Architecting Graphics Processors for Non-Graphics Compute Acceleration

Tor M. Aamodt
Dept. of Electrical and Computer Engineering
University of British Columbia, Vancouver, Canada
aamodt@ece.ubc.ca

Abstract

This paper discusses the emergence of graphics processing units (GPUs) that contain architecture features for accelerating non-graphics (or GPGPU) applications. It provides an introduction for those interested in undertaking research at the intersection of manycore computing and GPU architecture. First, the motivation for using GPUs for non-graphics processing rather than developing specialized hardware is outlined. Then, the features of GPU architectures and related programming models for non-graphics computing on GPUs are briefly described. Finally, some recent research and architecture challenges related to accelerating non-graphics workloads on GPUs are summarized.

1 Introduction

Trends in semiconductor scaling [5, 8] coupled with increasing design complexity have encouraged hardware manufacturers to once again explore parallel processing. One class of single chip parallel processor growing in prominence is the *graphics processing unit* (GPU). GPUs dedicate far more silicon area to arithmetic logic units (ALUs) than do general purpose processors. This can make GPUs far more cost effective in terms of silicon area when running highly parallel software.

In little over a decade, GPUs have rapidly evolved away from fixed-function pipelines towards highly programmable pipelines. In 2001 the first GPUs with programmable vertex shaders appeared. The motivation for this programmability, was not general purpose computing, but rather the increasing complexity of supporting various operating modes as graphics APIs evolved, along with the desire to enable application developers to provide custom visual effects [30]. While it is possible to achieve programmability even on fixed-function GPU pipelines by using multi-pass rendering techniques [40], directly enabling programmability in hardware improves performance where programmability is required.

The addition of programmability attracted the interests of many researchers interested in accessing the computing power in GPUs for uses other than rasterization based graphics (e.g., [41]). By 2002 programmable pixel shaders were introduced into GPUs. By 2005 hardware appeared (the Xbox 360) that unified the vertex and pixel shaders. With the growing interest in using GPUs for non-graphics computing, sometimes referred to as GPGPU (for General Purpose computing on GPUs) [1] the GeForce 8800 GTX introduced in November 2006 brought explicit hardware support for general-purpose computing to the GPU. This included the ability to write to arbitrary memory locations in graphics memory and small on-chip shared memories to reduce off-chip bandwidth demands and facilitate communication between threads [31].

The growing importance of GPUs is illustrated by announced plans to integrate GPU and GPU in future multicore architectures [6]. Moreover, the architecture of general-purpose multicore processors and GPUs are converging: Highly multithreaded GPUs are become more general-purpose and multicore processors are supporting ever-greater degrees of parallelism [34].

The rest of this paper is organized as follows: Section 2 outlines some arguments for why computer architects should be interested in influencing GPU architectures to be more general purpose than they are today. Section 3 discusses contemporary GPU architectures focusing on their support for non-graphics computing. Section 4 discusses the usage of GPUs for non-graphics workloads. Section 5 describes some challenges and recent research on architecting GPUs for non-graphics computing. Finally, Section 6 provides some suggestions for where to look for additional information.

2 Why Architect GPUs for Computing?

An important principle affecting hardware development is economy of scale. To recover the large non-recurring costs of developing a chip of similar complexity as a GPU (which today, can be over $0.5 billion), many chips must be sold. For a specialized hardware design serving a relatively small market to be competitive, it ought to offer significantly better performance per unit of recurring cost (e.g., for similar silicon area). For instance, the designers of the application specific Anton supercomputer, developed for accelerating molecular dynamics simulations, have presented data indicating that their specialized architecture will deliver roughly two orders of magnitude higher performance than a comparable cluster of 512 AMD Opteron cores [14].

While this result is impressive, many algorithms have been shown to obtain similar performance improvements over CPUs using GPUs [32]. It is too early to judge the commercial success of Anton in particular, but parallel computing has seen many commercial failures in the past. On the other hand, the market for interactive entertainment (i.e., video games) appears quite willing to repeatedly subsidize the costs of developing massively parallel hardware in the form of GPUs. Thus, if an application can be made to effectively utilize the GPU then it can benefit from the large investment in development costs underwritten by the traditional market for these devices. In contrast a specialized processor must amortize development costs over a potentially smaller market. On the other hand, a more general purpose processor must provide as good performance per unit cost as a GPU for interactive entertainment for it to command a large share of that market. Finally, GPUs are interesting to study simply because, given their highly programmable and multithreaded nature, GPUs can provide insight into how to design future manycore processors.

If one accepts the arguments above, the first question, for any given application becomes, "Can this application make effective use of existing GPUs?" Many applications with abundant parallelism do not achieve the full benefit of available GPU resources [10]. For such applications, the question of interest to architects is, "Would changes to the GPU architecture enable these applications to benefit as well and also benefit, or at least not reduce performance/cost of graphics?" In the next section we examine contemporary GPU architectures.

3 GPU Computing Architectures

This section briefly reviews modern GPU architecture features relevant to non-graphics applications. The focus is on features of current and announced commodity GPUs [39, 9, 28].

Contemporary GPUs, such as the GeForce 200 Series from NVIDIA or the ATI Radeon HD 4800 Series from AMD, employ a unified shader architecture. GPUs with unified shader architectures have programmable "cores" that can alternate among several of the tasks (e.g., vertex, geometry and pixel shading) in the graphics rendering pipeline of application programming interfaces such as OpenGL [2] and DirectX [3]. In addition to providing this flexibility, these GPUs also allow the same programmable cores to be used for non-graphics applications.

In this paper, the term *core* refers to one single-instruction, multiple data (SIMD) pipeline (equivalent to a Streaming Multiprocessor in NVIDIA terminology [31]). The use of SIMD hardware improves performance per unit cost for data parallel applications. To support shader programs containing data dependent control flow, modern GPUs use a variant of SIMD execution sometimes called single instruction multiple thread (SIMT) execution [31]. In SIMT execution, a set of individual threads are grouped together into a SIMD unit called a warp (in NVIDIA terminology) or a wavefront (in AMD terminology). The threads in a warp (or wavefront) execute in lock step. When they encounter a branch, some threads may execute the branch as "taken" while others execute the branch as "not-taken". The hardware has support to mask off the execution of threads following one path while allowing threads following the other path to complete until a control flow join point. Such support can take the form of a stack of SIMD processing element activity masks plus special instructions for updating the stack after a conditional branch [29, 7].

Each core employs fine-grained multithreading (also known as barrel processing [44]) to interleave execution of multiple warps (or wavefronts) on the SIMD pipeline on a cycle by cycle basis. This enables threads to partly hide long memory access and complex arithmetic operation latencies [30]. Fine-grained multithreading also helps overlap multiple memory requests to achieve a form of parallelism called memory level parallelism [20].

One of the most valuable resources for GPUs is pin bandwidth to off-chip DRAM. To make effective use of this resource, GPUs reorder memory requests to reduce delays due to DRAM page activate and precharge overheads [35, 16]. In graphics workloads for interactive entertainment applications, a significant fraction of off-chip memory accesses are for texture mapping. Texture mapping [13], used to "paint a decal" onto a 3D surface, is an integral part of interactive graphics in contemporary GPUs. A common technique for reducing off-chip bandwidth demands is the use of a hardware structure known as a cache [22]. The limited locality in the large volume of texture data is captured by small caches [21], but leaves relatively low hit rates (e.g., Montrym and Moreton indicate a design target texture cache hit rate of only 90% [35]). To increase the texture cache hit rate further would require unreasonably large caches.

Prior to the introduction of the GeForce 8800 GTX in 2006, the GPU programming model placed restrictions on the locations a shader program could *write*. This restriction was a natural consequence of the way the graphics pipeline is organized—pixel shaders may read arbitrary memory locations (called a gather operation) while performing texture filtering, but write the resulting color and depth information to a fixed location determined by the pixels location on the screen. Early GPGPU developers invented techniques to work around this limitation [12]. However, recent GPUs overcome this limitation more efficiently by directly enabling hardware to support writes to arbitrary memory locations (also known as a *scatter* operation). Hardware support for scatter requires an interconnection network between the pixel shaders and the raster operation units that write data to DRAM, and this interconnect is not useful in current raster based graphics pipelines [25, 11].

One of the challenges of obtaining high performance using GPUs is the need for the software developer to take steps to make effective use of on-chip storage. The raw computing power of contemporary GPUs relative to the off-chip bandwidth is significant—e.g., over a given interval of time the NVIDIA GeForce 285 GTX [38] can perform close to seven single-precision floating-point operations for every single byte of data transferred to or from off-chip graphics DRAM. A related concern is the need to amortize the overhead of transferring data from the CPU to the GPU [23]. In addition to providing registers that can be accessed from within a single thread, the GeForce 8 series also includes a small, 16KB on-chip random access memory per streaming multiprocessor, called *shared memory* [39]. In a graphics pipeline, such on-chip storage could potentially be used for buffering between rendering stages [43]. In the context of non-graphics computation, such on-chip memory is exploited by employing program transformations such as tiling [47], which are traditionally used to improve cache locality in general purpose computers.

GPUs will likely continue their rapid evolution and it seems likely that future GPUs will provide even more flexible programming models than current GPUs. For example, William Mark argues that future GPUs may benefit from support for cache-coherence [33]. Cache coherence is a technique that ensures no processor has a different view of memory. Supporting cache coherence increase the amount of traffic between cores, potentially limiting overall throughput. Mark argues supporting cache coherence in graphics hardware makes sense from a graphics perspective since more advanced rendering algorithms than those in use in GPUs today often require irregular data structures (e.g., trees) which are not well supported in current GPU architectures due to their requirement of explicitly managing off-chip data transfers [33]. The recently announced Larrabee architecture from Intel will have coherent caches [28]. Larrabee also replaces many graphics specific fixed-function hardware units, including rasterization, with software. One of the fixed-function units that remains in Larrabee is hardware support for texturing. Next we consider programming models.

4 Programming Models for GPU Compute

The programming models used for GPU Computing have evolved rapidly. Early approaches leveraging traditional graphics APIs such as OpenGL or DirectX are summarized by Owens et al. [24]. Today, GPU hardware vendors supply programming APIs, including CUDA [39], and Brook+ [9] that are extensions of C/C++ which greatly reduce the difficulty of learning these languages for experienced software developers. Furthermore, the introduction of the OpenCL standard [27], which is similar in many ways to CUDA and Brook+, indicates that there is strong interest in developing a common framework to ensure software portability. Since CUDA is currently the most widespread, this paper describes a few of its important features. A more complete introduction can be found elsewhere [37, 39].

A CUDA application starts by running on the CPU. Before doing any work on the GPU, the application may perform operations such as reading input from a file. It will typically transfer some data from the CPU to the GPU via a system interconnect such as PCI Express. Then the application will initiate computation on the GPU by calling a kernel function. While the GPU is running the kernel, the CPU may continue performing other computations. The kernel runs numerous threads in parallel on the GPU. A group of threads (32 on current NVIDIA GPUs) are organized into a warp, and multiple warps are grouped together into a cooperative thread array (CTA) which is also sometimes referred to as a block. Multiple CTAs are grouped together in a grid. Threads within a CTA can communicate with each other via the 16KB on-chip shared memory and can synchronize using barrier synchronization. After a kernel finishes computing results on the GPU, the CPU may initiate transfer of the results of the computation back to the CPU's memory. Next we consider architecture challenges for GPU computing.

5 Architecture Challenges

In this section we outline challenges related to supporting non-graphics computing on graphics processors. Sections 5.1 and 5.2 consider control flow and memory access. Section 5.3 considers architecture evaluation methodologies. From a software developer's standpoint, the first challenge with any parallel system is Amdahl's Law [19], a corollary of which is performance improvement is bounded by one over the fraction of the application that cannot be run in parallel. In the first three sections below, we are concerned with applications that exhibit sufficient parallelism that they can make good use of a GPU. In Section 5.4 we consider the potential benefits of integrating a GPU and GPU on a single chip.

5.1 Control Flow

The use of SIMT execution on GPUs supported with stack based reconvergence mechanisms, mentioned in Section 3, can incur significant performance overheads. The stack based reconvergence mechanism implemented in the Chap processor (developed at Lucasfilm, Ltd.) is described by Levinthal and Porter [29]. The basic idea is to selectively enable or disable SIMD processing elements based upon the outcome of branches executed for that processing element. A slightly more general approach, that uses the immediate post-dominator of a branch as a control flow reconvergence point is the PDOM mechanism described by Fung et al. [18, 17]. When individual threads within a warp or wavefront encounter different branch outcomes (e.g., some threads see the branch as "taken" other see the branch as "not-taken") an entry is created on the stack so that execution of one control flow path (the "taken" path) can proceed before the other (the "not taken" path). This serializes the execution of the two paths and leads to a loss in performance, a condition known as branch divergence.

One proposal for overcoming the branch divergence problem, proposed for stream processors, is to reorganize the computation at the software level (with hardware support) using conditional streams [45]. Each conditional branch generates a new stream of data which is input into a new kernel. While CUDA and Brook+ support the notion of streams, neither currently provides support for conditional streams. Others have proposed augmenting each SIMD processing element with its own instruction sequencer to directly enable temporary MIMD operation [26, 42]. Finally, a more recent proposal is to modify the hardware to allow warps (or wavefronts) to be reorganized dynamically during execution—an approach known as *dynamic warp formation* [18, 17]. Each proposal has limitations: Conditional streams include overheads for creating multiple kernels which can in some cases be hidden, but may be unavoidable with deeply nested control flow. Local sequencers introduce hardware overheads and their limited instruction capacities may result in instruction bandwidth limitations. Finally, dynamic warp formation is sensitive to the warp scheduling algorithm and the number of warps per CTA (fewer warps per CTA is worse). Another alternative is for hardware developers to simply decrease the number of threads per warp in future GPUs as demand for higher performance on control flow intensive applications increases.

5.2 Memory Operations

Memory operations can present a severe challenge for GPUs. We consider both off-chip and on-chip communication on the GPU as well as the link between the CPU and the GPU. Generally, there are two quantities of interest when considering memory access operations. The first is *bandwidth*, measured as the total amount of data that can be accessed over an interval of time. The second is *latency*, measured as the average time to complete a single operation. Since GPUs are fundamentally designed to hide latency by utilizing fine-grained multithreading, the primary concern is bandwidth. This is quite different than the case for single thread general purpose computing where the focus has primarily been on latency reduction using techniques such as prefetching.

The foremost concern is the cost of off-chip memory accesses. The challenge here is that, with each successive process technology generation, the number of I/O pins per chip does not increase as quickly as the number of transistors that can be integrated onto a single chip. Compounding this, modern DRAM technology is organized in such a way that the order that locations are accessed in the memory affects the time it takes to access data. This issue is described in more detail by Yuan and Aamodt [49] who provide a more detailed survey of recent research than possible here. A challenging problem is developing DRAM access scheduling algorithms that provide improved performance. Existing approaches essentially employ greedy scheduling algorithms. A open question is whether less greedy approaches can provide better performance and if so what form the solutions would take.

A related challenge is that, while individual threads in a warp can access arbitrary memory locations (scatter or gather), there is generally a penalty for doing so. This results from the stalling of the pipeline due to the need to issue multiple memory requests to satisfy each individual thread's access request. When individual threads within a warp access contiguous locations in memory, a single larger request can be sent.

Finally, on-chip communication also presents challenges. The local on-chip shared memory is typically banked to increase bandwidth, but does not have many read-write ports resulting in "bank conflicts" where two or more threads from a single warp want to access a resource at the same time, but only one is allowed (i.e., a structural hazard), even though they are accessing different data. While software tuning tools exist to help the programmer identify this issue, it may not be easy to eliminate these bank conflicts by making simple software changes.

5.3 Evaluation Methodologies

How does one go about evaluating a novel architecture idea they believe will improve performance? In the early days of computing, design decisions were made based upon low level performance measures such as the time to execute a single instruction [22]. Such techniques are highly ineffective for modern computers due to the large amount of parallelism employed (the execution of individual instructions is overlapped on all modern microprocessors). Since the 1980's and the *reduced instruction set computing* (RISC) movement, the computer industry has firmly embraced a quantitative methodology in which design decisions are informed by careful analysis of the impact on overall system performance as measured by the average execution time of real software applications [22]. In practice, this now translates to a methodology of evaluating architecture proposals using C/C++ based cycle level detailed performance simulators [36].

Computer system performance is governed by the equation [22]:

$$\text{Execution Time} = \frac{\text{IC} \times \text{cycle time}}{\text{IPC}} \tag{1}$$

This equation states that the total time it takes to run a program (Execution Time) is the product of the instruction count (IC) and the clock cycle time (cycle time) divided by the average instructions executed per cycle (IPC). While improvements in process technology and circuit design techniques can improve the clock cycle time, architecture modifications may impact all three of these components. C/C++ based cycle level simulators measure the instruction count and instructions per cycle but do not provide information on the cycle time. Furthermore, these simulators (even those used in industry) are typically only accurate to within at most $\approx$5% of the actual hardware (and often they are less accurate). The benefit of using such simulators is they abstract away details that have a minor impact upon performance resulting in a modeling methodology in which it is easier to rapidly evaluate design alternatives.

The GPGPU-Sim simulator, developed at the University of British Columbia, provides a framework for undertaking research on architecture for non-graphics computing on graphics processors [10]. GPGPU-Sim (which can be downloaded from `http://www.gpgpu-sim.org`), can run unmodified CUDA applications and enables architecture researchers to quantitatively evaluate the impact of their design proposals on average instructions per cycle. Other simulators focus on graphics workloads [15].

An important consideration that governs whether any proposed architectural feature will be used in practice is whether the benefits (e.g., performance gain) are worth the cost (e.g., silicon area). While building a hardware prototype is the most accurate way to estimate the area costs, in many cases the costs of certain hardware structures can be estimated using tools such as CACTI [4].

5.4 Heterogeneous Architectures

Recently, there has been much interest in the potential to integrate a CPU and GPU onto a single chip [6]. One argument in favor of this is that communication overheads between the CPU and GPU limit potential performance, as suggested by Amdahl's Law. Applications that have significant parallelism and do not require significant amounts of communication between CPU and GPU would have little reason to see a benefit from this arrangement. For this reason, a recent study [48] explored the performance potential for a more "general purpose" set of applications. The results of the study highlight that when parallelism is not extreme, lowering the read-after-write latency of the parallel cores is important. In addition, it shows that increasing the bandwidth and lowering the latency between the cores beyond values attainable today using PCI Express appear to have small effects. Since it is all but certain that hardware manufacturers will build heterogeneous systems, more research in the area of such systems is warranted.

6 Where to learn more

This paper has provided a very brief introduction to the topic of architecting GPUs for non-graphics applications. For readers interest in learning more, there are several sources to turn to.

Owens et al. [24] survey the state of general purpose computing on GPUs as it existed just prior to the introduction of CUDA. They summarize the motivations for using GPUs, provide an overview of GPU hardware features, along with GPGPU programming models, tools, techniques and applications. Nickolls et al. [37] provide a concise introduction to CUDA. Lindholm et al. [31] provide an overview of the NVIDIA Tesla GPU architecture. Garland et al. [32] describe their experience writing several CUDA applications and the performance achieved versus CPU-only execution. Most research on GPU architectures has historically appeared at graphics specific conferences such as ACM SIGGRAPH and Graphics Hardware. There have also been several publications at computer architecture conferences (ISCA, MICRO, HPCA, Supercomputing, and others). In particular, research on stream processors [43, 46] is highly relevant to architecting GPUs for non-graphics applications.

7 Acknowledgements

Some of the research described in this paper was supported by the Natural Sciences and Engineering Research Council of Canada. I would like to thank the students in my research group for their excellent work (whether it is referenced in this paper or not).

References

[1] http://www.gpgpu.org.

[2] http://www.opengl.org.

[3] http://www.microsoft.com/windows/directx/default.mspx.

[4] http://www.hpl.hp.com/research/cacti/.

[5] ITRS 2008 Update. http://www.itrs.net/Links/2008ITRS/Home2008.htm.

[6] Advanced Micro Devices, Inc. The future is fusion. http://sites.amd.com/us/Documents/AMD_fusion_Whitepaper.pdf, 2008.

[7] Advanced Micro Devices, Inc. *R700-Family Instruction Set Architecture*, 1.0 edition, 2009.

[8] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *ISCA*, pages 248–259, 2000.

[9] AMD Inc. *ATI Stream Computing User Guide*, 2009.

[10] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *IEEE Int'l Symp. on Perf. Analysis of Systems and Software*, pages 163–174, April 2009.

[11] D. Blythe. The direct3d 10 system. In *SIGGRAPH*, pages 724–734, 2006.

[12] I. Buck. Taking the Plunge into GPU Computing. In M. Pharr, editor, *GPU Gems 2*, chapter 32, pages 509–519. Addison Wesley, March 2005.

[13] E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah, 1974.

[14] D. Shaw et al. Anton, a special-purpose machine for molecular dynamics simulation. In *ISCA*, pages 1–12, 2007.

[15] V. del Barrio, C. Gonzalez, J. Roca, A. Fernandez, and R. Espasa. ATTILA: A cycle-level execution-driven simulator for modern GPU architectures. *IEEE Int'l Symp. on Perf. Analysis of Systems and Software*, pages 231–241, 19-21 March 2006.

[16] R. E. Eckert. Page Stream Sorter for DRAM Systems, United States Patent 7,376,803, May 2008.

[17] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *MICRO*, pages 407–420, 2007.

[18] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware. *ACM Trans. Architec. Code Optim. (TACO)*, 6(2), 2009.

[19] G. M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conf. Proc. vol. 30*, pages 483–485, 1967.

[20] A. Glew. MLP Yes! ILP No! In *Wild and Crazy Ideas Session, 8th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[21] Z. S. Hakura and A. Gupta. The design and analysis of a cache architecture for texture mapping. In *ISCA*, pages 108–120, 1997.

[22] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th edition, Sept. 2006.

[23] Ian Buck et al. Brook for GPUs: stream computing on graphics hardware. In *SIGGRAPH*, pages 777–786, 2004.

[24] J. Owens, et al. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.

[25] G. S. Johnson, J. Lee, C. A. Burns, and W. R. Mark. The irregular z-buffer: Hardware acceleration for irregular data structures. *ACM Trans. Graph.*, 24(4):1462–1482, 2005.

[26] K. Sankaralingam, et al. Universal Mechanisms for Data-Parallel Architectures. In *MICRO*, page 303, 2003.

[27] Khronos Group. *The OpenCL Specification*, 1.0.43 edition, May 2009.

[28] Larry Seiler, et al. Larrabee: a many-core x86 architecture for visual computing. In *SIGGRAPH*, pages 1–15, 2008.

[29] A. Levinthal and T. Porter. Chap - a SIMD graphics processor. In *SIGGRAPH*, pages 77–82, 1984.

[30] E. Lindholm, M. J. Kligard, and H. P. Moreton. A user-programmable vertex engine. In *SIGGRAPH*, pages 149–158, 2001.

[31] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, March-April 2008.

[32] M. Garland, et al. Parallel computing experiences with cuda. *Micro, IEEE*, 28(4):13–27, July-Aug. 2008.

[33] W. Mark. Future graphics architectures. *Queue*, 6(2):54–64, 2008.

[34] W. R. Mark. Future visualization platform. Panel Presentation at Visualization 2004, Oct. 2004.

[35] J. Montrym and H. Moreton. The GeForce 6800. *IEEE Micro*, 25(2):41–51, 2005.

[36] S. Mukherjee, S. Adve, T. Austin, J. Emer, and P. Magnusson. Performance simulation tools. *Computer*, 35(2):38–39, Feb 2002.

[37] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[38] NVIDIA Corp. GeForce GTX 285. http://www.nvidia.com/object/product_geforce_gtx_285_us.html.

[39] NVIDIA Corp. *NVIDIA CUDA Programming Guide*, 2.2 edition, 2009.

[40] M. S. Peercy, M. Olano, J. Airey, and P. J. Ungar. Interactive multi-pass programmable shading. In *SIGGRAPH*, pages 425–432, 2000.

[41] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. In *SIGGRAPH*, pages 703–712, 2002.

[42] R. Krashinsky et al. The vector-thread architecture. In *ISCA*, page 52, 2004.

[43] S. Rixner, et al. A Bandwidth-Efficient Architecture for Media Processing. In *MICRO*, pages 3–13, 1998.

[44] J. E. Thornton. Parallel operation in the control data 6600. In *AFIPS Proc. FJCC*, volume 26, pages 33–40, 1964.

[45] U. Kapasi, et al. Efficient conditional operations for data-parallel architectures. In *MICRO*, pages 159–170, 2000.

[46] W. Dally, et al. Merrimac: Supercomputing with Streams. In *ACM/IEEE Conf. on Supercomputing*, 2003.

[47] M. Wolfe. More Iteration Space Tiling. In *ACM/IEEE Conf. on Supercomputing*, pages 655–664, 1989.

[48] H. Wong and T. M. Aamodt. The Performance Potential for Single Application Heterogeneous Systems. In *8th Workshop on Duplicating, Deconstructing, and Debunking*, June 2009.

[49] G. L. Yuan and T. M. Aamodt. A Hybrid Analytical DRAM Performance Model. In *Proc. 5th Workshop on Modeling, Benchmarking and Simulation*, June 2009.