

ARMY RESEARCH LABORATORY



**On the Incorporation of Streaming Processors for Chemical,
Biological, Radiological, and Nuclear Models**

by Dale Shires, Song Park, and Chris Gaughan

ARL-TR-6093

August 2012

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory

Aberdeen Proving Ground, MD 21005

ARL-TR-6093

August 2012

On the Incorporation of Streaming Processors for Chemical, Biological, Radiological, and Nuclear Models

Dale Shires and Song Park
Computational and Information Sciences Directorate, ARL

Chris Gaughan
Simulation and Training Technology Center
U.S. Army Research Laboratory
Orlando, FL

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188		
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.					
1. REPORT DATE (DD-MM-YYYY) August 2012		2. REPORT TYPE		3. DATES COVERED (From - To)	
4. TITLE AND SUBTITLE On the Incorporation of Streaming Processors for Chemical, Biological, Radiological, and Nuclear Models			5a. CONTRACT NUMBER		
			5b. GRANT NUMBER		
			5c. PROGRAM ELEMENT NUMBER		
6. AUTHOR(S) Dale Shires, Song Park, and Chris Gaughan			5d. PROJECT NUMBER		
			5e. TASK NUMBER		
			5f. WORK UNIT NUMBER		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) U.S. Army Research Laboratory ATTN: RDRL-CIH-C Aberdeen Proving Ground, MD 21005			8. PERFORMING ORGANIZATION REPORT NUMBER ARL-TR-6093		
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)			10. SPONSOR/MONITOR'S ACRONYM(S)		
			11. SPONSOR/MONITOR'S REPORT NUMBER(S)		
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT High performance computing (HPC) is in a state of transition. HPC users have traditionally relied upon two things to supply them with processing power: speed of the central processing units (CPUs) and the scalability of the system. There are problems with this approach. Physical limitations are curtailing clock speed increases in general-purpose CPUs, the von Neumann load-execute-store approach does not map well to every computational problem, and systems of thousands of processors might be very inefficient depending upon processor interconnection limitations. Several versatile, commodity-based options are coming online that could help address these deficiencies. Each of these can be used to provide performance that at one time was only available by using application-specific integrated circuits (ASICs) or large-scale fixed HPC assets. Newer methodologies hold out the hope of being more cost efficient and deployable along with providing faster deployment and development times and allowing the use of algorithms that remain modifiable at all stages of development and fielding. This report assesses these technologies from the standpoint of system models found in chemical, biological, radiological, and nuclear (CBRN) defense systems. We discuss the characteristics of streaming processors and how they may be applied to CBRN problems for increased efficiency and cost-effective computing.					
15. SUBJECT TERMS Graphics processing unit, CBRN, SPM, GPU					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPHONE NUMBER (Include area code)
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UU	34	Dale Shires (410) 278-5006

Contents

Acknowledgments	vi
1. Introduction	1
2. Benefits of Heterogeneous Computing	2
3. Throughput Architectures	4
4. Programming for Parallelism	6
4.1 Traditional Multi-core	6
4.2 Streaming Processors	6
5. Understanding Performance	8
6. Heterogeneous Computing for System Performance Models	10
6.1 Decontamination	10
6.2 Contamination Avoidance	13
6.2.1 Lagrangian Particle Dispersion	13
6.3 Individual Protection	16
6.4 Collective Protection	16
7. Cost-benefit Analysis	17
8. Future Trends	19
9. Conclusions	20
References	22

List of Symbols, Abbreviations, and Acronyms	24
Distribution	26

List of Figures

1	Customized GPU-accelerated workstation.	4
2	Simplified GPU and thread execution.	5
3	Example signal processing control flow graph.	12
4	Floating-point performance per dollar.	18
5	Floating-point performance per watt.	18

Acknowledgments

This work was supported by a research grant from the Defense Threat Reduction Agency (DTRA).

1. Introduction

Traditional high performance computing (HPC) has been built upon a long history of various processor types and configurations. Vector processors, simple bit-based single instruction multiple data (SIMD) processors, and more mainstream von Neumann processors have all been successfully deployed in HPC architectures. Enhancements to these processors, including superscalar processors with out of order instruction support, have allowed for incremental advances along the way. There has also been no lack of creativity in constructing elaborate architectures for these systems. Three-dimensional torus configurations, bristled hypercubes, and the more current clustering technologies all seek to minimize performance degradation as processors communicate while computing across the global HPC configuration.

However, HPC is rapidly evolving. Changes are driven largely by market forces and physical limitations in current processor fabrication techniques. This evolution is moving away from scalar processors that follow the familiar von Neumann fetch-execute-store instruction path. Attempts to wring more performance out of these architectures by increasing clock rates were beginning to lead to prohibitive heat generation and power requirements. To compensate, chip manufacturers began shifting to more cores clocked at lower speeds (1).

An example is illustrative (2). Consider the equation for power:

$$p = c \times v^2 \times f \tag{1}$$

where p is power, c is capacitance, v is voltage, and f is frequency. To circumvent the unsustainable power trend, voltage scaling is applied to reduce a CPU's power. This shift to multi-core is a notable example of power influencing architecture design.

While standard computing cores were fundamentally changing, another class of computer hardware began to open up to software developers. Graphics processing units (GPUs), which have historically been add-on cards plugged into computer motherboards, are now general-purpose and can be used for things other than just graphics processing. These devices are quite different from standard computer cores but can deliver unmatched price/performance metrics for floating point intensive applications. Our research has shown that they also perform quite well on integer-based codes as well.

Usually when individuals decide to invest the time and energy to develop GPU-aware applications (and it can be an investment), they are after one goal in particular: code speedup. With the advent of programmable GPUs, many researchers now have the capability at delivering what was once considered HPC-level performance using only

commodity GPU systems. HPC capabilities and capacity are now available to a much wider audience. Getting optimal performance from these heterogeneous CPU-GPU systems, however, is not always easy and intuitive, but can be critical in many application areas from large-scale physics-based codes to operational codes that must perform at or near real-time speeds. Our particular focus area here is on chemical, biological, radiological, and nuclear (CBRN) defense systems. By leveraging GPU technology, it is possible to create an environment rich in computational capabilities that will allow for cost-effective and highly efficient systems in the CBRN range of operations.

2. Benefits of Heterogeneous Computing

Heterogeneous computing is an approach to computing that couples disparate computing cores. While not exhaustive, there are four processor types that are of interest to the discussion here as these are the primary types referenced in heterogeneous computing. Starting from the most hardware-centric and least flexible, we have Application Specific Integrated Circuits (ASICs), Field Programmable Gate Arrays (FPGAs), GPUs, and CPUs.

ASICs are geared primarily for performance and are dedicated architectures. They can often be found in embedded systems. ASICs can easily dominate the costs of system production (3). Accordingly, ASICs are most cost effective when produced in large volumes since initial circuit design and chip fabrication can add up. Additionally, ASICs can take a long time from initial design and testing to production (4). Finally, the functionality of the design is locked in upon fabrication. Should errors be identified, they can only be corrected by new logic on new chips. This is also the case should improvements be made to the logic.

Unlike ASICs, FPGAs are chips that are comprised of programmable logic circuits. Because of this, systems built predominately from FPGAs are known as reconfigurable systems (hence the term “reconfigurable computing” when using these devices). With an FPGA, data path and control flow can be modified in hardware as necessary to reduce an algorithm’s execution time. FPGAs are built upon static random access memory (SRAM) technology and are limited by the amount of memory fabric from which circuits can be built. Floating-point units can take a lot of space to implement on FPGAs and, for this reason, have not been competitive for these types of calculations. However, FPGAs have been shown to be good performers for bit-and integer-based applications (5).

GPUs have only recently been opened up to developers through newer application programming interfaces (APIs) such as NVIDIA’s Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL). In a sense, GPUs can be considered as a specialized solution for parallel floating-point operations. They were optimized to

perform many simple operations for graphics rendering support in a fast, parallel fashion. By opening them up using intuitive programming frameworks (over more complex Open Graphics Library (OpenGL) models geared towards graphics), data-intensive and parallel operations can now be offloaded leaving the CPU extra cycles for more complex codes.

Done right, heterogeneous computing can blend the various strengths of the various processing cores, CPUs, GPUs, and FPGAs, and even scale these assets to provide high performance computing solutions. In fact, this approach provides at least three major advantages over approaches founded only in standard CPU computing or ASIC-based solutions. First, heterogeneous computing is cost-effective since it is built on commodity resources. Potential costs to fielding systems is also reduced. Second, heterogeneous systems are flexible. They avoid dedicated circuit designs and allow the latest research advances to be incorporated into solutions since the system remains changeable through software. Finally, and perhaps more importantly, performance can scale quite high both per-core and as a clustered solution.

By intelligently packing as much computing power into small footprints as possible, heterogeneous systems built predominately with GPU accelerators can achieve high theoretical flop rates and still be deployable in normal operational settings. By using a specialized water block allowing for GPU water-cooling, the U.S. Army Research Laboratory (ARL) has custom-engineered a GPU-accelerated workstation allowing up to seven cards to be installed in a workstation chassis, which is illustrated in figure 1. The chassis was slightly modified to allow for more cooling, but this coupled with the water-based heat dispersion allows the system to run in just about any operational setting. The performance is bordering on what was once only common in large air-conditioned HPC centers. With the installed Sapphire ATI Radeon HD 4870X2 cards, the theoretical peak is about 16.80 TFLOPs using reference clocks and up to 20.16 TFLOPS at 20% overclocking. Double-precision performance comes in at 3.36 TFLOPS using reference clocks and 4.03 TFLOPS once again at 20% overclocking.

However, heterogeneous computing is not without its challenges. Different instruction sets can cause a problem in that codes cannot be easily ported or optimized across architectures. For instance, it is up to the developer to identify computing resources for different code segments pairings. Integer- or bit-based functions may be set aside for FPGAs, floating-point for GPUs, and overall control flow for standard multi-core CPUs. This approach requires experts in many differing areas. Novel techniques, such as the Low Level Virtual Machine (LLVM), are attempting to address this constant need to revisit code for new architectures. A common shared code representation at a low level can provide a way to not only easily move code between devices, but also opens up the possibility to optimal hardware selection on the fly and the ability to optimize codes across resources (inter-resource). For example, optimizing compilers can do a good job at



Figure 1. Customized GPU-accelerated workstation.

optimizing code within a function and can do moderately well interprocedurally. All of these analysis and optimization approaches hit a wall when trying to go interdevice. With LLVM, a goal is to be able to characterize each heterogeneous device and actually analyze, optimize, and execute across all available processors. This will all be facilitated by a common code representation.

3. Throughput Architectures

Throughput architectures come in various configurations and instantiations. Just about all of them have their origins in the commercial sector where virtual world simulation found in gaming engines requires large FLOP rates. At the bottom layer of execution, throughput architectures rely on SIMD parallelism. The SIMD execution model exploits data-level parallelism by operating a single instruction on different data sets. By having multiple SIMD capable processors on a chip, throughput architectures can execute independent SIMD instructions in parallel.

Hardware multi-threading attempts to hide the effect of stalls by letting the processors do some other productive task. This technique allows the general purpose use of GPUs to be promising for applications with large amount of threads. GPU processing cores are heavily multi-threaded, typically requiring thousands of threads per core to achieve good

performance. Two main factors can cause slowdown in these systems. The first deals with branches and data dependencies in a code. Codes with many switch statements and conditionals, as well as temporal dependencies, can lead to underuse of available computing units while waiting for results to be computed. The second is far worse and deals with a code that has to access remote memory. These stalls can cost hundreds to thousands of cycles. Creating a large number of threads helps to cover these stall delays by allowing the compute core to switch to a new thread when one stalls. Different memories are available at different speeds, thus it is usually up to the application developer to pay close attention to memory access patterns to achieve good speedup. Compiler’s optimizations to do this task are slowly making progress.

Figure 2 illustrates a simple GPU core and how threads are executed (context switching) to keep the core busy (6). The core (figure 2a) can execute an instruction from one thread for each clock cycle but can maintain thread states for four threads. With the arithmetic logic units (ALUs) acting as SIMD processors, each can execute a vector-type instruction in a single clock (in this case, 32 concurrent instructions). Floating-point general-purpose vector registers are available and partitioned among the thread contexts. At runtime, the GPU runs a copy of the executable on each of the four thread contexts (figure 2b). T0 is executed until a stall is detected at cycle 20 (resulting from a memory reference that needs to be fetched or an instruction to complete). A context switch occurs to allow T1 to continue at cycle 20. T0 becomes ready again at cycle 70 and begins to execute again at cycle 80 after the stall by T3. Notice that this schedule allows all 32 ALUs to be busy during the entire run. Also notice conceptually the large number of threads that are required to be running to allow the runtime system to efficiently keep the ALUs busy through context switching.

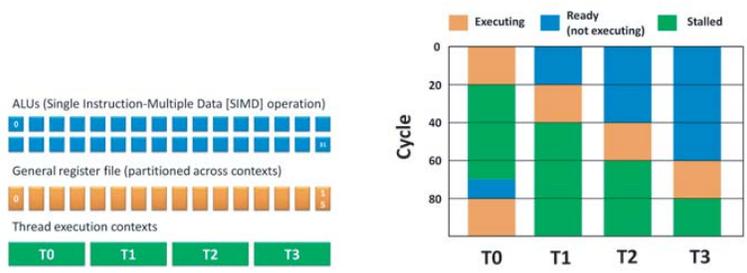


Figure 2. Simplified GPU and thread execution.

Just as HPC has sought performance gains by scaling CPU counts to very high levels, GPUs also seek performance by scaling as well. This scaling is both internal to the GPU in the form of more and more cores per chip, and also in aggregate numbers of GPUs by clustering. Just by crunching the numbers, it is easy to arrive at a machine with very high theoretical performance by coupling general-purpose CPUs and GPUs. However, designing an optimal configuration requires a bit more thought than simply buying lots of each.

4. Programming for Parallelism

In the following sections, we briefly discuss the methodologies used to program general-purpose computing processors and techniques that are common in the evolving world of GPUs.

4.1 Traditional Multi-core

The message passing interface (MPI) uses a single program-multiple data (SPMD) methodology where parallelism comes from each core running an identical copy of the program but with a distinct memory and program counter. Conditional statements within the code are used to control the behavior of the individual processes based on a unique MPI process identifier (usually polled once at program startup). For divide and conquer methods, process 0 usually acts as the boss with the rest of the pool functioning as the workers. MPI is known to be very efficient for wide classes of applications and parallelism (task, data, domain decomposition, etc.) (7). Low-level details of effective implementation, however, are mostly left to the developer. Hence, the time to field a completed application can be costly.

Multi-platform shared-memory systems have also been able to take advantage of the OpenMP implementation for describing parallelism (8). OpenMP uses a fork and join paradigm where worker threads are spawned at code segments with high levels of data parallelism (usually associated with loop constructs). OpenMP can achieve a quick turnaround for codes that have very isolated loops with large execution time requirements. It is also portable and works within the C/C++ and FORTRAN languages.

4.2 Streaming Processors

Achieving the maximum performance advantage on a GPU requires a detailed understanding of the hardware and memory systems on which they are designed. This is somewhat unfortunate, since with the advent of high-level programming languages we have always attempted to move away from low-level details. However, just as with general HPC software development, there will always be a need to dig deeper into software translation to low-level instructions when high performance is sought. Just as cache blocking techniques had to be developed to mitigate cache thrashing on tiered memory chips, so too do techniques have to be developed on GPUs that target memory hierarchy.

GPUs were originally bound to the OpenGL set of high-level instructions. Released in the early 1990s, the design of OpenGL had several goals including industry-wide acceptance,

consistent and innovative implementations, long life, and high quality, to name a few (9). However, the name says it all; this was a development geared mainly towards graphics processing. Using OpenGL to program GPUs for any reason other than graphics made little sense.

One of the first efforts to make GPUs available to the general computing public began at Stanford University in the early 2000s. As the hardware for GPUs continued to increase in performance and numbers of cores, the need for a programming environment to tap into these processors became evident. Brook and BrookGPU were developed as extensions to the ANSI C language to provide a new computational model known as streaming (10). The Brook API provides for two main benefits over traditional functional programming languages. First, it expresses data parallelism, for which GPUs are ideally suited. Second, it encourages developers to compute rather than communicate (11). GPUs are excellent at doing math, but communication causes stalls that severely limit performance.

Brook, also known as Brook+, found its way into a commercialization effort when AMD included it as part of their first release of the Stream Computing Software Development Kit (SDK) in 2007. This methodology was supported on AMD GPUs, which were organized into SIMD cores where each core contained some number of very long instruction word (VLIW) processors. Also included in this distribution were several other components and layers including the AMD core math library, AMD performance library, and the Compute Abstraction Layer (CAL) (12). CAL is a low-level access methodology that can give performance increases with tuning at this low level. This process, however, can be quite tedious. As of January 2011, AMD is now relaunching their ATI Stream SDK as the Accelerated Parallel Processing (APP) SDK as it now also supports accelerated processing units (APUs). The AMD Fusion APU is a chip that features both x86 cores and a GPU on the same die.

In early 2007, NVIDIA, another manufacturer of GPUs, released its CUDA as an API to allow for the general purpose programming of GPUs. CUDA supports the concept of threads on the NVIDIA GPU cores with the programmer's goal of finding enough parallelism in the code to allow for potentially thousands of threads in flight (13). These threads are then swapped in and out of processing depending upon their wait cycles for results to compute or memory reference lookups to complete. CUDA uses a subset of the C and C++ languages.

A major problem with these APIs is a lack of portability. Codes written in CUDA would only execute on NVIDIA hardware, Brook+ on AMD GPUs, and so forth. In 2008, a new API began to emerge that wasn't simply focused on portability across resources, but also targeted all resources within a heterogeneous platform (14). With the OpenCL, kernels can now be constructed to run in parallel on any series of devices concurrently, from CPUs to

GPUs or even digital signal processors (DSPs). It can effectively target modern processors housed in personal computers, servers, or the growing industry of handheld devices.

The OpenCL programming model consists of one host with at least one compute device. Each of these compute devices is further decomposed into one or more processing elements (14). When these compute devices happen to reside within the same node, OpenCL is usually sufficient to parallelize across all the devices. When there are multiple devices on multiple nodes, traditional clustering techniques, such as MPI, can be used to extend data parallel approaches across the devices (15). So, large-scale HPC using OpenCL requires some knowledge of traditional divide and conquer data processing.

5. Understanding Performance

There are several ways to analyze and study performance of computing applications. At perhaps the highest level is the simple determination of real-time versus non real-time computing requirements. If a code must perform at real-time speeds, then it must perform some function in a required wall clock time boundary (it should be noted that some real-time codes can be constructed to actually perform at greater than real-time speeds). These codes can become complicated largely due to issues with synchronizing input and output streams and determining how the code should react if input and output streams become delayed or saturated. Codes that do not or cannot operate in real-time frames are often based on scales that are simply too large or are too high fidelity to generate results in reasonable times.

Speedup and efficiency are calculated based on some lower bound time solution. Determining this lower bound is more difficult than it might first appear. For example, in some cases, the most efficient algorithm for serial (or one processor) execution might not be the best candidate when moving to two or more processors. This is also the case when comparing CPU-based solutions to GPU-based implementations. With the memory structure of CPUs, it is often best to compute heavily used scalars once and store those results to be looked up at each use. Doing this on a GPU can often stall the pipelines; it's often best to simply recompute reused scalars. Single or multi-core computing times are also heavily dependent on how much optimization was done on the source code. In many cases code can be heavily optimized with increased performance by using specialized instructions tailored for architectures. Perhaps the best example of this is the Streaming SIMD Extensions (SSE) instruction set extension to the x86 architecture. By packing data into 128-bit registers, SSE allows the same operation to be performed on multiple data where previously a loop or several instructions would be needed. Not counting branch and bound conditional checks inside a loop, it is fairly easy to see how one instruction vice four

could result in a decreased execution time of roughly 75%. Explicit programming of SSE optimization can be fairly time consuming, so using it is a decision that should be based on effort versus gain. In many cases where sequential or multi-core CPU solutions are compared against GPU times, the perceived gain by using GPUs might in fact not be as large if SSE were employed. This isn't to say that non-SSE optimized codes are straw man arguments when it comes to performance measurements versus GPUs (given the effort issue), but one should keep in mind that optimal performance on any architecture is a bit subjective to skill, effort, and a host of computer science issues.

As previously stated, we consider wall clock time to be the predominantly important factor when it comes to judging success of parallelism. A common question that lingers in the field has to do with the measured FLOP rate of a code, but this alone does not say much about a code in practice. High FLOP rate codes that are really using all of the available floating-point instructions per cycle do not always translate to “fast” codes. In fact, these codes may take far longer in wall clock time than other codes with lower FLOP rates but more efficient algorithms. For example, an algorithm with a minimal FLOP rate that takes 1 h to complete on one processor compared to an algorithm with a high FLOP rate that takes 10 h to complete is 10 times more efficient.

When discussing wall clock time comparisons or comparisons of completion time based on the number of processing units used, two metrics that can be useful are speedup and efficiency. The actual speedup achieved is defined as

$$S_p = \frac{T_l}{T_p}, \quad (2)$$

in which T_l is the linear (one processor) completion time for the algorithm and T_p is the parallel completion time using p processors. The second metric used is efficiency. Efficiency is defined as

$$E_p = \frac{T_l}{p \cdot T_p}. \quad (3)$$

This number indicates the overall efficiency of the p processors working on the problem. Ideally, this number should be as close to 100% as possible, but will suffer because of limited parallelism (Amdahl's Law), conditions of load imbalancing, communication costs, and various other parallelization overheads.

6. Heterogeneous Computing for System Performance Models

Given the advances and advantages of heterogeneous computing solutions over monolithic and ASIC approaches, it is understandable that many application areas are seeking their incorporation to increase performance or allow for higher fidelity models. Here we comment on applicability in the realm of system performance models (SPMs) as they are defined by the Defense Threat Reduction Agency (DTRA) Joint Science and Technology Office (JSTO). In particular, these models are for test and evaluation of CBRN material focused on decontamination, contamination avoidance, individual protection, and collective protection.

6.1 Decontamination

The Decontamination SPM, code named CREATIVE, is chartered with developing a predictive decontamination model that can accurately assess the performance of decontaminates and capture the residual surface hazards after decontamination events. These models are currently at the early generations of development. Laboratory scale with simple surfaces are the focus to provide a fundamental model, and each additional refinement increases complexity (such as to more complex surfaces) to eventually arrive at a full-scale scenario simulation. The concept is built on physical and semi-empirical deterministic models that employ a finite difference approach to describe the mass transport and system interactions, thus allowing for a post-decontamination risk assessment.

The parameter space for these models is quite large and grows quickly with each additional component that is factored into the simulations. Furthermore, determining the optimal and proper efficacy could involve numerous trials of various parameters with the requisite mining of outcomes. All of this adds significantly to the computing demands of the model, especially as it is extended to a more automated system.

Currently, models are coded using a development platform consisting of software running in MATLAB on single CPU cores. Model results are written to proprietary file formats in word processing and spreadsheet documents. While MATLAB is an extremely convenient development environment ideal for testing ideas and leveraging a huge library of functionality, it is not known for speed of processing. MATLAB supports GPU acceleration from within the Parallel Computing Toolbox as a way to use the GPU's power without the user having to code in C or FORTRAN. There are currently two different ways to execute MATLAB code on a GPU. First, GPUArray objects can be created and used in functions that support them. These are element-wise functions and very data parallel; mathematical

functions or transforms are applied to each element of the array. It is anticipated that the performance of this approach will increase with the increased size of the array. That way startup costs for the GPU are ameliorated over the entire runtime. Second, there is an approach that will allow users to run their functions on the GPU. There are mechanisms in place that allow for reuse of data (arrays) that are already on the GPU. This is a potential major performance improvement since data movement between GPU and CPU is a performance killer.

This last point brings up an important consideration for library-based accelerator calls. Code developed explicitly for accelerators like GPUs will, in general, perform better than code parallelized around convenience-routine function calls. This largely has to do with two issues: the performance penalties every time control has to move between the CPU and the GPU, and the possibility that the size of the data set, n , is too small to warrant this approach.

Consider the example program control flow graph shown in figure 3. Further assume that we have an FFT library that runs on the GPU accelerator. We can achieve some level of speedup by simply calling the GPU-based fast Fourier transform (FFT) library in place of a CPU-optimized FFT. However, this will only be possible if the size of the input A is of sufficient level (determined by numerous factors including GPU characteristics and bus transfer penalties). If the size of A is not sufficient, the code may actually take longer to complete.

Following this, the next logical steps would be to develop code for the `Filter_1` and `Filter_2` functions on the GPU. The modified results of A will remain resident on the GPU, thus removing any need to move this data over interconnect channels between the CPU and GPU. If the size of A is somewhat diminutive, then it may actually be advantageous for the code developer to put even simple versions of the FFT or other intermediate computations on the GPU just to keep the computing cycles resident there and not have to return to the CPU. As discussed in other areas of this report, at this point the developer may also wish to take the extra effort to investigate emerging languages such as OpenCL. The initial learning curve can be steep, but OpenCL currently supports execution on GPUs as well as x86-based architectures. Thus, it may be more portable and deliver performance where the required hardware exists.

Planned enhancements to the CREATIVE software involve contact testing that are based on Fick's second law. Literature searches for work on GPU implementations related to Fick's laws did not return any directly related results. However, since diffusion in two or more dimensions of Fick's second law is analogous to the heat equation, the search effort was expanded to also take into account the heat equation. Some success has been reported in solving large-scale three-dimensional (3-D) heat equations using GPUs and the CUDA

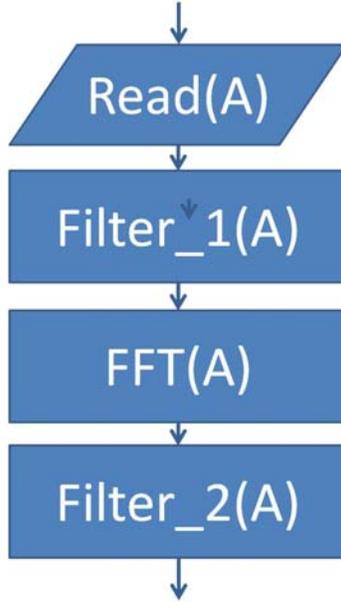


Figure 3. Example signal processing control flow graph.

framework (16). The implicit differential method is used for discretization and the conjugate gradient algorithm is applied to solve the discretized equation. Judging from the graphs summarizing the timing results, speedup was in the range of 7x to 20x when a Quadro FX 4600 was compared to four threads on a Xeon 5110.

Mass transport is the fundamental basis of decontamination modeling. This SPM uses a system of physical and semi-empirical deterministic models using a finite difference approach. Speedup is possible in two ways. First, working through the large parameter space could be treated as a task parallel approach with different processors operating on the various parameter combinations. Furthermore, inside of this, it is quite possible that the discretization of the system will port well to GPUs. An OpenCL-based implementation of a finite-difference time-domain (FDTD) method was developed for electromagnetic and seismic applications (17). The FDTD algorithms were developed in OpenCL and executed using ATI Radeon HD 5970 GPU co-processors. The authors compared a dual-GPU solution approach against an optimized C baseline code executing on an AMD Phenom II X6 1090T CPU. Acceleration for the electromagnetic and seismic FDTD algorithms were 3.7 \times and 4.2 \times , respectively, when executed on the dual GPU devices.

Other work has been done to accurately model the two-dimensional (2-D) acoustic wave using the FDTD approach with applications running on the GPU (18). The authors also used multiple GPUs, in this case four, by creating parallel thread counts using OpenMP on the CPU equal to the number of GPUs in the system. Double-precision was used for the increased precision required in their simulation environment. The authors varied the grid sizes between 800 squared and 12,800 squared grid cells. Little detail is provided on the

characteristics of the code executing on the CPU except that it is written in the C language. Not surprisingly, the larger the grid size the better the GPU performed. Four GPUs on the 6,400 squared grid size had a speedup of roughly $60\times$ over a single core CPU while only about $10\times$ on the smallest grid size. A more optimized C implementation would push these down somewhat.

The additional GPUs allowed not only for scaling comparisons, but in some cases, the testing of larger grid sizes where one or two GPUs would not have sufficient memory to run them. Larger grid sizes equate to more data available for the threads of the GPU to process, thus providing better efficiency and more speedup. For the smallest grid size, 800 squared, the total simulation time went from 78 s to 40 s when going from one to four GPUs. With an increase of $4\times$ in processing resources, only roughly $2\times$ was achieved in speedup. The largest grid size that would compute on one GPU, 6,400 squared, took 3,823 s to compute. With 4 GPUs, this time went down to 986 s. This speedup is just shy of $4\times$. More data, within memory limits, equates to better efficiency of the GPU resources.

6.2 Contamination Avoidance

The Contamination Avoidance (CA) SPM is concerned with detectors and contamination avoidance systems across a wide range of conditions. Either deliberate or accidental releasing of harmful lighter-than-air particles poses significant problems for first responders and officials trying to determine contamination areas along with areas that need to be evacuated. These determinations are best made with real-time atmospheric data (such as wind speed and direction) and can be complicated when urban settings are introduced. Ideally, these computer-driven models should be augmented with in-position sensor grids that can monitor dispersion at fixed, and possibly mobile, points.

It is somewhat easy to see that large-scale and long-term simulations in this regard quickly become taxing to even the largest of computer systems. However, it may also be desirable to field these simulation technologies in mobile platforms in order to field deployable systems in portable footprints to avoid potentially lengthy delays in setting up connectivity to large-scale HPC centers. This section discusses the possible use of GPU accelerators to address problems in CA. It should be noted that there does not, anecdotally, appear to be a large body of work in development of codes for this topic in the realm of GPU accelerators. We discuss some efforts using GPUs in the sections that follow.

6.2.1 Lagrangian Particle Dispersion

The National Center for Atmospheric Research (NCAR) is the primary agency investigating areas in CA with respect to the SPMs. Researchers at NCAR have also begun to develop Lagrangian Particle Dispersion Models (LPDMs) instantiated on various

computing platforms (19). Researchers employed a Lagrangian stochastic model (LSM) driven by large-eddy simulation (LES) data. Lagrangian models use passive particles that are released into the problem domain and tracked through time. The methodology is fairly floating-point intensive with a bulk of the computation being devoted to particle location updates. The methodology employs only single precision arithmetic as deemed appropriate by subject matter experts.

Researchers developed four implementations to determine GPU applicability and performance to the CA problem. The host computer system contained dual quad-core Intel Xeon X5450 processors, 8 GB of RAM, and PCI-Express bus. The operating system was Red Hat 4.1.2-46. The GPU device was the NVIDIA GTX 285 with 1 GB of RAM, 240 cores, and based on driver 190.18. Compilers used included gfortran from GCC 4.4.2, gcc from GCC 4.1.2, and CUDA release 2.3 V0.2.1221. Descriptions of the test input trial are found in reference 19. It should be noted that the code was recast to allow for single vice double precision accuracy. This allows for greater speedup on GPU assets.

However, it appears from the documentation that there were still serialization barriers within the code on file I/O.

1. Single-core Fortran 90

The single-core run time for base comparisons is 25,041 (averages release scenario “ave2”) s. The author of this study was able to restructure some array accesses for cache blocking and employed the -O3 level of optimization. In this regard, the serial code appears to be fairly well optimized for a baseline.

2. MPI and OpenMP multi-core Fortran 90

MPI outperformed OpenMP in all the test cases at 2, 4, 6, and 8 cores. This is somewhat to be expected as MPI forces parallelization across all threads since a separate copy of the program is running in each case. OpenMP focuses on parallelization at the loop level and usually, in turn, targets a subset of the entire program. Performance flattened quickly as core counts increased. Speedup was 1.9 and 1.7 for the MPI and OpenMP cases, respectively, at 2 cores. By 8 cores, the speedups were only 4.3 and 3.9, respectively. Considering efficiency, at 2 cores using MPI, the code was 96% efficient, which is a fairly impressive number. At 8 cores, MPI efficiency dropped to a mere 54%. Put into context, the processors at this point were idle about 50% of the time.

3. Brute force GPU based on CUDA C/C++

The author of this study wisely chose to follow suggestions and tips found in the CUDA programming guide made available from NVIDIA. Intelligently programming GPU devices can lead to an order of magnitude or more if not only done correctly

but intelligently. This brute force implementation focused on replacing two inner-most loops with a single call to a CUDA kernel installed on the GPU device. The four steps included preparation of the GPU for the kernel run, moving the data to the device, performing the computation, and moving results back to the host CPU. The GPU is primarily being employed for particle movement computation.

For this analysis, a completely new version of the code was constructed in C/C++ based on the original Fortran 90 code. If the source code is not “too” long, this is a fairly common approach since CUDA is itself like C/C++. The author appears to have done this conversion by hand and not relied upon source-to-source converters. The author notes that the C/C++ version in some areas was an order of magnitude slower than the Fortran 90 versions. Apparently, an investigation was not done but it is stated that the reason was not due to cache misuse. The theory put forth was that it was due to pointer dereferencing. The comment seems to indicate that it was verified that arrays were being referenced in the correct order to promote cache affinity for C vice Fortran. More detailed analysis is warranted here as properly coded C/C++ should not perform at this much of a slowdown.

Six different test cases were performed and measured against the new C/C++ code. Three were average release scenarios ranging from 8,000 to 81,000 particles and three were single release scenarios ranging across the same number of particles. Speedup values were calculated based on the fastest speeds achieved using the Fortran 90 serial version. This comparison is fair since it takes into account the fastest sequential runtime. Since the speedup profiles were roughly the same for both the average and single release workloads, we report only the times for the average cases, and in this set we report only the results for the greatest total particles released. The brute force approach came in at a speedup of $14\times$ over the sequential version, and was $2.7\times$ faster than the 8-core MPI method.

4. Scalable GPU based on CUDA C/C++

The brute force approach was limited in a number of ways. First, the limited memory of GPU boards was a problem should the LES data sets grow. Second, the workflow had the CPU and GPU constantly swapping control. This methodology can cause significant delays with processor startup and memory traffic on the PCI bus. Other factors such as limited memory reads and going across multiple GPUs were not addressed in the brute force approach. The authors attempted to rectify several of these issues in a more scalable approach by using a decomposition strategy based on tiling (multiple GPUs were not considered during the study). By intelligently governing particle movement within tiles and using asynchronous computing (the CPU and GPU active at the same time), much better results were achieved.

Compared to the brute force’s speedup gains of $14\times$ over the sequential version, this scalable approach achieved an impressive $20\times$ speedup over the Fortran one-core

result. None of this, however, appeared to come easy as many changes and modifications were required.

As follow on to this effort, ARL's Computational and Information Sciences Directorate (CISD) has made an offer to allow NCAR to study the performance of this simulation engine on a multiple GPU cluster. Variations in architecture design and processor interconnections create interesting opportunities for performance improvements by limiting data movement distances (20). This application should benefit to some extent from this careful analysis.

6.3 Individual Protection

Individual Protection (IP) modeling software was designed as a practical tool for non-experts. The models provide sufficient fidelity and accuracy of analysis on garment fit, closures, and physical exertion. Mapping of Airflow for Performance Prediction uses mass, species, and energy conservation principles. Simplified physics-based models equations are incorporated for protection against chemical and biological threats. From a meeting conducted at the Edgewood Chemical Biological Center (ECBC), researchers involved in the project expressed that the computational runtime for IP SPM was not currently demanding or time consuming. However, if higher order modeling is employed for enhancing prediction and evaluation, then high performance computing systems can assist with the increased computational requirement. For instance, reference 21 lists reported computational fluid dynamics (CFD) performance improvements achieved with GPU assistance.

6.4 Collective Protection

Collective Protection (ColPro) and the Joint Expeditionary Collective Protection (JECP) are performing science and technology research to improve collective protection in multi-functional shelter materials and air purification systems, those that may be deployed rapidly. As with other SPMs, ColPro's binary computing requirements deal mostly with modeling and simulation. This can involve areas of CFD, especially in areas dealing with the Protective Entrance (PE) of tents and shelters, and extend to areas of computational chemistry when dealing with self-decontaminating materials. The ColPro SPM is currently under the direction of the Naval Surface Warfare Center (NSWC), Dahlgren Division.

The momentum of heterogeneous computing provides opportunities for large CFD simulations. Massively parallel GPU architectures can offer relief to slow simulation time while improving fidelity of simplified models. In reference 22, GPU platforms are leveraged for implementation of an incompressible 3-D Navier-Stokes solver to compute wind fields in

urban environments. Transport and dispersion of contaminants in urban areas are analyzed and computed on platforms equipped with GPU cards. The results in this paper show that a CFD-based urban dispersion model can be simulated in a few minutes, indicating a feasibility of real-time CFD problems. Moreover, the paper extends work to a multi-GPU setup. Multi-GPU scaling analysis of incompressible Navier-Stokes solvers shows performance dependence on problem size, where larger domains size improves scaling on multi-GPUs. This speed improvement on a multiple GPUs setup is observed when a minimal ratio of one CPU per each GPU device is satisfied.

7. Cost-benefit Analysis

Cost-benefit analysis is always a difficult thing to do, and in the world of determining accelerator-based computing, it certainly does not get any easier. In our studies of performance versus effort, one always has to be aware that getting 80% of a requirement might be easy, but it is that last 20% that can be very difficult. Economics might also come into play. Moreover, programming effort and time plays a greater role in reported performances as described in reference 23.

Here, theoretical computational power from an architecture view point is considered for analysis. A summary of raw performance, price, and power for GPUs and CPUs is organized for comparison. Theoretical arithmetic performance of 32-bit floating-point operations is calculated based on clock frequency and its respective compute elements. Maximum power dissipation in watts is reported for each processor unit as rated by the manufacturers. Listed prices reflect online retail value results obtained in April 2011. Processors from Nvidia, AMD, and Intel are analyzed where mid-range to high-end devices are chosen for each vendor. The graph of GFLOPS per dollar and GFLOPS per watt are illustrated in figures 4 and 5. Looking at performance and price, the high number of simple cores and mass-market demand gives compute power and cost advantage to graphics processors. In terms of power consumption, despite higher wattage requirement for high-end graphics cards, GPU power efficiency per watt surpasses CPU architectures.

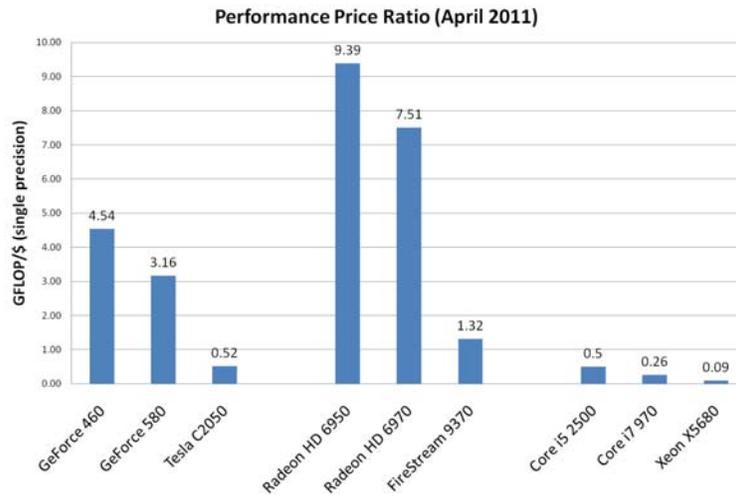


Figure 4. Floating-point performance per dollar.

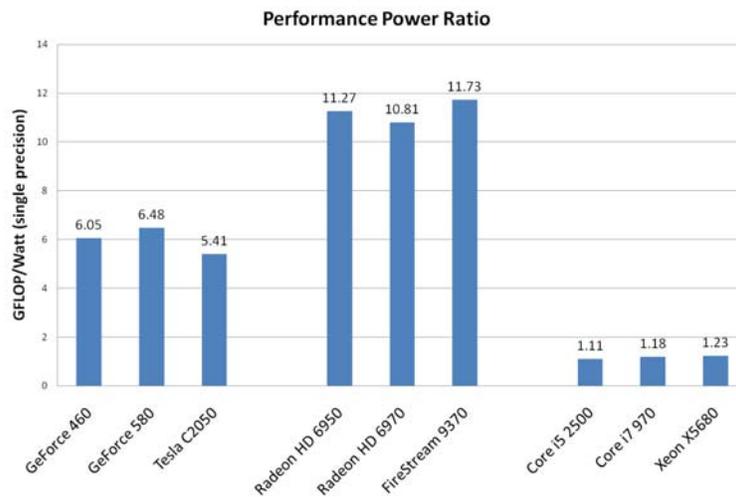


Figure 5. Floating-point performance per watt.

8. Future Trends

Trends toward increasing levels of heterogeneous computing at every level are unlikely to diminish anytime soon. Hybrid systems consisting of mixed computing resources range from mobile devices to large data centers. It is unclear what these end products may look like; vendors are continuously working on various combinations of traditional cores, GPUs, and even vector processing units on a single die. It is quite probable that there will not be a single mass-market-based approach but rather more than one answer for various markets.

Power-aware computing is also quickly growing in importance as a vast majority of future processing cores will be located inside of mobile or embedded devices. Issues with power in heterogeneous HPC systems and mobile devices manifest in different ways. For HPC systems, aggregated power to fuel systems with thousands of cores quickly scales. For example, the average electrical power consumption for the top five systems on the Top 500 list in November 2010 was 3.3 MW. This equates to \$2.89 million per year (assuming \$0.10/KWh). FPGAs remain under consideration for some deployed settings since they traditionally use much less power than CPU- or GPU-based devices.

Other techniques may also limit power consumption, both in hardware and software. Scaling precision from double to single usually results in lower energy consumption. Furthermore, the largest transistor allocation inside an Intel Nehalem die is devoted to cache-related logic circuits (distributed L1 and L2 cache inside each core, the shared L3 cache, and control). This implies that the majority of power is dissipated in the cache circuitry of the CPU. Coupled with the energy required to transfer data from main memory, data movement seems to impact power consumption at multiple levels. Therefore, it is expected that new research into optimizing this memory structure will ramp up significantly.

The Advanced Reduced Instruction Set (RISC) Machine, or ARM processor is a processor architecture that currently dominates the mobile and embedded computing markets. In terms of power efficiency, ARM processors have a historical advantage in that they evolved within low-power and battery-operated environments. The simplified RISC design coupled with pipelining allows for competitive performance from ARM chips when compared to other non-RISC designs such as x86-based architectures.

These trends highlight the need for active vigilance in the world of application design for HPC. It will be especially important for all applications, including the various codes running under the SPMs, to be efficient in terms of performance and power, as well as portable and deployable to a wide-range of architectures that will be fielded as part of an integrated and heterogeneous system. While advances in computer science, and in

particular, compiler and just-in-time computing strategies will no doubt alleviate some of the stress, it will be critical for software designers and application experts to deliver solutions that can be both efficient and portable to evolving systems.

A central question remains how to effectively capture the performance that will be available in these hardware developments. It is quite possible that the complexity of the hardware, and the fact that software development systems usually trail the hardware they are targeting, will require the continued significant investments of time in software engineering that we have seen in the past. A recognition that this requirement can be somewhat “abstracted” away is showing evidence with the growing interest in Domain Specific Languages (DSLs) (24). DSLs can target hybrid and complex systems by providing a language within the domain of interest for the researcher, whether it be signal processing, finite elements, or any other discipline. DSLs are not new. Some well-known examples of DSL languages, compilers, and execution environments include MATLAB and OpenGL. What is new is the idea of extending the compilers of DSLs and coupling to runtime systems for DSLs to be more hybrid-architecture aware. Runtime adjustment will undoubtedly be critical in some systems, such as those with complex domains (such as unstructured finite elements) or those seeking real-time performance. Smart, adaptable runtime infrastructures should be able to do dynamic load balancing and code migration inside complex systems. Significant investments in these DSLs have the potential to allow developers to code within their domains while effectively using the resources of the heterogeneous system. As changes occur to the configuration of the hardware, the DSL’s compiler back-end code generator and runtime systems can be updated accordingly, thus ensuring a retargetable and portable coding system.

9. Conclusions

Constrained by power issues, processing technology has advanced toward parallel computing in recent years. In this paradigm, embracing parallelism is the key for achieving higher performance. Regardless of computing platforms, software development needs to leverage the pervasive parallel architectures. Reaching an optimal performance level will require the knowledge of processor’s hardware characteristics such as memory hierarchy and execution model. Typically, mapping an application to processing hardware involves restructuring the algorithm and possibly the data.

Importantly, portability should not be ignored in the rapidly changing hardware environment. The availability of hardware diversity and options (x86, Fusion APU, Nvidia,

AMD GPU, Cell, SPARC64, ARM, and etc) are much greater in number today than in previous years. It would be ideal for an application to avoid locking into a particular architecture.

One size does not fit all in the modern computing era. In general, inherently sequential algorithms will opt for superscalar x86 microprocessors while data-parallel computations will prefer hardware multithreaded, SIMD architectures. Real world applications can exhibit a combination of serial and parallel execution flow where combined CPU/GPU systems would be a better candidate.

Driven by the mass market demand, computational power (FLOPS) has become affordable to everyday users. Previous generations of supercomputing capability are attainable in smaller form factors with manageable power. However, harnessing the underlying compute capability rests on the software developers. CBRN models should benefit from all of these hybrid system developments. This paper has demonstrated several examples of models that should do well now and scale well into the future. Level of effort and performance achieved, unfortunately, track on the same curve. Should software engineers be so inclined, it is quite possible that a dedicated CBRN library could be developed that factors in functionality across the SPMs, thus providing highly tuned software that targets an evolving hardware profile. In the long run, extending this to a complete language and runtime system using a DSL may be the best option.

References

- [1] Bhandarkar, D. Multi-core Microprocessor Chips: Motivation and Challenges; Intel Corp., May 2006.
- [2] Lowney, G. Why Intel is Designing Multi-core Processors; Intel Corp., May 2006.
- [3] Brassington, M. Semi-custom ASIC Technology Trends. *In Custom Integrated Circuits Conference, 1995. Proceedings of the IEEE 1995*, 1995.
- [4] Abdel-Aty-Zohdy, H.; Jozefowicz, P. Semicustom Design of ASICs: Silicon Chips-versus-FPGAs for Engineering Education. *In Circuits and Systems, Proceedings of the 36th Midwest Symposium on*, 1993.
- [5] Shires, D.; Park, S. J.; Henz, B.; Clarke, J.; Nguyen, L.; Kirk, K. *Asymmetric Core Computing for U.S. Army High Performance Computing Applications*; ARL-TR-4788; U.S. Army Research Laboratory: Aberdeen Proving Ground, MD, 2009.
- [6] Fatahalian, K.; Houston, M. GPUs: A Closer Look. *Queue* **2008**, *6*, 18–28.
- [7] Shires, D.; Mohan, R. An Evaluation of HPF and MPI Approaches and Performance in Unstructured Finite Element Simulations. *Journal of Mathematical Modelling and Algorithms* **2002**, *1*, 153–167; 10.1023/A:1020534421393.
- [8] The OpenMP API, <http://openmp.org> (accessed: 08/24/2011).
- [9] Akeley, K.; Hanrahan, P. Real-Time Graphics Architecture. Technical presentation, 2001.
- [10] Buck, I. Stream Computing on Graphics Hardware, Thesis, Stanford University, 2006.
- [11] BrookGPU: Introduction, <http://graphics.stanford.edu/projects/brookgpu/intro.html> (accessed: 02/17/2011).
- [12] AMD FireStream, http://en.wikipedia.org/wiki/AMD_FireStream#Software_Development_Kit, Accessed: 02/24/2011.
- [13] Nickolls, J.; Buck, I.; Garland, M.; Skadron, K. Scalable Parallel Programming with CUDA. *Queue* **2008**, *6*, 4053.
- [14] OpenCL: Introduction and Overview, 2010, Khronos Group.

- [15] Bowden, J. C. Application of the OpenCL API for Implementation of the NIPALS Algorithm for Principal Component Analysis of Large Data Sets. *IEEE International Conference on e-Science Workshops* **2010**, 0, 25–30.
- [16] Ji, X.; Cheng, T.; Li, D.; Wang, Q. Solving Large-scale Three-dimensional Heat Equations on CUDA. *2010 3rd International Conference on, In Advanced Computer Theory and Engineering (ICACTE)*, Vol. 2, 2010.
- [17] Richie, D.; Mandl, P. Using Parallel Processing on Graphics Processors to Accelerate Finite-Difference Time-Domain Algorithms for Electromagnetic and Seismic Applications; National Aeronautics and Space Administration: 2010 NASA Tech Brief, Embedded Technology.
- [18] Eller, P.; Cheng, J.-R.; Albert, D.; Liu, L. Performance Improvement of the 2-D Finite Difference Time Domain Acoustic Wave Simulation using Multiple GPUs. *In 27th Army Science Conference*, Orlando, FL, 2010.
- [19] Hurst, J. Parallelizing a Data Intensive Lagrangian Stochastic Particle Model Using Graphics Processing Units. Masters thesis, University of Colorado, 2010.
- [20] Shires, D.; Henz, B.; Mohan, R. Architecturebased Communication Improvements for Domain Decompositions. *In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Volume 4; PDPTA 02, CSREA Press: 2002.
- [21] Nickolls, J.; Dally, W. The GPU Computing Era. *Micro, IEEE* **2010**, 30 (2), 56–69.
- [22] Thibault, J. C. Implementation of a Cartesian Grid Incompressible Navier-Stokes Solver on Multi-GPU Desktop Platforms Using CUDA. Masters thesis, Boise State University, 2009.
- [23] Lee, V. W.; Kim, C.; Chhugani, J.; Deisher, M.; Kim, D.; Nguyen, A. D.; Satish, N.; Smelyanskiy, M.; Chennupati, S.; Hammarlund, P.; Singhal, R.; Dubey, P. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. *SIGARCH Comput. Archit. News* **2010**, 38, 451–460.
- [24] Hanrahan, P. Domain-Specific Languages for Heterogeneous GPU Computing. NVIDIA Technology Conference, 2009.

List of Symbols, Abbreviations, and Acronyms

2-D	two dimensional
3-D	three dimensional
ALU	arithmetic logic units
APIs	application programming interfaces
APP	accelerated parallel processing
APUs	accelerated processing units
ARL	U.S. Army Research Laboratory
ASICs	application specific integrated circuits
CA	Contamination Avoidance
CAL	compute abstraction layer
CBRN	chemical, biological, radiological, and nuclear
CFD	computational fluid dynamics
CISD	Computation and and Information Sciences
ColPro	collective protection
CUDA	compute unified device architecture
DSLs	domain specific languages
DSPs	digital signal processors
DTRA	Defense Threat Reduction Agency
ECBC	Edgewood Chemical Biological Center
FDTD	finite-difference time-domain
FFT	fast Fourier transform
FPGAs	field programmable gate arrays
GPUs	graphics processing units
HPC	high performance computing

IP	individual protection
JECP	Joint Expeditionary Collective Protection
JSTO	Joint Science and Technology Office
LES	large-eddy simulation
LLVM	low level virtual machine
LPDMs	Lagrangian particle dispersion models
LSM	Lagrangian stochastic model
MPI	message passing interface
NCAR	National Center for Atmospheric Research
NSWC	Naval Surface Warfare Center
OpenCL	open Computing Language
OpenGL	open graphics library
PE	protective entrance
RISC	advanced reduced instruction set
SDK	software development kit
SIMD	single instruction multiple data
SPAMs	system performance models
SPMD	single program-multiple data
SRAM	static random access memory
SSE	Streaming SIMD Extension
VLIW	very long instruction word

NO. OF COPIES	ORGANIZATION
1 ELEC	ADMNSTR DEFNS TECHL INFO CTR ATTN DTIC OCP 8725 JOHN J KINGMAN RD STE 0944 FT BELVOIR VA 22060-6218
1	US ARMY RSRCH DEV AND ENGRG CMND ARMAMENT RSRCH DEV & ENGRG CTR ARMAMENT ENGRG & TECHNOLGY CTR ATTN AMSRD AAR AEF T J MATTS BLDG 305 ABERDEEN PROVING GROUND MD 21005-5001
1	US ARMY INFO SYS ENGRG CMND ATTN AMSEL IE TD A RIVERA FT HUACHUCA AZ 85613-5300
1	US ARMY RSRCH LAB ATTN RDRL HRT M C GAUGHAN 12423 RESEARCH PARKWAY ORLANDO FL 32826
1	US GOVERNMENT PRINT OFF DEPOSITORY RECEIVING SECTION ATTN MAIL STOP IDAD J TATE 732 NORTH CAPITOL ST NW WASHINGTON DC 20402
5	US ARMY RSRCH LAB ATTN RDRL CIH C D SHIRES ATTN RDRL CIH C J CLARKE ATTN RDRL CIH C S PARK ATTN RDRL CIH R NAMBURU ATTN RDRL CIH S L BRAINARD ABERDEEN PROVING GROUND MD 21005
1	DIRECTOR US ARMY RSRCH LAB ATTN RDRL ROE V W D BACH PO BOX 12211 RESEARCH TRIANGLE PARK NC 27709
3	US ARMY RSRCH LAB ATTN IMAL HRA MAIL & RECORDS MGMT ATTN RDRL CIO LL TECHL LIB ATTN RDRL CIO LT TECHL PUB ADELPHI MD 20783-1197