19].

GPUs stand out in a unique way from all these innovative solutions because they are produced as commodity processors and their floating point performance has significantly outpaced that of any other processor. In addition, GPUs have become easier to program, which allows developers to effectively exploit their computational power.

Modern GPUs have been at the leading edge of increasing chip-level parallelism over the past five years. Scaling from 8 to 240 processors in the most popular saga of Nvidia GPUs, they have completed a steady transition from multi-core to many-core processors. The high degree of parallelism achieved, combined with their wide availability and affordable budget, has ultimately confirmed GPUs as a popular platform among universities and students to run computationally expensive simulations [1].

More recently, several companies that supply leading edge electromagnetic simulation software have joined this movement to ease code transition to the GPU for all kind of users belonging to this area regardless of their programming skills. Some illustrative examples are Acceleware and CST, which have announced a new GPU-based solution for accelerating lengthy electromagnetic design simulations, reporting performance gains of up to 40% compared to previous products [20,21]. This software uses CUDA, a programming interface particularly designed to solve complex computational general-purpose problems, which we describe later in Section III. Large corporations and research institutions have also been able to tap into clusters of GPUs for large scale simulations [22], enabling a step forward in performance while maintaining a limited budget. This way, the GPU technology aspires to have a tremendous impact on engineering electromagnetic education, as universities and research centers worldwide will be able to simulate realistic problems with affordable GPU-based hardware platforms, which will also be available to students on their own personal computers.

Successful implementations of electro-magnetic algorithms on GPUs can be seen as the key for the integration of simulators into design and optimization tools [23]. The GPU power may be combined here with the development of behavioral models and multi-grid, graded mesh and multi-resolution techniques for boosting the performance of electromagnetic simulations.

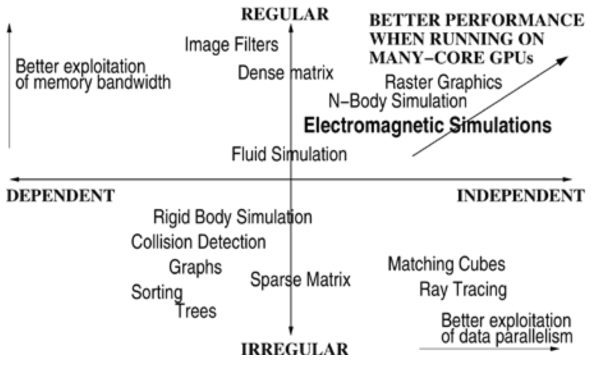

Fig. 1. An overview of general purpose applications evaluated by GPU performance according to two major features: Amount of parallelism extracted (on X axis) and memory bandwidth exploitation (on Y axis).

B. Characterization

The GPU has been extensively used in scientific computing over the past five years, but the degree of success has been different depending on algorithm features and how they meet GPU hardware idiosyncrasies. Nvidia [13,24] has reported a list of illustrative examples. Just to mention a few involving simulations, we have: molecular dynamics (36x), fluid dynamics (17x), multi-fluid (50x), astrophysics (100x), multi-body mechanical (13x), financial (149x), oil and gas (18x), DNA and liquids (18x), and interactive visualization of volumes (146x).

In general, expectations for a particular algorithm to reach certain levels of speedup factor when running on GPUs depend on a number of features which conform a list of requirements to be fulfilled. From less to more important, we have:

1. Small local data requirements (memory and registers).
2. Stream computing (non-recursive algorithms).
3. Arithmetic intensity (high data reuse).
4. Bandwidth (fast data movement).
5. Data parallelism (data independency).

The two key factors are analyzed in Fig. 1, where some of the most popular applications are placed in conjunction with electromagnetic simulations to quantify the memory bandwidth and data parallelism each algorithm can benefit from. This gives us an estimation about how successfully each code can run on GPU platforms.

C. Upsides

Simulations usually consists of a mixture of fundamentally serial control logic and inherently parallel computation. Furthermore, those computations are often data-parallel in nature, which matches the programming model that CUDA adopts (see Section III-B), basically a sequential control thread capable of launching a series of parallel kernels. This makes it relatively easy to parallelize an application's individual components as kernels, rather than requiring a wholesale rewriting of the entire application.

In our case of a typical electromagnetic simulation, the same executable is invoked multiple times on each parallel processor by a job-queuing algorithm and the results are then reassembled. This constitutes an embarrassingly parallel computing model, as it does not require much internode communication or global data sharing. Electromagnetic computations are in fact very close to graphics processing in this respect: Million of operations can be performed in parallel exhibiting a speed which can reach up to two orders of magnitude when compared to the computational power shown on typical quad-core CPUs.

On the other hand, simulations often deal with a large amount of data, which are responsible for the realism and accuracy of the simulated physics. GPUs reach data bandwidth with video memory around ten times higher than CPUs with main memory, and because of the way data is transferred, regular access patterns in the code behave better when running on GPUs.

A third issue is also worth mentioning: Arithmetic intensity. Electromagnetic simulations usually require the computation of complex mathematical formulas, which are efficiently mapped to the GPU platform due to the presence of those units devoted to a typical graphics rendering. Moreover, newer generations of GPUs like GeForce Series 8 include internally a powerful co-processor devoted to the computational physics required in many realistic animation and effects. Such co-processor, called PhysX [25], was originally invented by Ageia, whose design was inspired in those we found in GPUs for building arithmetic units and massively parallelism.

Finally, we leave on the CPU those parts of our simulation that do not have high arithmetic intensity or do not expose substantial amounts of data or thread-level parallelism. This way, that tough part of our application remains unchanged and can benefit from overlapping computations on a bi-processor CPU-GPU platform.

D. Downsides

For the GPU to succeed as the favourite platform to run electromagnetic simulations in the future, we still envision two main challenges in the horizon: Accuracy and memory capacity.

Accuracy. The lack of 32-bit floating-point precision was a major drawback in many application areas during the first half of this decade. Starting in 2008 with the GT 200 series from Nvidia, the situation has reversed and all major GPU vendors now offer 64-bit massively parallel hardware which will further enhance modelling and simulation capabilities. For example, the Tesla T10P GPU from Nvidia provides full IEEE rounding, fused multiply-add, and denormalized number support for double precision.

The problem arises when you look at execution times, since in most cases performance drops from five to ten times when you migrate your algorithm from single to double precision. This is mainly due to the reduced degree of parallelism we can exploit in the architecture, as usually the ratio of single to double precision floating-point arithmetic units available in a typical GPU is four to one or even eight to one. In the past, the primary argument for not to overcome this lack was that classical rendering did not require such enhancement. With the recent movements towards general-purpose GPU-like architectures, double precision floating-point will be offered at a much lighter performance penalty as more applications demand it.

E. Memory size

Some of the large scale simulations are not necessary complicated in nature, but they require a large amount of memory space. For example, modelling of the near electromagnetic fields around antennas fall into this category, and more in general, field and signal analysis for high-speed electronic circuits and systems has become increasingly difficult due to the complexity of new electronic devices. GPU memory has progressed at a higher speed rate than the CPU counterpart over the last decade, and GDDR5, the video memory currently available, keeps consistently two generations ahead versus CPU DDR3 memory placed on the mainboard. But when it comes to capacity, the reduced form factor (size) of the graphics card in conjunction with its wider bus width versus the GPU, introduce serious routing problems which prevent video memory capacity from growing at the same rate. We believe that the solution to this problem lies more in the software layer, particularly in programmer's hands, who has to be able to partition data efficiently and ultimately perform computations through a blocking strategy to overcome memory constraints.

III. CUDA

The Compute Unified Device Architecture (CUDA) [13] is a programming interface and set of supported hardware to enable general-purpose computation on Nvidia GPUs.

The CUDA programming interface is ANSI C extended by several keywords and constructs which derive into a set of C language library functions as a specific compiler generates the executable code for the GPU in conjunction with the counterpart version running on the CPU acting as a host.

Since CUDA is particularly designed for generic computing, it can leverage special hardware features not visible to more traditional graphics-based GPU programming, such as small cache memories, explicit massive parallelism and lightweight context switch between threads.

A. Hardware Platforms

All the latest Nvidia developments on graphics hardware are compliant with CUDA: For low-end users and gamers, we have the GeForce series starting from its 8th generation; for high-end users and professionals, the Quadro FX 5600/4600 series; for general-purpose computing, the Tesla boards. Focusing on Tesla, the C870 is an homogeneous CMP endowed with 128 cores and 1.5 GB of video memory to deliver a theoretical peak performance of 518 GFLOPS (single precision), a peak on-board memory bandwidth of 76.8 GB/s and a peak main memory bandwidth of 4 GB/s under its PCI-express x16 interface.

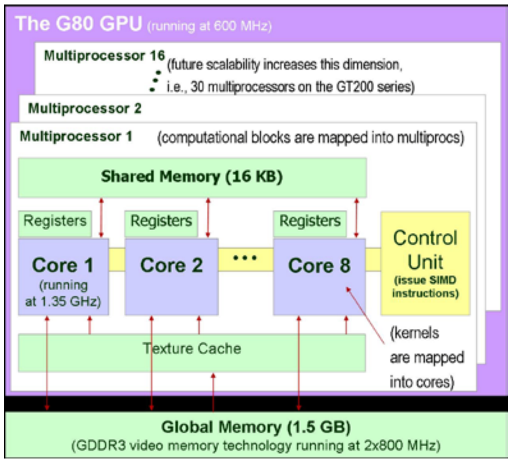
The G80 GPU (running at 600 MHz)
Multiprocessor 16 (future scalability increases this dimension, i.e., 30 multiprocessors on the GT200 series)
Multiprocessor 2
Multiprocessor 1 (computational blocks are mapped into multiprocs)
Shared Memory (16 KB)
Registers
Registers
Registers
Core 1 (running at 1.35 GHz)
Core 2
Core 8
Control Unit (issue SIMD instructions)
Texture Cache
(kernels are mapped into cores)
Global Memory (1.5 GB)
(GDDR3 video memory technology running at 2x800 MHz)

Fig. 2. The CUDA hardware interface.

B. Execution Modes

The G80 parallel architecture is a SIMD (Single Instruction Multiple Data) processor endowed with 128 cores. Cores are organized into 16 multiprocessors, each having a large set of 8192 registers, a 16 KB shared memory very close to registers in speed (both 32 bits wide), and constants and texture caches of a few kilobytes. Each multiprocessor can run a variable number of threads, and the local resources are divided among them. In any given cycle, each core in a multiprocessor executes the same instruction on different data based on its *threadID*, and communication between multiprocessors is performed through global memory (see Fig. 3).

Future architectures from Nvidia will support the same CUDA executables, but they will be run faster in order to include more multiprocessors per die, or more cores, registers or shared memory per multiprocessor. For example, the GT200 architecture contains 30 multiprocessors for a total of 240 cores, while registers and shared memory per multiprocessor remain the same.

The CUDA programming model guides the programmer to expose fine-grained parallelism as required by massively multi-threaded GPUs, while at the same time providing scalability across the broad spectrum of physical parallelism available in the range of GPU devices.

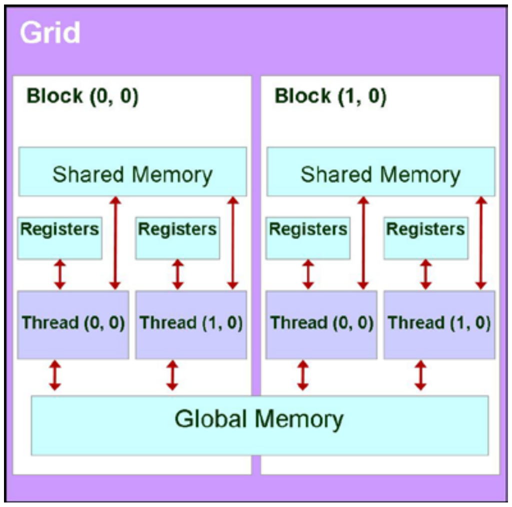

Fig. 3. The CUDA programming model.

C. Memory Spaces

The CPU host and the GPU device maintain their own DRAM and address space, referred to as host memory and device memory (on-board memory). The latter can be of three different types. From inner to outer, we have constant memory, texture memory and global memory. They all can be read from or written to by the host and are persistent through the life of the application. Texture memory is the more versatile one, offering different addressing modes as well as data filtering for some specific data formats. Global memory is the actual on-board video memory, usually exceeding 1 GB of capacity and embracing GDDR3/GDDR5 technology. Constant memory has regular size of 64 KB and latency time close to a register set. Texture memory is cached to a few kilobytes. Global and constant memories are not cached at all.

D. Programming Elements

There are some important elements involved in the conception of a CUDA program that are key for understanding the programming model as well as the optimizations we have carried out during the implementation phase. We describe them below and Fig. 3 summarizes their relations.

A program is decomposed into **blocks** running in parallel. Assembled by the developer, a block is a group of threads that is mapped to a single multiprocessor, where they can share 16 KB of memory (see Fig. 2). All the threads in blocks concurrently assigned to a single multi-processor divide the multiprocessor's resources equally amongst themselves. The data is also divided amongst all of the threads in SIMD fashion explicitly managed by the developer.

A **warp** is a collection of 32 threads that can physically run concurrently on all of the multiprocessors. The size of the warp is less than the total number of cores due to memory access limitations. The developer has the freedom to determine the number of threads to be executed, but if there are more threads than the warp size, they are time-shared on the actual hardware resources. This can be advantageous, since time-sharing the ALU resources amongst multiple threads can overlap the memory latencies when fetching ALU operands.

A **kernel** is a code function compiled to the instruction set of the device, downloaded on it and executed by all of its threads. Threads run on different processors of the multiprocessors sharing the same executable and global address space, though they may not follow the same path of execution, since conditional execution of different operations on each multiprocessor can be achieved based on a unique *threadID*. Threads also work independently on different data according to the SIMD model described in Section III-B. A kernel is organized into a grid as a set of **thread blocks**.

A **grid** is a collection of all blocks in a single execution, explicitly defined by the application developer, which is assigned to a multiprocessor. The parameters invoking a kernel function call define the sizes and dimensions of the thread blocks in the grid thus generated, and the way hardware groups threads in warps affects performance, so it must be accounted for.

A **thread block** is a batch of threads executed on a single multiprocessor. They can cooperate together by efficiently sharing data through its shared memory, and synchronize their execution to coordinate memory accesses using the *__syncthreads()* primitive. Synchronization across thread blocks can only be safely accomplished by terminating a kernel. Each thread block has its own *threadID*, which is the number of the thread within a 1D, 2D or 3D array of arbitrary size. The use of multidimensional identifiers helps to simplify memory addressing when processing multidimensional data. Threads placed in different blocks from the same grid cannot communicate, and threads belonging to the same block must all share the 8K registers and 16 KB of shared memory on a given multiprocessor. This tradeoff between parallelism and thread resources must be wisely solved by the programmer to maximize performance on a certain architecture given its limitations.

At the highest level, a program is decomposed into kernels mapped to the hardware by a grid composed of blocks of threads scheduled in warps. No inter-block communication or specific schedule-ordering mechanism for blocks or threads is provided, which guarantees each thread block to run on any multiprocessor, even from different devices, at any time.

The number of blocks in a thread block is limited to 512. Therefore, blocks of equal dimension and size that execute the same kernel can be batched together into a grid of thread blocks. This comes at the expense of reduced thread cooperation, because threads in different thread blocks from the same grid cannot communicate and synchronize with each other. Again, each block is identified by its *blockID*, which is the number of the block within a 1D or 2D array of arbitrary size for the sake of a simpler addressing to memory.

Kernel threads are extremely lightweight, i.e. creation overhead and context switching between threads and/or kernels is negligible.

IV. OPTIMIZATIONS

Once that major hardware and software limitations have been introduced, it becomes clear that managing those limits is critical when optimizing applications. Programmers still have a great degree of freedom, though side effects may occur when deploying strategies to avoid one limit, causing other limits to be hit.

We consider two basic pillars when optimizing an application to run on CUDA GPUs: First, organize threads in blocks to maximize parallelism, enhance hardware occupancy and avoid memory banks conflicts. Second, access to shared memory wisely to maximize arithmetic intensity and reduce global memory usage. We address each of these issues separately now.

A. Threads Deployment

Each multiprocessor contains 8192 registers which will be split evenly among all the threads of the blocks assigned to that multiprocessor. Hence, the number of registers needed in the computation will affect the number of threads which can be executed simultaneously, and the management of registers becomes important as a limiting factor for the amount of parallelism we can exploit.

The CUDA documentation suggests a block to contain between 128 and 256 threads to maximize execution efficiency. A tool developed by Nvidia, the CUDA Occupancy Calculator, may also be used as guidance to attain this goal. For example, when a kernel instance consumes 16 registers, only 512 threads can be assigned to a single multiprocessor. This can be achieved by using one block with 512 threads, two blocks of 256 threads, and so on.

We followed an iterative process to achieve the lowest execution time: First, the initial implementation was compiled using the CUDA compiler and a special *-cubin* flag that outputs the hardware resources (memory and registers) consumed by the kernel. Using these values in conjunction with the CUDA Occupancy Calculator, we were able to analytically determine the number of threads and blocks that were needed to use a multiprocessor with maximum efficiency.

B. Memory Usage

Even though video memory delivers a magnificent bandwidth, it is still a frequent candidate to hold the bottleneck when running the application because of its poor latency (around 400 times slower compared to shared memory) and the high floating-point computation performance of the GPU. Attention must be paid to how the threads access the 16 banks of shared memory, since only when the data resides in different banks can all of the available ALU bandwidth truly be used.

Each bank only supports one memory access at a time; simultaneous memory bank accesses are serialized, stalling the rest of the multiprocessor's running threads until their operands arrive. The use of shared memory is explicit within a thread, which allows the developer to solve bank conflicts wisely. Although such optimization may represent a daunting effort, sometimes can be very rewarding: Execution times may decrease by as much as 10x for vector operations and latency hiding may increase by up to 2.5x.

Another critical issue related to memory performance is data coalescing. A coalesced access involves a contiguous region of global memory where the starting address must be a multiple of region size and the kth thread in a half-warp must access the kth element in a block being read. This way, the hardware can serve completely two coalesced accesses per clock cycle, maximizing memory bandwidth, bus usage and throughput. It is programmer's responsibility to organize memory accesses in such a way, though CUDA has relaxed the conditions to be fulfilled for coalescing in their latest versions (from Compute Capabilities 1.2 on).

V. CONCLUDING REMARKS

We have presented the CUDA programming model and hardware interface as a very compelling alternative for high-performance computing when applied to electromagnetic simulations. Particular features of these simulations are identified and a number of techniques and optimizations are introduced to wrench the full performance out of the GPU resources for a large class of important scientific applications, even unveiling opportunities for further innovation.

GPUs are highly scalable and become more valuable for general-purpose computing. We envision electromagnetic simulations as one of the most exciting fields able to benefit from GPUs in the future of this emerging architecture. Additionally, new tools like CUDA and OpenCL may assist non-computer scientists with a friendlier interface for adapting these applications to GPUs. This computational power may then be multiplied on a cluster of GPUs to enhance parallelism and provide even faster responses to electromagnetic simulations at a very low cost.

Alternatively, we may think of a CPU-GPU hybrid system where an application can be decomposed into two parts to take advantage of the benefits of this bi-processor platform, and the programming models must evolve to include programming heterogeneous manycore systems including both CPUs and GPUs.

GPUs will continue to adapt to the usage patterns of both graphics and general-purpose programmers, with a focus on additional processor cores, number of threads and memory bandwidth available for electromagnetic simulations. In addition, the programming models must evolve to include programming heterogeneous manycore systems including both CPUs and GPUs.

REFERENCES

[1] GPGPU, “General-purpose computation using graphics hardware”, http://www.gpgpu.org, 2009.

[2] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell, “A survey of general-purpose computation on graphics hardware,” *Journal of Computer Graphics Forum*, vol. 26, pp. 21–51, 2007.

[3] S. Guha, S. Krisnan, and S. Venkatasubramanian, “Data visualization and mining using the GPU,” *Tutorial at 11th ACM Intl. Conference on Knowledge Discovery and Data Mining*, 2005.

[4] N. K. Govindaraju, B. LLoyd, W. Wang, M. Lin, and D. Manocha, “Fast Computation of Database Operations Using Graphics Processors,” *ACM SIGMOD International Conference on Management of Data*, pp. 215–226, 2004.

[5] R. Yang and M. Pollefeys, “A Versatile Stereo Implementation on Commodity Graphics Hardware”, *Real Time Imaging*, vol. 11, no. 1, pp. 7–18, February 2005.

[6] T. Sumanaweera and D. Liu, “Medical Image Reconstruction with the FFT,” *GPU Gems*, March 2004.

[7] I. Viola, A. Kanitsar, and M. E. Groller, “Hardware Based Nonlinear Filtering and Segmentation Using High-Level Shading Languages,” *IEEE Visualization*, pp. 309–316, October 2003.

[8] M. Hadwiger, C. Langer, H. Scharsach, and K. Buhler, “State of the art report on GPU-based segmentation,” *VRVis Research Center*, Tech. Rep. TR-VRVIS-2004-17, 2004.

[9] W. Wu and P. Heng, “A hybrid condensed finite element model with GPU acceleration for interactive 3D soft tissue cutting: Research articles”, *Computer Animation and Virtual Worlds*, vol. 15, no. 3-4, pp. 219–227, 2004.

[10] M. Harris, “Fast Fluid Dynamics Simulation on the GPU,” *GPU Gems*, 2004.

[11] P. Sander, N. Tartachuk, and J. L. Mitchell, “Explicit Early-Z Culling for Efficient Fluid Flow Simulation and Rendering”, *ATI Research Journal Technical Report*, August 2004.

[12] Y. Zhao, Y. Han, Z. Fan, F. Qiu, Y. Kuo, Kaufman, and K. A., Mueller, “Visual simulation of heat shimmering and mirage,” *IEEE Trans. on Visualization and Computer Graphics*, vol. 13, no. 1, pp. 179–189, 2007.

[13] CUDA, “Home page maintained by Nvidia” http://developer.nvidia.com/object/cuda.html.

[14] Brook+, “Web Page maintained by AMD”, http://ati.amd.com/technology/streamcomputing/AMD-Brookplus.pdf, 2009.

[15] “Nvidia Tesla GPU computing solutions for HPC” http://www.nvidia.com/object/tesla_computing_ solutions.html, 2009.

[16] Firestream, “AMD Stream Computing”, http://ati.amd.com/technology/streamcomputing.

[17] T. K. Group, “The OpenCL Core API Specification, Headers and Documentation,” http://www.khronos.org/registry/cl, 2009.

[18] E. Kelmelis, J. Durbano, P. Curt, and J. Zhang, “Field-programmable gate array accelerates FDTD calculations,” *Laser Focus World*, September 2006.

[19] S.E. Krakiwsky, L.E. Turner, M.M. Okoniewski“, Acceleration of finite-difference time-domain (FDTD) using graphics processing units (GPU),” *IEEE MTT- S Int. Conference*, June 2004.

[20] http://www.acceleware.com/em

[21] http://www.cst.com/

[22] T. Hartley, U. Catalyurek, A. Ruiz, M. Ujaldon, F. Igual, and R. Mayo“, “Biomedical Image Analysis on a Cooperative Cluster of GPUs and Multicores,” *22nd ACM Intl. Conf. on Supercomputing*, 2008.

[23] P. So, “EM-based simulation tools for signal and systems analysis”, *International Symposium on Signals, Systems and Electronics*, August 2007.

[24] M. Harris, “Manycore parallel computing with CUDA”, *Keynote Session at the 22nd ACM Intl. Conference on Supercomputing*, June 2008.

[25] Ageia, "The PhysX co-processor", http://www.nvidia.com/object/nvidia_physx.html.

[26] T. R. Halffill, "Parallel Processing with CUDA", *MicroProcessor Report Online*, January 2008.

[27] Nvidia Compute Unified Device Architecture (CUDA) Programming Guide v. 1.1, Nov. 2007.

[28] Nvidia CUDA CUBLAS Library v. 1.1, Sep. 2007.

[29] Nvidia CUDA CUFFT Library v. 1.1, Oct. 2007.

Manuel Ujaldon received his B.S. degree in Computer Science from the Univ. of Granada (Spain, 1991) and his M.S. and Ph.D. degrees in Computer Science from the Univ. of Malaga (Spain, 1993 and 1996). During 1994 and 1995 he was a Research Assistant in the Computer Architecture Dept. at the University of Malaga, where he became Assistant Professor in 1996 and Associate Professor in 1999.

Dr. Ujaldon was a predoctoral and postdoctoral researcher at the Computer Science Dept. of the University of Maryland (USA, 1994, 1996/97) and Biomedical Informatics Department of the Ohio State University (USA, 2003-08).

He has published 8 books on computer architecture and more than 50 papers in international peer-reviewed journals and conferences. His research interest includes streaming architectures as well as compiler and software development for running general-purpose scientific applications on GPUs.