© 2005 - 2009 JATIT. All rights reserved.

GPU PROGRAMMING PARADIGM

[1]J. ÁLVAREZ-CEDILLO, [1]I. RIVERA-ZÁRATE,

[1]J. CARLOS HERRERA LOZADA,[1]M. OLGUIN-CARBAJAL

[1]CIDETEC-IPN México

E-mail: {jaalvarez, irivera, jlozada, molguin}@ipn.mx

ABSTRACT

Using the graphics processing unit (GPU) to accelerate general-purpose computations has become an important technique in scientific research. However, the development complexity is significantly higher than for CPU-based solutions, due to the mainly graphics-oriented concepts and development tools for GPU-programming. As a consequence, general-purpose computations on the GPU are mainly discussed in the academic domain and have not yet fully reached industrial software development. This paper presents a novel contribution to general-purpose GPU programming – the analysis of the new paradigms of programing for all the programing languages based GPU .

Keywords: *GPU architecture, Computer graphics card, GPU performance, Programming paradigms.*

1. INTRODUCTION

Commodity graphics hardware has evolved tremendously over the last years – it started with basic polygon rendering via 3dfx's Voodoo Graphics in 1996, and continued with custom vertex manipulation four years later, the graphics processing unit (GPU) now has improved to a full-grown graphics-driven processing architecture with a speed-performance approx. 750 times higher than a decade before (1996: 50 b/s, 2006: 36,8 b/s).

This makes the GPU evolving much faster than the CPU, which became approx. 50 times faster in the same period (1996: 66 SPECfp2000, 2006: 3010 SPECfp2000) [1]. Figure 1.1 shows the GPU performance over the last ten years and how the gap to the CPU increases.

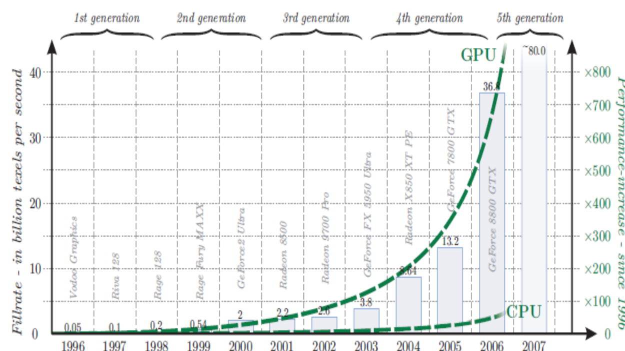


Fig. 1.1 The performance-increase of computer graphics hardware over the last decade . The green trend line shows that the GPU doubles its speed-performance every 13 months (i.e. GPU of 2006 are approx. 750 times faster than G Pus of 1996). In contrast, the performance of the CPU doubles only every 22 months [1].

2. GENERAL PURPOSE COMPUTATION USING GPU

Early graphics processing units of the 1980s and 1990s were built as 2D accelerators. Common operations included bit blitting, which is a combination of two bit maps using a raster op (for example: AND, OR). Today, GPUs are a cheap commodity solution to having a data-parallel co-processor alongside the CPU. In their evolution as processors, they became fully programmable in three stages of the graphics pipeline Figure 2.1.

In addition, they specialize in massively parallel scalar processing; modern GPUs have up to 128 independent processors (NVIDIA 2006), whereas CPUs have, at the desktop level, only reached 8 cores.

GPUs are also out pacing CPUs in terms of raw processing horsepower. At the time of this writing, Nvidia's G80 is capable of a theoretical peak performance of 340 G flops, compared to the almost 40 G flops of Intel's Core 2 Duo processor (NVIDIA 2008, NVIDIA CUDA). GPU pixel processing is increasing at a rate of 1.7 times per year, and vertex processing is increasing at a rate of 2.3 times per year,

contrasted with a 1.4 multiplier for CPU performance (Owens et al. 2005).

Recently, there has been an increasing interest in general purpose computation on graphics hardware. The ability to work independently alongside the CPU as a coprocessor is interesting but not motivating enough to learn how to apply problems to the graphics domain. However, with full programmability and the computational power of GPUs out pacing CPUs (in terms of a price/performance ratio), the GPU has moved to a place of being the primary processing unit for certain applications, with the CPU managing data and direction of the GPU.

This is motivation enough to learn how to apply certain problems to the domain of computer graphics, thus we achieve general purpose computation on graphics hardware.

3. PROGRAMING PARADIGMS OF THE GPU

As the name implies, the GPU was initially designed for accelerating graphical tasks.However, it was soon being exploited for performing non-graphical computations, for instance, the work of Lengyel uses graphics hardware to compute robot motion, the Cypher Flow project by Kedem and Ishihara exploits the graphics accelerator to decipher encrypted data [4, 5, 6].

Nevertheless, while more and more algorithms are implemented for graphics hardware, most of the work stays in the academic field and has not yet found its way into industrial software engineering.

The reason is that GPU-based application development is much more complex, mainly because the developer has to be an expert in two domains in the application's domain,and in computer graphics. This means that changing the graphics-oriented paradigms and corresponding GPU development tools may significantly reduce development complexity.

Most of the existing GPU-based development systems are founded on the graphics oriented paradigm that is illustrated in the figure 3.1 the visual paradigm. This approach has been in pronounced by the entertainment and special-effects industry, where the software developer creates the so-called rendering engine and the graphics artist uses the engine to create so-called shader programs that compute visual phenomena.

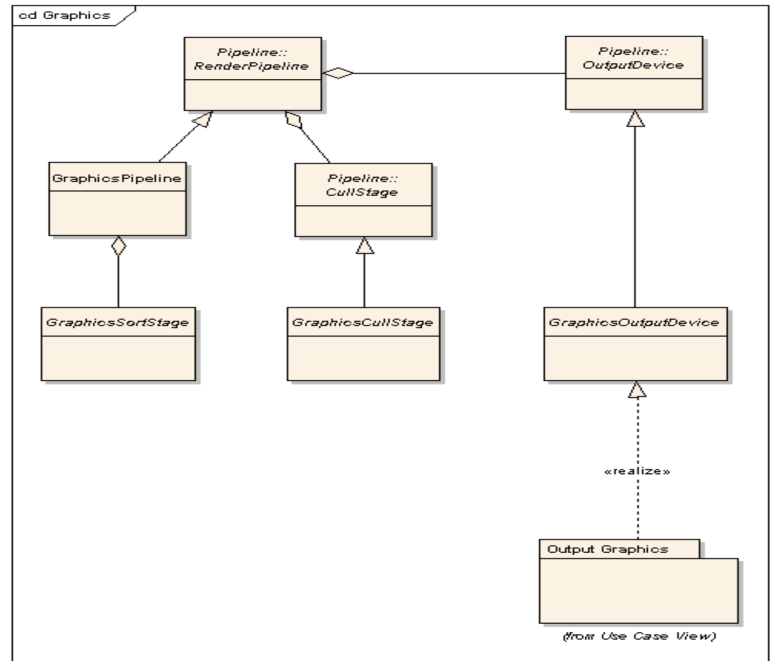


Fig. 3.1. The Graphic Pipeline.

For instance, a very popular rendering engine is the Render Man software while the software itself is developed by Pixar, the visual effects" (i.e. shader programs) are created by individual special-effects companies [12]. In this case, the visual paradigm makes sense, because the graphics artists and the software developers usually do not know each other and therefore, have separated each work that are tailored to the specific needs of each group.

Figure 3.1: The visual paradigm knows two participants the software developer and the graphics artist each having its own work

The software developer creates the engine that is used by the graphics artist to create shader programs (that run as part of the engine). Examples for GPU-based development systems that are founded on the visual paradigm are Render Man, Gelato, Pfman, Interact-SL, RTSL, Cg, and GLslang [13, 14, 15, 16, 17, 18].

However, such development systems are inappropriate to implement general-purpose algorithms on the GPU, mainly because of the following two reasons:

Due to the distinct specialized work, broad knowledge is required in both disciplines software engineering (developer) and computer graphics (artist).

The visual paradigm also forces the software developer to leave his familiar development environment and to develop in the graphics artist's programming language.

Furthermore, because CPU and GPU-based code are developed in different programming languages, additional binding code is required to glue the different functionality together.

Please note that there are GPU-based development systems that relax the aforementioned issues. For instance, Brook for GPUs, CUDA, and CTM are not graphics oriented,but they still separate between CPU- and GPU-based code [18, 20, 21]. On the other hand, Sh allows mixing the code of both processor platforms in the same programming language, but still requires knowledge in the computer graphics domain [19].

As a matter of fact, to efficiently develop general-purpose applications that are accelerated by the GPU, a significantly different approach is required the algorithmic paradigm

Figure 3.2 illustrated the paradigm: the software developer creates the complete software that contains code for the main and the graphics processor at the same time. This means that there is a single development environment, where the same programming language is used into CPU- and GPU-based code side by side and no binding code is required that connects the variables of the different processor platforms.

Fig. 3.2. The algorithmic paradigm knows only the software developer, who creates the whole software including code for CPU and GPU. The GPU-based functionality of the software has to be extracted and transformed to the graphics hardware's internal format.

In other words, the algorithmic paradigm conceptually eliminates the separation between both processor platforms. In practice, GPU-based development systems that are founded on this paradigm have to deal with the following three challenges:

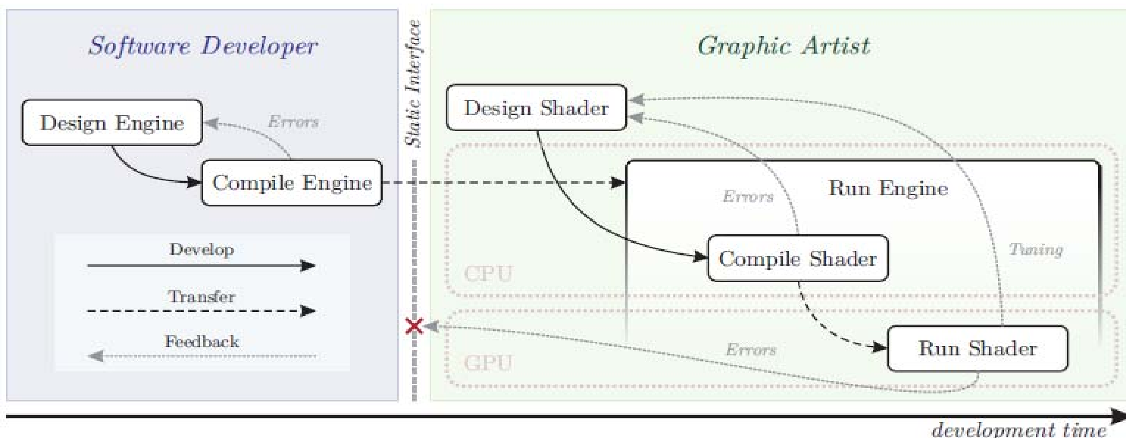
a) Compact Set of Generic Concepts

The development system has to present a consistent set of generic concepts to abstract techniques and terminology that are specific to computer graphics. For instance, vertex and fragment processing is hard to understand by the common software developer and need to be abstracted by a generic concept.

The challenging task is that the new concepts have to be easier to learn, compared to the graphics-oriented concepts of the visual paradigm. While this is hard to evaluate, the rule-of-thumb is: the less concepts, the easier to learn

b) Uniform Development Environment

Both processor platforms have to be accessible in the same development environment. In other words, CPU- and GPU-based code should be mixable in the same source code , i.e. the code is speciefed in the same general-purpose programming language. Ideally, the software developer is able to use the same familiar syntax, the same compiler collection, and the same testing environment he is used to for both processor architectures. Please note that familiar syntax" means that mathematical expressions are speciefed in the same way, no matter of the target platform i.e. that code should be interchangeable.



c) Seamless Data Interchange Between CPU and GPU

Development systems that follow the algorithmic paradigm have to eliminate the need of binding code variables have to be accessed easily on both processor architectures. In other words, computation results of the graphics hardware have to be directly accessible in CPU based code, and main processor variables have to be usable as GPU-based inputs.

Please note that the prior challenges mainly addressed the syntax and style of source code and the software development work flow, while this is focused on the seamless interchange and automatic transfer of data between the main and graphics processor.

The aforementioned challenges are accomplished by using the following two-stage approach: On a theoretical level, the existing graphics-oriented concepts for programming the GPU are abstracted to generic concepts, e.g. pixel and fragment processing is abstracted via a unificated kernel definition, and the vector processor concept is abstracted via vector fusion. Furthermore, on a practical level, graphics-hardware-based code is seamlessly integrated into the general-purpose C++ programming language, which is realized by using advanced object-oriented techniques like "ad-hoc polymorphism" and "generic programming".

To further reduce the development complexity and to improve run-time performance, a variety of optimization strategies are automatically applied to the GPU-based code.

4 PERFORMANCE

Successful use of the GPU for general purpose computation requires taking into account the significant overhead incurred in executing and managing GPU kernels. This includes queuing overhead, scheduling overhead,, and the GPU progress check period.

a)GPU Overhead

Our study of GPU overhead examines the setup and management costs for a GPU kernel. This includes kernel startup, buffer allocation, and memory transfer from the CPU to the device and in the other direction. We chose not to include the overhead involved in data transfer to on-chip memory. Since these units of memory are small, consisting of tens of KB, the overhead of data transfer in this context is small.

b)Kernel Startup and Termination Overhead

Kernel startup overhead is the time from the point when the kernel is invoked on the CPU to when it starts executing on the GPU. Kernel termination overhead is the time from when the kernel nishes execution on the GPU to the point when the CPU can continue executing after a blocking kernel-synchronization call.

We looked at the sum of kernel startup and termination costs by timing the synchronized execution of an empty CUDA kernel, repeating this operation 50 times within a loop. We executed the program twice with different kernel grid structures.

The first grid consisted of a single block with a single thread. The second kernel had a 64 by 64 grid of blocks with each block consisting of 256 threads.

The results are summarized in table 4.1. For both grid sizes we noticed that the first kernel execution in the program took significantly longer, presumably dueto some CUDA setup operation. All subsequent kernels produced very consistent results, as evident from the low standard deviation. The average kernel overhead for the simple grid was 10.0 fs. For the larger grid the average overhead was 36.6fs.

These results reveal that the constant costs associated with kernel execution are quite high. We can see, for example, that a piece of CPU code which runs for less than 10 fs will not benet from execution on the GPU, even if the kernel can give a good speedup over the CPU code.

We should also examine the difference in overhead between the first, light, kernel grid, and the second kernel grid, which consists of over a million threads. The overhead in the second case is about 3.6 times higher, Streaming Multiprocessors have a limited block and thread capacity. Only a subset of the blocks will initially be assigned to the SMs, with the remaining blocks scheduled after previous ones nish execution. This is most likely the reason for the slower execution. We should keep in mind that in practice some of this overhead could be hidden by the execution of a compute-intensive kernel. Nonetheless, our set up gives us an upper bound for kernel overhead in the case of a big grid

c) Memory Allocation and Transfer

We tested GPU memory performance, timing memory allocation and CPU-GPU data transfers. We conducted each test with buffer sizes from 1 byte to 128 MB.

Results are shown in figure 4.1. We conducted the tests on a workstation having an NVIDIA

GeForce G92 8800 GTS graphics card, an Intel Core 2 Quad processor clocked at 3.0 GHz, 4 GB of DDR2 RAM, and a motherboard using P35 chipset.

We can see from the plot that GPU memory overhead is quite high. A closer look reveals that the relationship between the size of the buffer and the time for allocation or transfer is complex, changing as size of the buffer grows.

The allocation overhead is roughly constant at 1-2 fs for buffers up to 2KB. After a sudden increase to 90 fs for 4 KB, the overhead remains roughly constant again, reaching 100 fs only at 64KB. From there the overhead increases sub linearly with buffer size until 32 MB, and in roughly linear fashion after that. For a 128 MB buffer, the overhead is 3.5 ms.

Table 4.1. Kernel startup and termination overhead. Values represent time in microseconds.

	Grid 1	Grid 2
Minimum	**9**	**36**
Maximum	**24**	**51**
Mean	**10**	**36.6**
Std. Dev.	**2.07**	**2.11**

The transfer overhead is about 15 ss for buffers less than 4 KB. For larger buffer sizes, the relationship between buffer size and transfer time is roughly linear, with a higher average rate of growth than for memory allocations. It should be noted that transfer from the GPU to the CPU is somewhat faster than in the other direction, particularly for large buffers As an example, transferring a 128 MB buffer from the CPU to the GPU takes almost 89 ms. Transfer in the other direction is 40 % faster,at 54 ms.

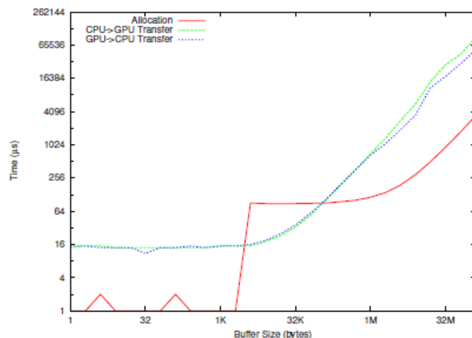


***Fig. 4.1.** Memory allocation and transfer overhead.*

Based on the results of these tests, we recommend reusing work request buffers whenever possible, and only transferring GPU data back to the CPU when necessary.

As an example, if the GPU data needs to be modified between the execution of two kernels, it might be better to perform the transformation on the GPU, perhaps using a separate kernel, even if the efficiency of the transformation is lower on the GPU.

5. CONCLUSION

The acceleration of general-purpose computations by using programmable graphics hardware has become very popular over the last decade, mainly due to the massively parallelized processor design and the enormous availability of GPUs as an inexpensive standard component of today's personal computers. However, most existing GPU-related development systems are graphics-oriented and developing general-purpose applications in such environments is a challenging task for programmers who are not familiar to computer graphics.

Its factible integrate a new programing language. This integration going to be realized in two ways: First, the graphics-oriented programming paradigms going to replace by generic concepts, using novel techniques like the unificated kernel definition and the vector fusion approach.

Second, the definition of GPU-based code going to be embedded into the C++ programming language using ad-hoc polymorphism and operator overloading. In addition, the development complexity is further reduced by providing automatic optimizations of the GPU-based code.

REFERENCES

[1] J. L. SPEC CPU2000: measuring CPU performance in the new millennium. IEEE Computer, 33(7):28-35, 2000.

[2] D. Kirk. The future: programmable GPUs & cinematic computing. Presentation at WinHEC'03, 2003. On line available at http://developer.nvidia.com/object/cg_tutorial_teaching.html.

[3] W. R. Mark. Future visualization platform. Panel Presentation at IEEE Visualization (VIS'04), 2004. On line available at

http://wwwcsl.
csres.utexas.edu/users/billmark/talks.

[4] J. Lengyel, M. Reichert, B. R. Donald, and D. P. Greenberg. Real-time robot motion planning using rasterizing computer graphics hardware. In Computer Graphics (SIGGRAPH'90 Proceedings), volume 24, pages 327-335, August 1990.

[5] G. Kedem and Y. Ishihara. Brute force attack on UNIX passwords with SIMD computer. In USENIX Security Symposium (SECURITY'99 Proceedings), pages 93-98, August 1999.

[6] K. E. Hof III, T. Culver, J. Keyser, M. Lin, and D. Manocha. Fast computation of generalized Voronoi diagrams using graphics hardware. In Computer Graphics (SIGGRAPH'99 Proceedings), pages 277-286, July 1999.

[7] P. Kipfer, M. Segal, and R.Westermann. UberFlow: A GPU-based particle engine. In ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware (EGGH'04 Proceedings), pages 115-122, 2004.

[8] E. S. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. In High Performance Networking and Computing (SC'01 Proceedings), November 2001.

[9] T. Jansen, B. von Rymon-Lipinski, N. Hanssen, and E. Keeve. Fourier Volume Rendering on the GPU using a Split-Stream-FFT. In Vision, Modeling, and Visualization (VMV'04 Proceedings), November 2004.

[10] OpenGL Architecture Review Board. ARB_vertex_buffer_object. OpenGL Extension,
2003. On line available at http://www.opengl.org/registry/specs/ARB/vertex_buffer_object.txt.

[11] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In Symposium on Volume Visualization (VVS'94 Proceedings), pages 91-98, October 1994.

[12] OpenGL Architecture Review Board, D. Shreiner, M. Woo, J. Neider, and T. Davis. OpenGL Programming Guide: The Oficial Guide to Learning OpenGL. Addison-Wesley Professional, second edition, 2005.

[13] P. Hanrahan and J. Lawson. A language for shading and lighting calculations. In Computer Graphics (SIGGRAPH'90 Proceedings), volume 24, pages 289-298, August 1990.

[14] M. Olano and A. Lastra. A shading language on graphics hardware: The PixelFlow shading system. In Computer Graphics (SIGGRAPH'98 Proceedings), volume 32, pages 159{168, July 1998.

[15] M. S. Peercy, M. Olano, J. Airey, and P. J. Ungar. Interactive multi-pass programmable shading. In Computer Graphics (SIGGRAPH'00 Proceedings), volume 34, pages 425-432, July 2000.

[16] K. Proudfoot, W. R. Mark, S. Tzvetkov, and P. Hanrahan. A real-time procedural shading system for programmable graphics hardware. In Computer Graphics (SIG-GRAPH'01 Proceedings), volume 35, pages 159-170, August 2001.

[17] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. In Computer Graphics (SIGGRAPH'03 Proceedings), volume 37, pages 896-907, July 2003.

[18] R. J. Rost. OpenGL Shading Language. Addison-Wesley Professional, second edition.