A Comparison of Modern GPU and CPU Architectures: And the Common Convergence of Both

Jonathan Palacios and Josh Triska

March 15, 2011

1 Introduction

Graphics Processing Units (GPUs) have been evolving at a rapid rate in recent years, partly due to increasing needs of the very active computer graphics development community. The visual computing demands of modern computer games and scientific visualization tools have steadily escalated over the past two decades (Figure 1). But the speed and breadth of evolution in recent years has also been affected by the increased demand for these chips to be suitable for general purpose parallel computing as well as graphics processing.

In a campaign that has been perhaps most aggressively pushed by the company NVIDIA (one of the leading chip designers), GPUs have moved closer and closer to being general purpose parallel computing devices. This movement began in the computer graphics software research community around 2003 [15], and at the time was called General Purpose GPU (GPGPU) computing [16, 20, 8]. Using graphics APIs not originally intended or designed for non-graphical applications, many data parallel algorithms, such as protein folding, stock options pricing Magnetic Resonance Image (MRI) reconstruction and database queries, were ported to the GPU.

This prompted efforts by chip designers, such as NVIDIA, AMD and Intel to produce architectures that were more flexible with more general purpose components (perhaps the most notable change has been the *unified shader model*).

This blurring of roles between the CPU (which, in the past, has been considered the primary general purpose processor) and the GPU has caused some interesting dynamics, the full ramifications of which are not yet clear. GPUs are becoming much more capable processors, and unlike CPUs, which are struggling to find ways of improving speed, their raw computational power increases dramatically every generation, as they add more and more functional units and "processing cores". CPUs are also adding cores (most CPUs are now at least dual-core), but at a much slower rate. Still, CPUs are much more suited to certain tasks where there is less potential for parallelism. In any case, GPUs and CPUs seem to be engaged in some sort of co-evolution.

Figure 1: On the left is a screenshot of the game System Shock released in 1999. The improvement in graphical quality between this and the 2008 game, Crysis (right), illustrates the increasing demands of the computer graphics development community.

In this work, we hope to explore how GPUs differ from CPUs, how they are evolving to be more general purpose, and what this means for both classes of processors. The remainder of the paper is organized as follows: In Section 2 we briefly cover the history of the GPU to give some context for it's role in computing. We then examine what the chip structure looks like for both GPUs and CPUs and how the former has evolved in Section 3. In Section 4 we cover the data pipeline in the GPU, and how it works for graphics and general purpose computing. Section 5 examines the GPU memory hierarchy, and Section 6, the instruction set. Finally, we cover some applications of general purpose GPU computing in Section 7, and give our conclusions in Section 8.

2 History of the GPU

The modern GPU has developed mostly in the last fifteen years [19]. Prior to that, graphics on desktop computers were handled by a device called a video graphics array (VGA) controller. A VGA controller is simply memory controller attached to some DRAM and a display generator. It's job is essentially to receive image data, arrange it properly, and feed it to a video device, such as a computer monitor. Over the 1990s, various graphics acceleration components were being added to the VGA controller as semiconductor technology increased, such as hardware components for rasterizing triangles, texture mapping, and simple shading. And in 1999, NVIDIA released the "GeForce 256" and marketed as the world's first GPU. There were other graphics acceleration products on the market at the time (by NVIDIA itself, as well as other companies such as ATI, and 3dfx), but this represented the first time the term GPU was used.

The first GPUs were fixed-function throughput devices, essentially designed to take a set of triangle coordinates, and color, texture, and lighting specifications, in order produce an image efficiently. Over time, GPUs have become more and more programmable, so that small programs (called shaders) can be

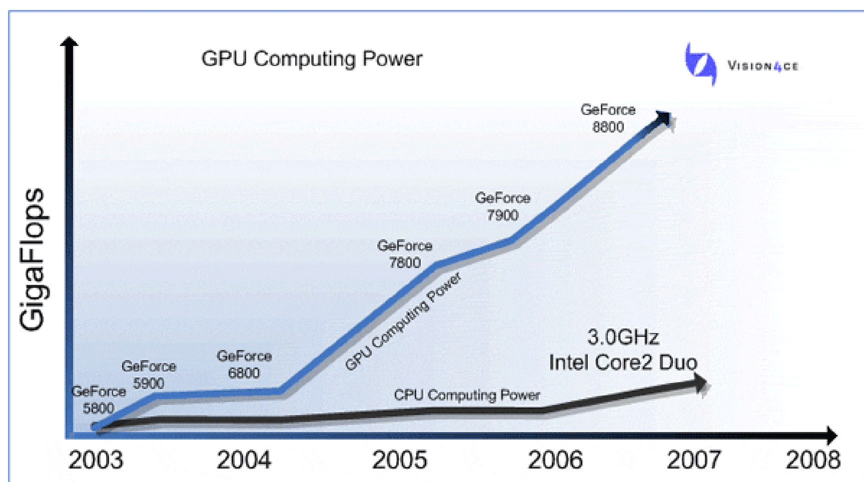

Figure 2: NVIDIA GPU raw computing power in GigaFlops relative to time alongside Intel CPU computing power. AMD (not shown), the other major GPU designer, has made gains similar to that of NVIDIA.

run for each triangle, vertex and pixel processed, vastly expanding the kinds of visual effects that can be generated quickly (reflections, refractions, lighting detail). This is in addition to the huge increase in raw parallel computing power (Figure 2), dwarfing the CPUs gains in this area, although the two technologies are clearly optimized for different applications.

Increasingly, logic units dedicated for special purposes, such as vertex processing or pixel processing, have given way to more general purpose units that can be used for either task (the unified shader model standardized the instruction set used across vertex and pixel shaders, so that only one type of logical unit was needed). Computation has also become more precise over time, moving from indexed arithmetic, to integer and fixed point, to single precision floating point, and most recently double precision floating point storage and operations have been added. GPUs have essentially become massively parallel computing devices with hundreds of cores (ALUs) and many more threads.

In the past five years, the instruction set and memory hardware have also been expanded to support general purpose programming languages such as C and C++. So it is clear, that, although they may still have some limitations relative to CPUs (more restrictive memory access, etc.), GPUs are becoming more and more applicable to a great many more purposes than those for which they were originally intended; and this trend is likely to continue [5].

It is worth noting (since it has affected the development of and the literature on the GPU so dramatically) that although when the market started there were upwards to twenty companies competing with each other, the field has since narrowed significantly. Now only Intel, NVIDIA, and AMD (formally

AMD's GPU division was the company ATI) are serious competetors in the GPU market, and only NVIDIA and AMD develop high-end graphics cards [10].

3 Modern Chip Structure

GPU Chip layouts have undergone a great deal of reorganization and re-purposing over the past few years, trending towards general purpose computing; this is in addition to the huge increase in functional units. CPUs to a lesser degree, likely because they must maintain the many advances they have made in reducing latency (at a cost in die real-estate), and also because of the need to maintain their compatibility legacy. However, CPUs have also been adding more cores.

3.1 Some GPU Chip Layout History

In this section, we cover an abridged history of the GPU chip structure (focusing on NVIDIA chips), to give some context for more recent developments. The earliest GPUs were very special purpose devices that performed fixed function tasks like transformation and lighting, texture mapping, and rasterization of triangles into pixels. From the programmers perspective, one could set a few options, and pass them lists of triangles, vertices, textures and virtual lights and cameras to be processed. The story became much more interesting when GPUs started to become programmable (2000-2004) [19].

NVIDIA's 7800 (Figure 3) was one of the earlier GPUs to have both pixel and vertex shaders. However, the processor "cores" (which contained the functional units) were still divided up into vertex processors and fragment (pixel) processors. This limited the flexibility of the chip somewhat, because there were a fixed number of components that could be dedicated to particular tasks.

The unified shader model was introduced in 2008, after which vertex and fragment shaders relied on the same instruction set, and thus the same hardware; there were no longer separate vertex and fragment processors. This can be seen in NIVDIA 8800 chip (Figure 4). The processing power is essentially divided up among eight Thread Processing Clusters (TPCs) each of which contain two Streaming Multiprocessors (SMs). Each of these contains eight Scalar Processors (SPs) which each contain their own integer ALUs and FPUs and can run multiple threads. These are where most of the actual computation, for both vertices and fragments (and the more recent geometry shaders, which process triangles), takes place. Much of the chip is dedicated to getting an optimal distribution of work to the processor hierarchy, and also to making sure the processors have access to the right data.

3.2 NVIDIA's Fermi Architecture

NVIDIA's more recent chip, the Fermi architecture (Figure 5), has gone towards a more general purpose design. The TPCs have been done away with, and each of the 16 SMs is larger (Figure 6), and now contains two warp instruction

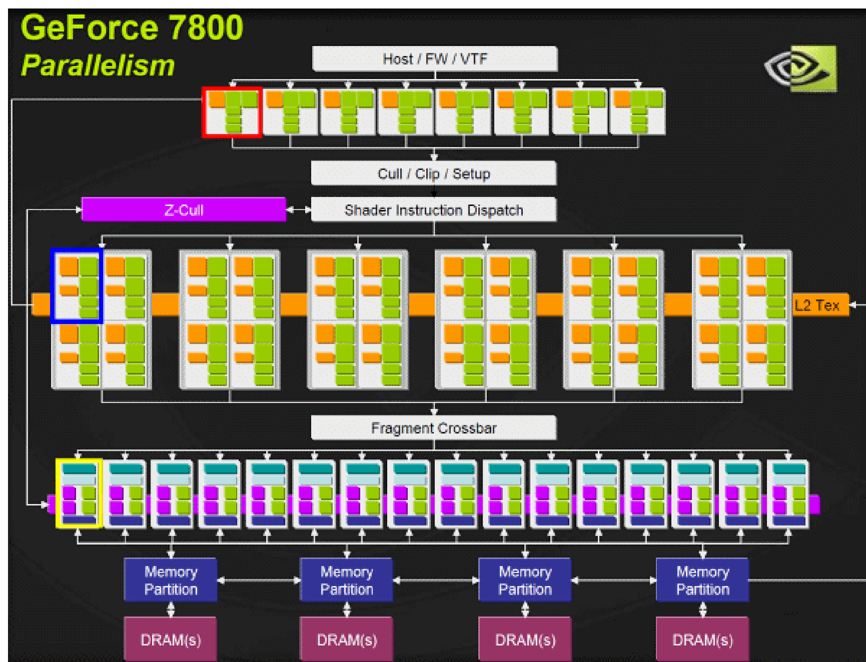

Figure 3: NVIDIA's 7800 chip, released in 2005. Something to note here is how the computing functionality is divided different kinds of processing components, as this chip design predates the unified shader model. One of the vertex processors is highlighted in red near the top, near the middle a fragment processor is highlighted in blue, and a raster operator is indicated by a yellow box near the bottom.

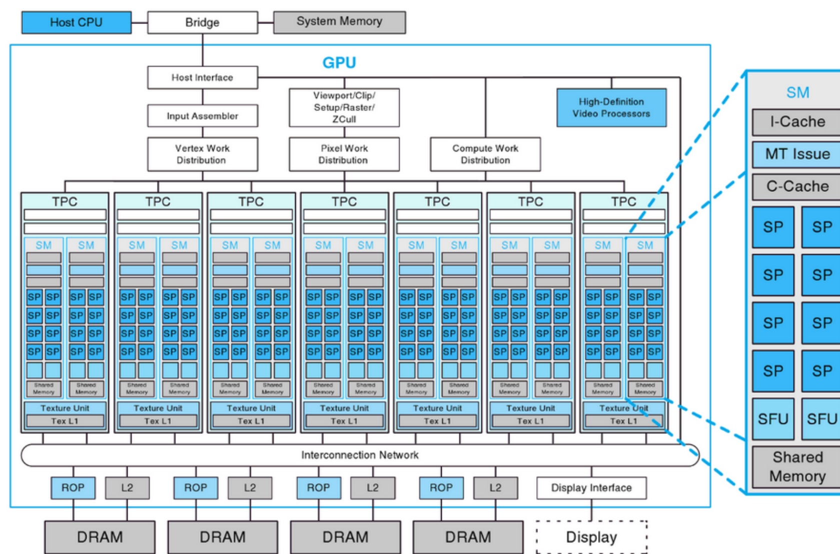

Figure 4: NVIDIA's 8800 chip, released in 2008. All differentiation between processors has been removed, and now there is only one kind of "streaming multiprocessor"; This processor contains instruction cache and dispatch logic, and 8 "scalar processors" which each contain an ALU and FPU.

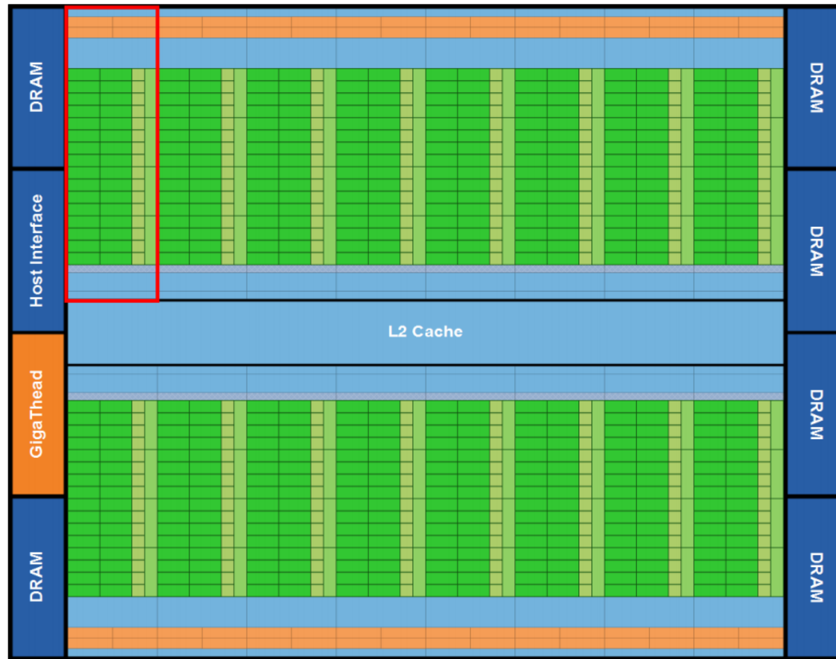

Figure 5: NVIDIA's Fermi Architecture. A close up of one of the SMs (highlighted in red) is shown in Figure 6.

schedulers, which issue instructions to two clusters of SP cores (or CUDA Cores) respectively. Each SM also has it's own register file, 16 load/store units, and 4 Special Function Units (SPUs), which perform functions such as sine and cosine. This larger, more capable SM is aimed at being able to handle more general data intensive parallel programs, such as those found in scientific computing, while still maintaining the ability to efficiently process graphical data.

This architecture has also added better and faster floating point calculations, a unified address space, error-correcting code logic for memory and many other features that make it all the more suitable for general purpose computing.

Something to note here is how large the difference in the total SPs in just one generation of NVIDIA's product: The number of SMs has remained the same, but the number of SPs has quadrupled. This trend is seen throughout the graphics architecture development community, and it is one of the primary differences between how GPUs and CPUs have been evolving: GPUs are always using smaller transistor size to dramatically increase the number of processors, aiming at ever-larger data throughput. CPUs, rather, focus on instruction Level parallelism and reducing latency.

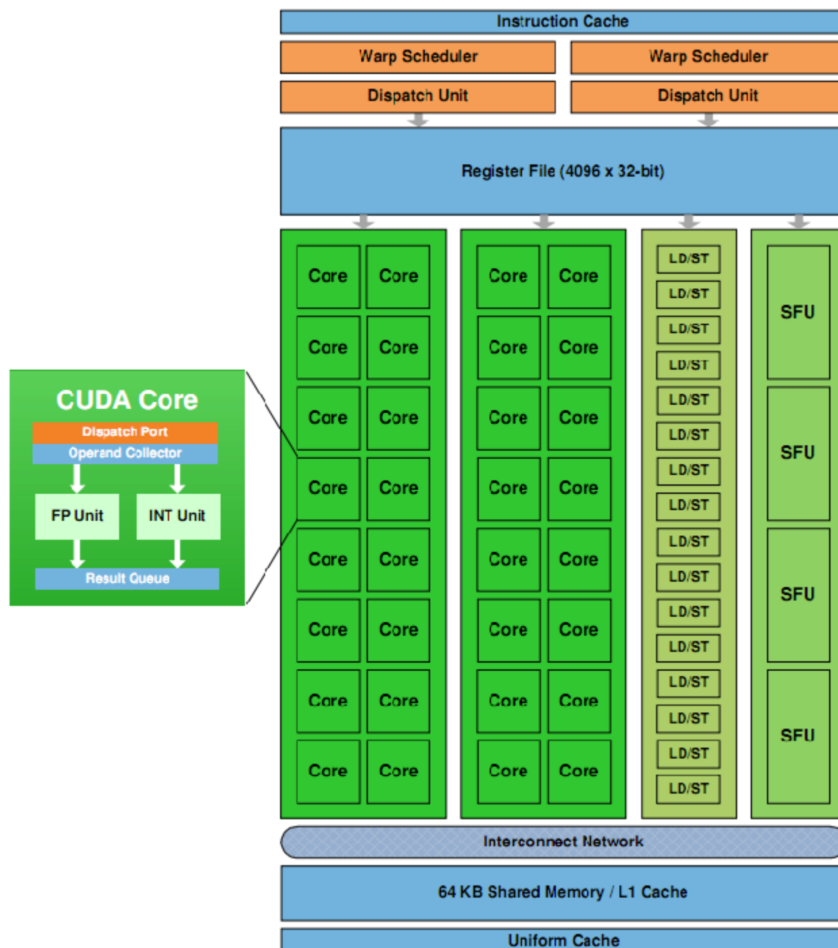

Figure 6: A close of of one of Fermi's SMs. Note that the compute power (number of cores) has quadrupled in comparison to the 7800 (Figure 3).

3.3 Intel Core i7

Modern CPUs have a very different distribution for for their chip layout; One of Intel's latest chips is the Core i7 (Figure 7), and it is apparent from looking at the die micrograph how much less emphasis is placed on raw computing power. As CPUs add more cores, each core comes with all of the logic that has been won through years of research, and at the expense of space for functional units (Figure 8). This has both benefits as well as drawbacks; CPUs are much better at performing a much larger array of tasks than GPUs, and are able to much more effectively reduce latency for individual tasks.

It is clear that CPUs and GPUs have very different priorities when it comes to distributing silicon real-estate to different functionalities. GPUs focus is on increasing raw compute power, so that more primitives (vertices, triangles, pixels) can be processed, and this large amount of throughput capability lends itself to applications like scientific computing, which is why GPU companies are courting those markets. For CPUs on the other hand, the focus has always been on reducing latency for serial tasks.

It is unlikely that either approach will become unnecessary; both devices fill important niches in the market. But perhaps they will continue to grow closer together and cooperate more and more. Indeed, the NVIDA Tegra chip (designed for mobile phones) already has a GPU and a CPU on the same piece of silicon (Figure 9).

4 Data-flow

GPU hardware is designed to compute operations on millions of pixels using thousands of vertices and large textures as inputs. The pixels must be presented at a frame rate of approximately 60 Hz to be smoothly presented to the human eye. These requirements lead to two properties of GPUs in general: very high throughput and very high latency compared to CPUs. The requirement of large throughput is self-explanatory. Large latencies are allowed because delays on the order of milliseconds are not detectable by the human eye, especially if the delays are within the computation time of a single frame, in which case they are completely hidden. The lack of stringent latency requirements has led to extremely deep pipelines in GPU hardware to take advantage of the potential increases to throughput. The entire graphics pipeline may take many thousands of cycles to complete a single pixel value, but because of the pipeline depth, may have throughput hundreds of times that of a CPU.

The main sections of a GPU pipeline are shown in Figure 10. Between the year 2000 and 2001, shaders became able to run custom pixel-generation code [13]. This led to interest from the scientific community in using the inherently parallel and powerful GPUs for general-purpose computing. The vertex, geometry, and fragment shaders (processors, cores, etc.) are the main programmable elements in the traditional graphics pipeline and are the processors used in general-purpose GPU programming.

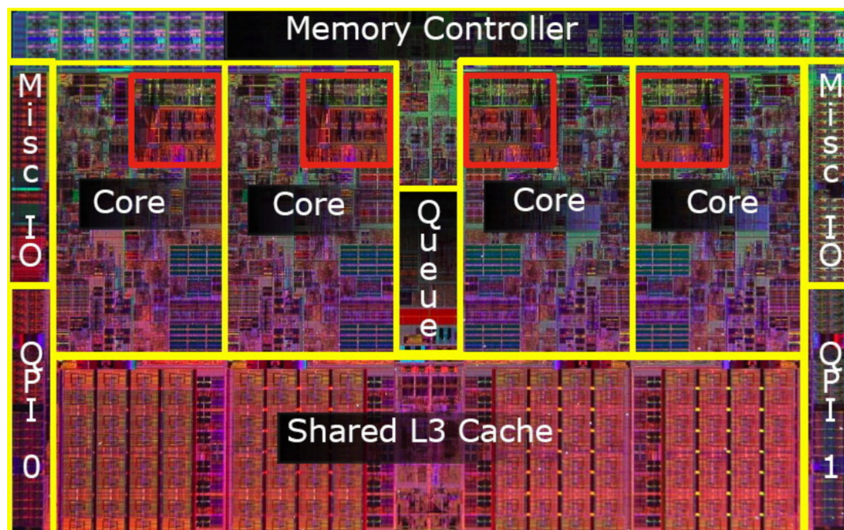

Figure 7: A die micrograph Intel's Core i7, which uses the nehalem architecture for each core (Figure 8). The Cores (on this chip there are four) are shown in yellow. The portion of each core that is dedicated to actual computation is highlighted in red.

Starting in 2006, the different programmable shaders unified their core instruction set [1] which allowed code to run in any portion of the graphics pipeline independent of the type of shader type. With the introduction of the unified shader architecture (now known from the software perspective as Shader Model 4.0), the GPU becomes essentially a many-core, streaming multiprocessor with raw mathematical performance many times that of a CPU. As of 2010, the single-precision FP calculation performance of Nvidia's Fermi (1.5 TFLOP) and AMD's 5870 (2.7 TFLOP) vastly surpasses that of CPUs. The transition to a unified shader model occurred primarily to ease load balancing [14], as present-day games vary widely in the proportion of vertex or pixel shader usage. With a unified architecture, any shader can process data, leading to better utilization of GPU resources.

5 Memory Hierarchy

There are significant differences between the memory structure of GPUs and CPUs. One reality that GPU manufacturers must deal with is the small amount of cache available to each processor, due the large number of processors and limited space on the die. Cache hit rates are frequently much lower for GPUs than CPUs (90% for the GeForce 6800 versus close to 100% for moden CPUs [12]).

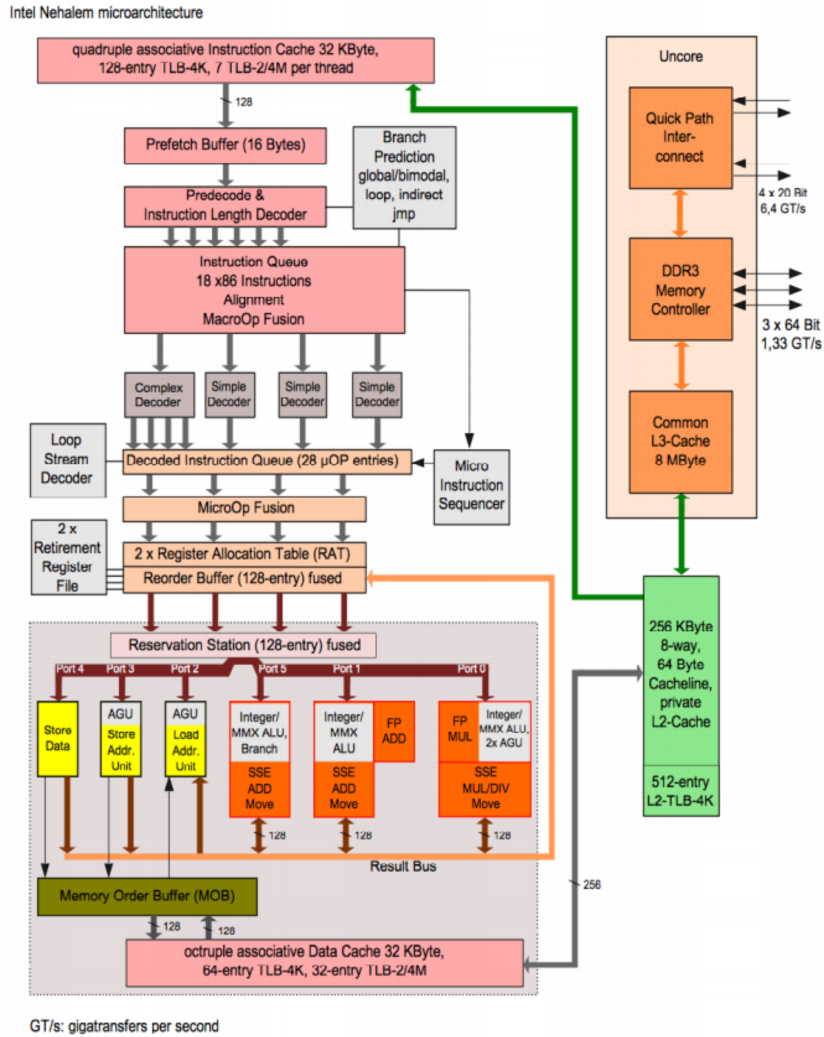

Figure 8: Intel's Nehalem architecture. Along with functional units for doing actual computation, each core on also contains reservations stations, reorder buffers, and complex branch prediction.

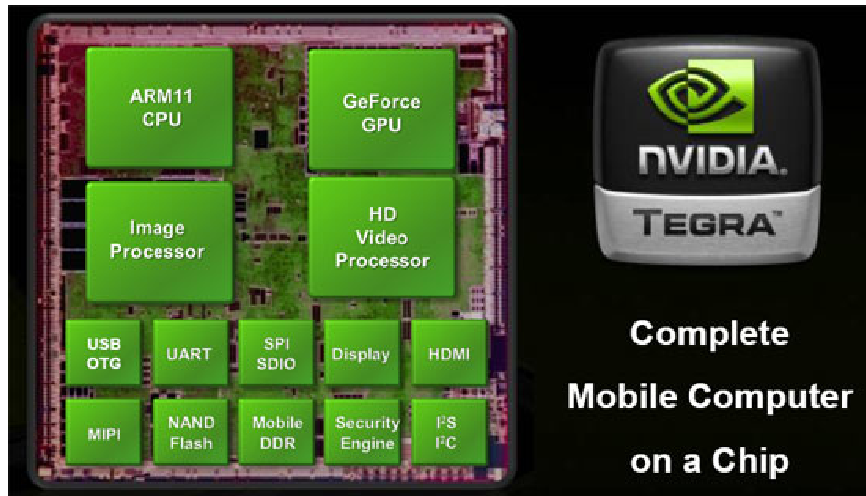

Figure 9: NVIDIA's Tegra chip (designed for mobile devices, such as smart phones) has a CPU, a GPU and image and HD video processing hardware all on the same piece of silicon.

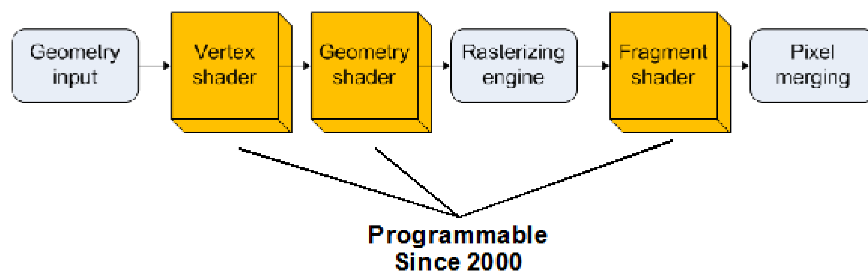

Figure 10: Simple sequential model of a graphics pipeline showing the three main types of shader processors: vertex and fragment.

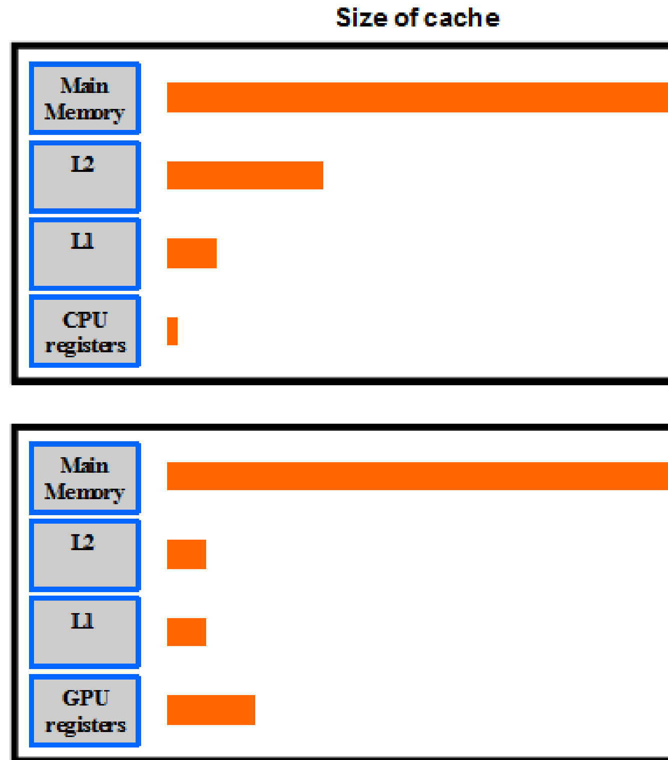

Figure 11: Typical CPU (top) versus GPU (bottom) cache hierarchy. Note the larger size of the register array than both the L1 and L2 cache for GPUs.

This may seem like a crippling situation, but the high-latency nature of the GPU pipeline can allow memory to be retrieved from a miss within acceptable delay times. Furthermore, modern thread scheduling allows a new thread to quickly take over the idle processor while it waits for the missing data to be transferred from memory [4]. The relatively high miss rate of smaller GPU caches coupled with the extremely memory-intensive nature of texture mapping results in GPU designs that aim to maximize the available memory bandwidth. There are requirements, however, for the shader programs that need to be met in order to take full advantage of the high bandwidth capability of GPU memory. A a comparison of typical cache memory distribution for CPUs and GPUs are shown in Figure 11.

The memory and pre-fetching hardware in GPUs is highly optimized toward mapping of 2D arrays [7, 6], which are the format of textures. The memory controller pre-fetches sets of data that correspond to the mapping of sequential 2D arrays (Figure 12), thus decreasing the miss rate by exploiting multidimensional spatial locality. This is in contrast to the typical pre-fetching algorithms

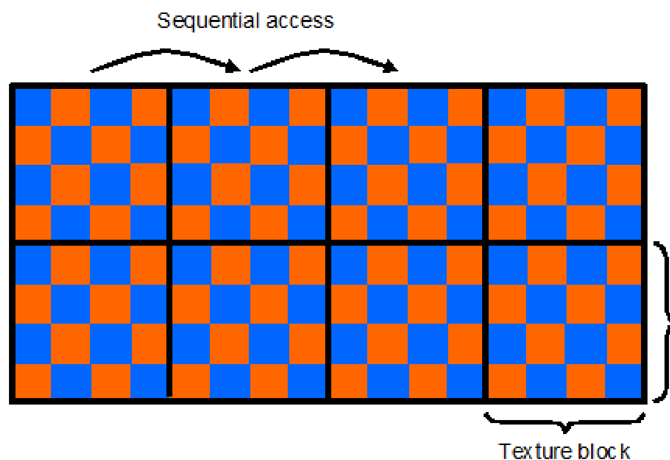

Figure 12: Sequential 2D texture memory block pre-fetching. Sequential elements in the array are pre-fetched, rather than sequential elements in memory.

of CPUs, which assume linear spatial locality. For example, to store a large array in a GPU program, it would be "wrapped" in a 2D block for quick access by the cache (Figure 13). There is a growing body of research literature dealing with the modification of various scientific computing programs toward optimizing memory access based upon the existing pre-fetching hardware.

The nature of typical graphics demands on GPUs, such as gaming, is such that a well-defined set of inputs (verticies, textures, lighting) is heavily processed through thousands of cycles and then a result (pixel colors) are finally written at the end. Because of the lack of interleaving threads which are typical of CPU programs randomly accessing memory, caches in GPUs are generally read-only. They function primarily as a way to limit the number of requests to the memory bus, the use of which is a precious and scarce resource, rather than to reduce the latency of read/write misses [6].

6 Instruction Set

In comparing GPU instruction sets to modern CPU instruction sets, it is important to consider the history of the GPU. Prior to the recent evolution of GPUs toward general-purpose computing capabilities, the GPU was a fixed-function resource, highly optimized to perform optimally those tasks that are relevant to producing 3D graphics, such as vertex operations (assembling and shading vertices into triangles) and fragment operations (texturing and coloring pixels). Dedicated hardware performed each operation, and the instruction set was specific to each hardware section depending on what tasks it did. The recent shift to general-purpose capabilities has resulted in a common set of hardware among

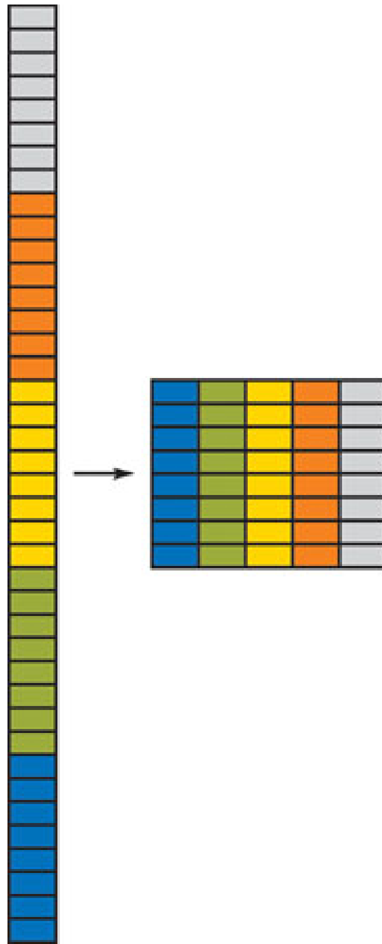

Figure 13: Conceptual wrapping of a 1D array into a 2D cache block.

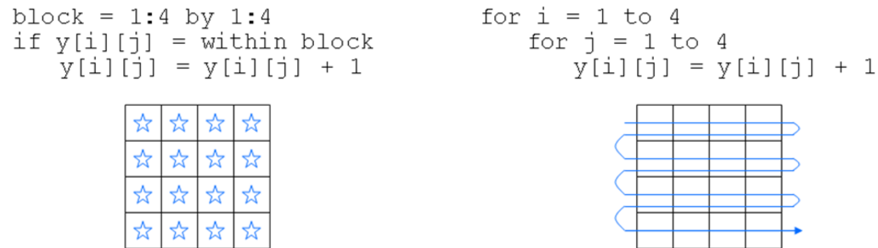

Figure 14: Comparison of an array-increment operation on GPUs (left) and CPUs (right).

all computing elements, and a unified instruction set. Some special-purpose hardware has remained, but all computing elements are capable of executing code within the unified instruction set which brings some important capabilities to the previously limited set of common instructions [17]:

- Support for floating-point calculations across all processors
- Arbitrary reading from memory
- Loops and conditional statements

With the common instruction set, as well as program space for custom vertex and fragment code, a wide variety of programs can be implemented. In recent years, double-precision FP units have become more and more common in computing elements, increasing the range of options available for GPU programs. The instruction sets have also been more and more optimized for running C-code. From the early academic compilers like BrookGPU [2] to modern C-like languages such as Nvidia's CUDA and AMD's Stream, there has been a constant improvement in software tools year after year.

GPU instructions are inherently single-instruction, multiple-data (SIMD). In producing graphics, shaders will run the same program on thousands of pieces of data to do such tasks as apply filters and calculate lighting. As an example of how this would affect a CPU-to-GPU code transition, consider incrementing all the elements in an array (Figure 14). On a CPU, the standard doubly-nested loop would traverse the array and perform an operation on each element. In a GPU, however, a single SIMD operation is able to operate on all elements in parallel. A section of memory to which the operation will be applied is mapped, and the operation simple executes if it is within the prescribed bounds. In the left example of Figure 14, the code would be run on 16 processors, each operating on a different element.

Comparison of an array-increment operation on GPUs (left) and CPUs (right).

There are other significant differences between programs compiled for CPU and programs compiled for GPUs. Branches incur a large penalty due to branch

granularity; all elements in a computational block that share register memory (i.e. Nvidia's thread processing clusters) are forced to compute both sides of a branch if different elements branch in different directions, which forces the threads to become serial until the code paths converge again. The number of elements in the block is known as the branch granularity. The branch granularity results in very large branch penalties (compared to CPUs) that cannot be overcome by software optimizations, if the programmer desires each element to compute a different branch.

A recent innovation in exploiting parallelism is single-instruction, multiple thread (SIMT) instructions. SIMT add capability to ignore the SIMD instruction width and instead specify only the behavior of a thread, which can then be dynamically scheduled with others. Modern techniques such as warp scheduling SIMT instruction help alleviate the branch divergence penalty to some extent by dynamically rearranging the threads available to compute [4], but it does not eliminate the problem altogether.

An instruction set difference which is slowly disappearing is the prevalence of "hacks" in the architecture of GPUs to increase speed at the cost of floating-point accuracy [9, 17]. In modern GPUs, IEEE 754 floating-point precision is becoming standard [13] with greater demands for true algorithm compatibility.

7 Applications

There are specific computing applications which are particularly suited to GPU implementation. To take full advantage of the unique resources of GPUs, a program should have some of the following characteristics:

- Large amounts of inherent parallelism
- Requirement of large throughput (i.e. real-time computation)
- Linear code execution with minimal branching between threads
- High density of mathematical operations (i.e. finite-element models or solving very large sets of differential equations)
- Threads with identical but data-independent code (i.e. physics simulation or molecular dynamics, which compute the same model on many objects)

Early work in GPU computing produced very fast programs for solving large sets of differential equations, such as for the Navier-Stokes fluid flow equations. Applications which have found particular utility in the computing power of GPUs include:

- Solving dense linear systems: large linear systems comprise a huge portion of scientific computing, and often result in the largest computational penalties.

- Sorting algorithms such as the bitonic merge sort: sorting algorithms are not typically thought of as parallel operations, except for bitonic sorts, in which GPUs perform excellently.
- Search algorithms such as binary search: Per unit of energy spent, GPUs are quite efficient at sorting. Applications such as search engines and databases may benefit from efficient search capabilities.
- Complex physical simulation such as protein folding: there are many unsolved problems in the medical community with regard to simulation of protein behavior. Access to very low cost per TFLOP computing power such as GPU arrays could possibly increase the pace of solutions.
- Real-time physics in games and rendering: throughput dominates the computational demand in real-time applications involving many elements such as in games.
- N-body simulations: these can be broken down into parallel few-body problems and computed separately before combining

These are applications that have all been solved by CPUs and supercomputers in the past. However, GPUs are much more affordable than large clusters of CPUs, and are already needed for common graphics demands.

8 Conclusion

The recent additions which have made general-purpose computation possible have been largely toward CPU-like elements, such as universal floating-point capability, loops, conditionals, and instruction set compatibility with C-like compilers.

GPUs and CPUs still fill different niches in the market for high performance architecture. GPUs are built for large throughput, and running fairly simple, but costly programs, but despite their general purpose aspirations, are still designed to be optimal for a very specific purpose. CPUs, on the other hand are designed to lower latency in a single task as much as possible, can handle the most complex programs with ease, but still lag behind GPUs when large amounts of simple calculations can be done in parallel.

Both will likely always be needed; it is unlikely that the need for either approach will ever disappear. But the distinctions are certainly beginning to blur. The variety of demand for computational power will likely produce more innovations that bring computation paradigms across the GPU-CPU border. Having a highly parallel, high throughput computation resource like the GPU easily available and accessible in addition to the ubiquitous CPU on modern PCs will allow the best of both worlds to be realized.

References

[1] D. Blythe. The direct3d 10 system. *ACM Trans. Graph.*, 25:724–734, 2006.

[2] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM SIGGRAPH 2004*, pages 177–186, 2004.

[3] K. Fatahalian and M. Houston. A closer look at gpus. *Commun. ACM*, 51:50–57, October 2008.

[4] W. Fung, I. Sham, G. Yuan, and T. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. *IEEE/ACM International Symposium on Microarchitecture*, 407-420, 2007.

[5] P. Glaskowsky. Nvidia's fermi: The first complete gpu computing architecture. Technical report, NVIDA, 2009.

[6] N. Govindaraju, S. Larsen, J. Gray, and D. Manocha. A memory model for scientific algorithms on graphics processors. *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.

[7] Z. Hakura and A. Gupta. The design and analysis of a cache architecture for texture mapping. *SIGARCH Comput. Archit. News*, 25:108–120, 1997.

[8] M. Harris. GPGPU: General-purpose computation on GPUs. In *Presentation at the Game Developers Conference*, 2005.

[9] K. Hillesland and A. Lastra. Gpu floating-point paranoia. *Proceedings of GP2*, 2004.

[10] M. Kanellos. Nvidia buys out 3dfx graphics chip business. *CNET News*, 2000.

[11] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28:39–55, March 2008.

[12] J. Montrym and H. Moreton. The geforce 6800. *IEEE Micro*, 16, 2006.

[13] J. Nickolls and W. Dally. The gpu computing era. *IEEE Micro*, 30:56–69, 2010.

[14] Nvidia. Geforce 8800 architecture overview tb-02787-001. Technical report, Nvidia, 2006.

[15] NVIDIA. Nvidia's next generation cuda architecture: Fermi. Technical report, NVIDIA, 2009.

[16] J. Owens. Data-parallel algorithms and data structures. In *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[17] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. Gpu computing. *Proceedings of the IEEE*, 96:879–899, 2008.

[18] D. Patterson. The top 10 innovations in the new nvidia fermi architecture and the top 3 next challenges. Technical report, U.C. Berkley, 2009.

[19] D. Patterson and J. Hennessy. *Computer Organisation and Design*. Elsevier, 2008.

[20] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A gpgpu compiler for memory optimization and parallelism management. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 86–97, New York, NY, USA, 2010. ACM.