…When They

Attack In Packs

“Polygons feel no pain” Volume 3

Ingemar Ragnemalm

Course book in GPU computing

Foreword

In the past, I wrote “Polygons feel no pain” and “so how can you make them scream?”. For several years I have considered making a small course book for my part of TDDD56 “Multicore and GPU programming” so you have something that firmly summarizes my part of the course, complete and without too much fluff.

There are other books on the topic, of course, and some are pretty neat and non-bulky. However, they are usually focused entirely on CUDA, possibly with some OpenCL material but hardly more. This book, although brief, has the ambition to give an introduction to the field with a wider scope, including several other platforms of interest.

I may not be a known expert in the field, but having worked with this concept since 2005, giving my first GPU computing course before CUDA even existed, I feel that I should be capable of providing a decent book.

Parts of the book are based on the GPU computing chapter of Volume 2. Much of the contents is also based on NVidia’s CUDA programming guide. [3] Another important source is Mark Harris’ blog entries, e.g. [15]

You should immediately see that the name fits with the names of the earlier books. It also refers to a rarely heard joke that I remember, I think it was from a crowded university party:

“Ingen panik! Anfall i flock! Alla på en gång!”

which translates to

“No panic! Attack in packs! Everybody at the same time!”

For some reason I never forgot that and it fits right in.

Cover image by myself, “a pack of ‘cudas”.

Related web page: http://www.computer-graphics.se

ISBN 978-91-7773-719-3

First edition 2018.

1. Introduction

This book was written as course material for the latter part of the course in “TDDD56 Multcore and GPU programming”.

1.1 Who should read this book?

Since this book is written for part of the course TDDD56 at the University of Linköping, attendants of that course are the target audience. It is also likely to be used in PhD courses on the GPU Computing concept. Anyone else interested in the subject is also in the target audience!

The purpose of the book is to provide a broad overview of the GPU Computing programming subject, far broader than most other books.

1.2 What should you expect to learn from this book, and its course?

This book spends most of its pages on CUDA, but it is not a CUDA book. Rather, it has an unusually broad scope, covering most GPU computing technologies of interest.

- 1. GPU history and architecture.
- 2. CUDA
- 3. OpenCL
- 4. Compute Shaders
- 5. Fragment shaders
- 6. Specific problem areas.

We start with CUDA for the simple reason that it is the API that is by far easiest to get started with. Once you know it, however, you will find it pretty easy to move to any of the others.

Most code examples will be in CUDA, but the book will aim towards covering the other platforms as well, with emphasis on OpenCL and OpenGL Compute Shaders. We can not

put all code examples for all platforms in the book, but there will be a (slowly) growing amount of accompanying code on my website, computer-graphics.se.

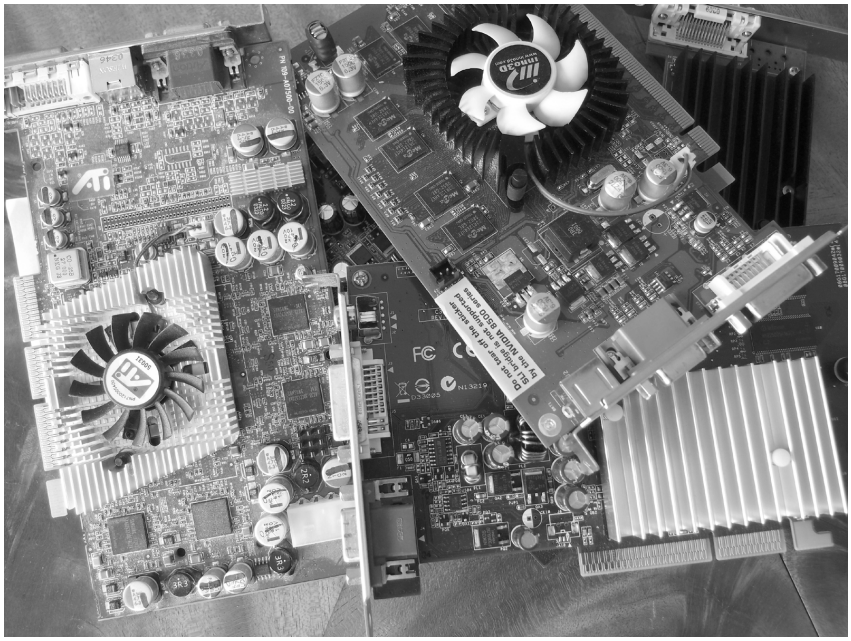
Thus, let us emphasize this: This is not a CUDA book, this book is not only about CUDA. It is a general GPU Computing book, although mostly using CUDA for examples for space and readability reasons, but I strongly advocate diversity and being knowledgeable of alternatives. And there are some strong ones to consider.

1.3 Acknowledgments

The material for this book comes from a variety of sources, books, papers, web pages.

The first contributor was Erik Pettersson [2]. In 2005, he made an early GPU Computing project as master thesis, with me as examiner. Later, Johan Hedborg, Fredrik Viksten and Jens Ogniewski have contributed, first to my PhD courses and later to the TDDD56 course. Finally, the collaboration with Christoph Kessler and his group over the course has been most important and fruitful. The recent (2017) move towards more image processing in the course was particularly nice for me, since I have a background in image processing, and has influenced chapter 18.

Finally, all previous students on the courses have contributed in various ways. Your interest for certain problems have been very important for the course.



At the top-right in this pile is an NVidia 8500 GT, my first CUDA capable GPU, and on top-left (that is left in the picture) is an ATI Radeon 9800 PRO, the card on which I made my first GPGPU experiments.

2. Table of contents

3. How did we get here?

Let us start with having a look at how we got here, what developments that led us to the hardware that we have today. This is more than just a peek back in history, it may also give some insights in what to expect from the GPU hardware.

Since personal computers arrived, the development of CPUs can be summarized as follows:

80's: CPU and the memory bus had the same speed, the same clock frequency. The concept "Zero wait states" was an honor word, the CPU never had to wait for the bus more than one clock cycle.

1993: About this time, the 1-1 mapping between CPU and the system was scrapped, instead the term "clock doubling" described the new way; The CPUs was faster than the rest of the system, by 2 or 3 times. This gave us a rapid raise of CPU frequency, making it possible to do more computations between every memory access.

Late 90's to present: We saw multi-CPU systems in the late 90's, but they were initially a limited success. Since then the operating systems have been adapted to fully support multiple CPUs, and we got multi-core CPUs with two cores on a chip. Today, 8 cores is getting increasingly common while 16 cores is available but expensive. Even embedded systems like phones are multi-core, and systems with less than 2 cores are getting rare.

CPUs are still improving, but going for higher frequency is not as obvious as before.

During this time, graphics hardware has undergone an even faster evolution.

80's: Graphics hardware in these days mainly referred to hardware that read pixels from VRAM and put them on a screen. The fanciest hardware acceleration was probably hardware sprites. Graphics programming was very focused on writing pixels to VRAM with low-level code, optimized assembly code.

1993: With the clock doubled/tripled CPUs we got enough power to produce textured 3D games like Wolfenstein 3D and Doom. In a glorious 320x240 resolution, these games were sensations! However, rendering was still a job for the CPU.

But were there not any GPUs? Yes, there was, but they were professional 3D boards, insanely expensive from a home computing point of view. This all changed 1996:

1996: With the 3dfx Voodoo1 board, we suddenly got a GPU that anyone could buy! It was priced pretty much like today's mid-range boards. Of course it did extremely little from a modern point of view, and had a strange solution for switching between 2D and 3D graphics, with a special cable over to the 2D board, but it took gaming from a blocky 320x240 resolution to a slick-looking 640x480.

2001: This year gave us a revolution that was bigger than most people understood: Programmable shaders. For the first time, we could put our own program code into the GPU!

2006: The G80 gave us the "unified architecture", which was much more suited for GPU computing than the older architectures, and with that followed NVidia's CUDA.

2009: The non-NVidia part of the world struck back with OpenCL.

2010: The Fermi architecture was the first GPU architecture that was clearly aimed at GPU computing. It was no success, it made NVidia fall behind in gaming performance, but it was still a milestone, showing what was to come.

During this process, the number crunching performance of GPUs has increased extremely fast. NVidia tends to show off with graphics like Figure 1.

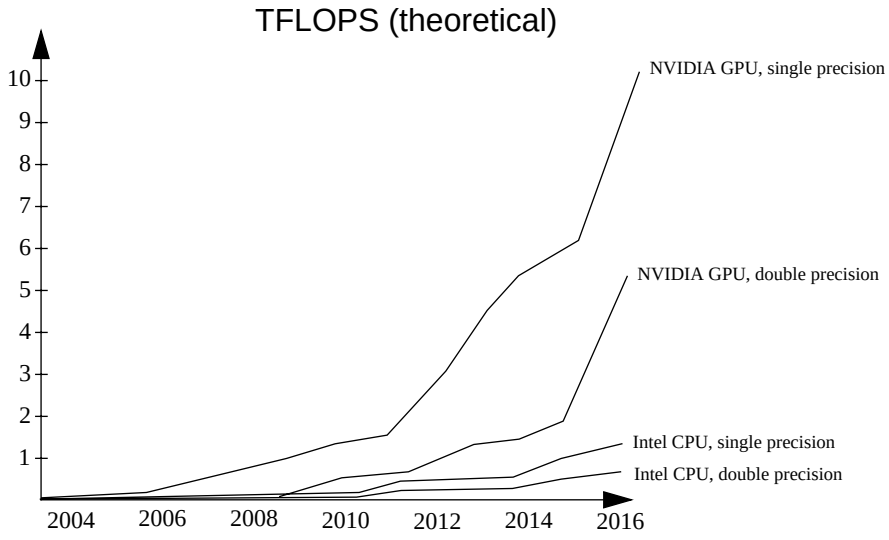


FIGURE 1. Typical "GFLOPS race" graph

These graphs look impressive and we get the impression that GPUs are improving at a rate that is way over what CPUs can perform, accelerating away leaving CPUs standing still at the starting line. However, the picture is somewhat misleading. Note that the graphs are always with *linear* scales. If you would make them with logarithmic scale for perfor-

mance, it would look much more like what it is; GPUs and CPUs are both improving at similar rates. GPUs are faster, but the proportions are not changing so much.

But is this a fair comparison? Let us compare apples with apples:

GFLOPS for both!

If we take this in numbers, we get a table like this:

```
           GPU        CPU
1995:      0.001      0.09
2005:      40         5.6
2011:      2488       91
2015:      7000       176
2016:      16380      400-700*
2017:      110000**   4000***
* Theoretical, 16 cores
** Claimed by NVidia on Titan V [5]
*** Theoretical peak performance [6]
(Other data from various sources, not documented.)
```

The 2017 part of the table reflects late 2017/early 2018.

CPUs can not compete in peak performance. Even after the biggest steps of progress, they are far behind. On the other hand, remember that this is for the most optimal problems. The advantage of GPUs drop significantly on problems that are less suited for the architecture.

Let us also consider economy. How much does a GFLOPS cost?

```
1961:      8.3 trillion
1984:      42 million
1997:      42000 (CPU cluster)
2000:      836-1300
2007:      52
2012:      0.73 (AMD 7970)
2013:      0.22 (PS4)
2015:      0.08 (Radeon R9 295)
(Source: Wikipedia)
```

That is a price/performance improvement of 104 trillion times since 1961, and only since 2000 computing has become 16000 times cheaper! And that is in an era where CPU clock frequency has stalled, and thus many people may believe that performance doesn't improve! If so, they are just 16000 times wrong (and that is not even counting the improvement since 2015).

3.1 Related efforts

So, NVidia and AMD have been the two big ones in graphics and gaming, and had considerable success in general purpose computing. We also have Intel making GPUs integrated in CPUs. They, however, tend to be of little interest for high performance applications.

The GPUs are, of course, mostly compared to the main cores of CPUs. It should be noted that they are less often compared to the processing power of CPUs while using the vector processing units in the CPUs. But there have also been other parallel architectures. At this time, many of them are counted out. IMB had the Cell processor, which even made it into Sony Playstation 3 as well as Namco Bandai arcade boards, but in 2009, its next generation was cancelled, signalling its decline. Intel made the Larabee, which was cancelled in 2010.

However, a successor of Larabee, the *Xeon Phi*, is actively developed, competing with NVidia and AMD for the growing GPU computing market.

The following two tables come from an investigation on the performance on the Phi compared to a CPU, using vector processing extensions, and a high performance GPU. Alas, I have lost the source and am unable to find it. If anyone involved in it reads this, please enlighten me.

	Xeon E5-2670	**Xeon Phi 5110P**	**Tesla K20X**
Cores	8	60	14 SMX
Logical cores	16 (HT)	240 (HT)	2688 CUDA cores
Frequency	2.60 GHz	1.053 GHz	735 MHz
GFLOPS (double)	333	1010	1317
SIMD width	256 bits	512 bits	N/A
Memory	≈16-128 GB	8GB	6GB
Memory BW	51.2 GB/s	320 GB/s	250 GB/s
Threading	software	software	hardware

So, how does it complete? The same investigation (I *think* it was) also gave us benchmarks, with the following table:

Paths	**Sequential**	**Sandy-Bridge CPU***	**Xeon Phi***	**Tesla GPU**
128K	13.062 s	694 ms	603 ms	146 ms
256K	26.106 s	1.399 s	795 ms	280 ms
512K	52.223 s	2.771 s	1.200 s	543 ms

* using SIMD vector intrinsics

My conclusion of this particular investigation is that the GPU still wins, even by a considerable margin, but the Phi, and even a CPU using vector extensions, are still fighting to give it a run for the money.

Note how much faster the CPU was by using the often ignored vector intrinsics. A 20 times speedup, for something that we always have available! But all these except the poorly performing sequential CPU solution require you to code in parallel! So whatever you do, if you want to compete in performance, you must learn parallel programming, you must *attack in packs*!

And that is why you are here, right?

3.2 Why did GPUs get so much performance?

So, the GPU is pretty fast. Let me now argue for why the GPU got all that power in the first place.

There have been many earlier attempts to construct parallel computers. However, generally speaking, they have all failed to be mass market product due to lack of a big problem suited for parallel implementation with wide enough user base to get the volumes up.

This problem was provided by the gaming industry, by the demand for good graphics. This makes it an early problem with large amounts of data, with its complex geometry and millions of output pixels. It could, with great benefits, be accelerated. The graphics pipeline is designed with excellent opportunities for parallelism!

So, we got a volume product! The 3D graphics boards quickly became a central component in the game industry. Everybody wants one, so every producer of game equipment needs to put one in.

An interesting bonus was that the hardware was designed to hide memory latency by parallelism. This is a smart trick that suits the thread model in the GPU shaders well.

So, graphics performance went up, but it didn't stop there. If a new GPU could make new impressive features, it would sell both games and GPUs. Thus, many important advancements started as game features.

So, it all started with that a GPU must process many pixels fast! This was the #1 task, so early GPUs could draw textured, shaded triangles much faster than the CPU.

The next generation could do matrix multiplication and divisions fast, in order to transform vertices and normalize vectors, which had then become a bottleneck on the CPUs.

The next step was programmability, programmable shaders. This was added to make Phong shading and bump mapping, new visual effects that were hard to do, or could only be done in inflexible ways.

Finally, floating-point support was added! This, too, was for visual effects, namely for light effects, using high dynamic range.

So a GPU should

- process vertices, many in parallel, applying the same transformations on each
- process pixels (fragments) in parallel, applying the same color/light/texture calculations on each

Both these tasks are suitable for parallel implementation. It is better than that, the problem is easily split into parts can be processed by one single program executed for multiple data. This makes it a SIMD friendly problem, single instruction, multiple data!

For such computations, we need less control per computation. The hardware will control many calculations instead of one.

This also gave us a different kind of threads. The whole process could have been expressed as a vector processor, explicitly grabbing chunks of data and feeding them to a vector processor. However, it was instead expressed as separate threads, processed in parallel running the same program.

This gave us the SIMT model, *single instruction, multiple threads*, which is SIMD hidden under an thread-like abstraction, but also with hardware support for this, providing each thread its identity, giving us an impression of independent threads. This model was good for graphics operations: Shader threads calculate one pixel or one vertex. CUDA/OpenCL threads may calculate anything, but typically one part of the output, and this can usually be made independent of each other.

Thus, these vital improvements all are based on needs of the game programmers, and thereby the needs of the gamers. They paid for our parallel computing platforms!

3.3 The arrival of GPU computing: General Purpose computation on Graphics Processing Units

The concept of GPU computing was first called GPGPU, *General Purpose computation on Graphics Processing Units*, coined by Mark Harris in 2002. The idea is to perform demanding calculations on the GPU instead of the CPU. At first, this appeared to be a wild idea, or at least a marginal possibility, but since then it has grown into a very important factor in modern computing. Results were highly varied in the early years, but the GPU advantage has grown bigger and bigger.

The concept has since then been renamed GPU computing, and more general platforms have appeared, making it easier to program and also enabling better optimizations.

3.3.1 Key components of the GPGPU trend

What made this possible was of course the massive parallelism of GPUs, which comes directly from the need to process large amounts of vertices and pixels,

The next key component was programmability, the introduction of shader programs, which made the GPUs much more flexible, reprogrammable for any problem.

The third vital component was the arrival of floating-point buffers. Without them, GPUs could only store and output integer information. Thereby it was vital for general purpose computing.

Initially, the support had poor precision. We could have as little as 16- or 24-bit floating point numbers, but with 32-bit floating-point we at least had decent precision, although not really impressive. High precision computations were not possible, but granted the

promising results, 64-bit floating-point support eventually arrived and has been steadily growing since then.

3.3.2 GPGPU/GPU computing approaches

There are several technologies for GPU computing, so let me give a brief overview. Here is a list of the most important alternatives.

- Fixed pipeline graphics
- Shader programs
- CUDA
- OpenCL
- Compute shaders
- Direct Compute
- Vulkan

The list is not exhaustive, there are many packages on top of these, and new solutions are being developed, and the current ones are revised.

3.3.3 Fixed pipeline GPGPU

Even before programmable shaders, there were several results based on the old, fixed pipeline technology. Some problems could successfully be reformulated to something that could be, at least partially, computed by standard graphics operations.

Early results include Voronoi diagrams by Hoff et. al. 1999 [7], matrix multiplication by Larsen and McAllister [8] and face tracking by Ahlberg in 1999/2002 [13].

Thus, GPUs were usable for computing even back then, but the scope of algorithms was highly limited. This kind of algorithms is not of any practical interest today.

3.3.4 Fragment shader based GPGPU

When programmable shaders arrived, the scope widened considerably. In 2005, I took part of a GPGPU project for the first time [2], and we could see an overall speedup of image processing operations of about 8 times, an amazing improvement in a business where you are happy when you can squeeze out a 10% speedup with a lot of work.

Programming is made in shader languages such as GLSL, Cg or HLSL. Initially, an assembly language was used but it was quickly phased out.

This solution has two major drawbacks. First, it requires you to re-map your data to textures, to image data, typically with four channels per pixel. This gives us a data organization that can be rather clumsy to work with. Second, the visibility of hardware features, most notably shared memory (see chapter 7), is bad compared to the following platforms.

Thus, fragment shader based GPU computing is likely to be outperformed in many applications by CUDA and other later platforms.

Still, this approach should not be counted out. It is by far the most portable one. It runs on old and new GPUs. It is easy to make your algorithm run as efficiently on the latest GPUs as well as 10 year old ones. All GPUs on the market can use shaders, so there is no need for extra software, and you can run on all brands, NVidia, AMD and Intel. All you need is the standard software/drivers.

3.3.5 CUDA

A popular platform is CUDA from NVidia. It only works on NVidia hardware, which limits it considerably, but on the other hand, it is probably the most actively developed GPU computing platform.

It requires extra software installations, and there is a big risk that a software using CUDA has steep hardware demands due to its rapid changes. However, the active development also means that new features come here early, which often give CUDA the edge.

We often see excellent results with CUDA. For problems of highly parallel nature, 100x speedups are common, even before optimizing! Even low-end GPUs give significant boosts.

3.3.6 OpenCL

OpenCL is often considered the main alternative to CUDA. It works on various hardware, not only GPUs. It is developed by Khronos Group, and initially had significant support by Apple.

It is noticeably harder to get started, partially due to the wider scope of hardware, but also since the whole model is closer to OpenGL, to the extent that there are considerable similarities in how you program shaders.

3.3.7 OpenGL Compute shaders

An alternative that is given much less attention is OpenGL compute shaders. This is a GPU computing solution built into OpenGL. That makes it similar to OpenCL, but being part of OpenGL makes it easier to make programs with OpenGL visualization. (See chapter 15.2.) This also gives it good portability, because just like fragment shaders, it exists on all installations with a recent enough OpenGL, and all you need is the GPU drivers.

3.3.8 Direct Compute

DirectX also has compute shaders, a part of DirectX/Direct3D called Direct Compute. It predates the OpenGL Compute Shaders significantly. Its biggest weakness is of course that it is limited to Microsoft systems only.

3.3.9 Vulkan

Vulkan has sometimes been called the “new OpenGL”. It is a redesign that focuses on providing good multi-thread support to graphics programming, which has become a weakness with OpenGL. It arrived officially in 2016, but now, in 2018, it is still in the process of propagating. This makes it a “Bleeding edge” technology that is not the easiest to get started with.

This could be the future main generic GPU platform for both graphics and computing. However, lack of interest from major players like Apple and Microsoft may pose problems.

3.4 Applications

I claim that this is so important, but is it? Is it being used? Let me summarize some of the strongest application areas.

3.4.1 Image processing, image analysis and video coding

From the very first steps of GPU computing, it was clear that almost any kind of image processing fits the concept well. Among the first GPU computing results were image processing tasks, and even today, it remains one of the strongest fields. Any decent video coder must use the GPU today or it will be unreasonably slow compared to the competition.

3.4.2 Crypto currency mining

In the early days of bitcoins, they were often mined using GPUs. Today, that task has been taken over by ASICs, but other crypto currencies have appeared, less suited for ASIC solutions, and those currencies are mined with GPUs.

3.4.3 Deep learning

Deep learning basically means learning systems based on very large neural networks. Learning with neural networks has been around for a long time, and was a hot topic in the 90’s, but until GPU computing arrived, it was unfeasible to handle large networks in real time.

But this is a good problem for GPUs! Simulating and updating a neural network is a highly parallel problem. And it has produced remarkable results and is now a hot trend in computer vision as well as other fields. And it was the GPUs that opened the door!

4. GPU architecture

So far, I have discussed why we got here and what the performance is. Now, let us look closer to how that performance is made possible.

4.1 SIMD and SIMT

How is this possible? In the CUDA design guide, they talk about area use, that the GPU has more space assigned to computations, while CPUs waste much space on cache. This picture is no longer valid since GPUs also use a lot of space on caches these days. Instead, I would claim that the difference is that the GPU is a SIMD machine, single instruction, multiple data, while the CPU is a MIMD. These terms come from Flynn's taxonomy (Figure 2), where we also find SISD, that is old single-core systems, and the must more exotic MISD, multiple instruction, single data, where multiple processors do the same work for redundancy, for safety.

SISD Single instruction, single data Old single-core systems	MISD Multiple instruction, single data Multiple for redundance
SIMD Single instruction, multiple data GPUs, vector processors	MIMD Multiple instruction, multiple data Multi-core CPUs

FIGURE 2. Flynn's taxonomy

SIMD, single instruction, multiple data, has a couple of advantages. It simplifies instruction handling, in that several cores get the same instruction. That means that the whole system for handling instructions is shared between many computations. In this sense, there is better area use.

This is, obviously, excellent for operations where one operation must be made on many data elements. So, is that so common? It is more common than you may realize, and many algorithms can be rewritten to be more SIMD-friendly.

But what about algorithms that are not SIMD-friendly, where there are differences? That can be managed by boolean operators, boolean variables used as masks. If you need two different variants, two branches, you compute both, and then trash the one that one particular line of computations does not need.

Moreover, SIMD computations are also easier to synchronize. You know exactly when the other computations (or at least part of them) are computed so some synchronizations can be skipped.

A rule of thumb here is to store data in arrays. Linked lists, pointers, tree structures, they can be hard to process in parallel, while arrays are easily passed to multiple processors.

This kind of processing situation is called (or at least closely related to) *Data Oriented Programming* (DOP). [14] While OOP tries to optimize programming for the programmer, DOP optimizes for performance, for the machine and the end user. Data structures are selected to fit the computations, instead of the programmer!

Optimizing for the end user instead for the programmer sounds like a good idea! This view is popular in the game industry, but seems virtually unknown otherwise.

4.2 SIMT, Single Instruction, Multiple Thread

NVidia uses the concept SIMT, Single Instruction, Multiple Thread, for their computing model. This is a variant of SIMD. I would argue that SIMT is a reformulation of SIMD, hiding the SIMD processing from the programmer, who sees the parallelism as separate threads, and gets the view of the computing as to be made in independent threads. This, however, is not the case. The threads are computed a number at a time, a *warp*.

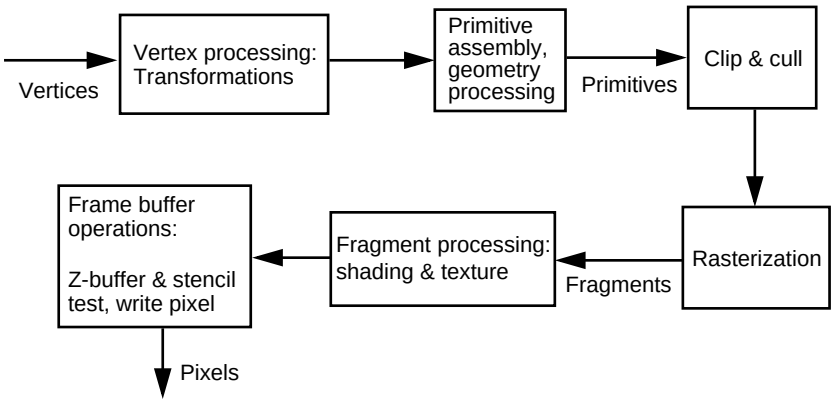


FIGURE 3. The OpenGL pipeline (simplified)

Parallelism expressed as threads is still a great improvement over handing arrays in chunks. This gives us a programming model that demands that the hardware can handle threads very fast, which is also the case for GPUs. As mentioned in chapter 3.2, this model fits a graphics processor very well.

4.3 The unified architecture

The current GPUs have a unified architecture. This is unlike early GPUs, which had a structure that closely followed the graphics pipeline, like the OpenGL pipeline, shown in Figure 3. The hardware basically had separate hardware for each step, including a number of computing cores in the vertex and fragment stages. For NVidia, this was the case up to the G70 architecture, which was roughly structured as shown in Figure 4. Note that it very closely follows Figure 3.

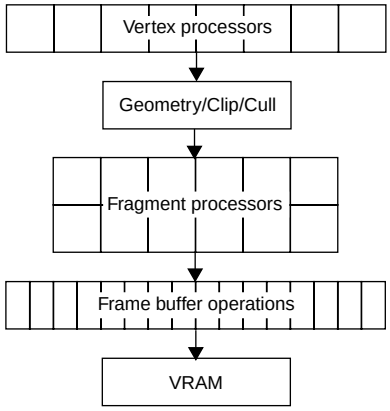


FIGURE 4. Schematic overview of the G70 GPU

In 2006, the G80 changed this, totally, All the cores in the vertex and fragment stages were collected into a pool of computing cores, all capable of performing the tasks for both stages, *unified shaders*. The data flow now passes twice through the pool of cores, as shown in Figure 5.

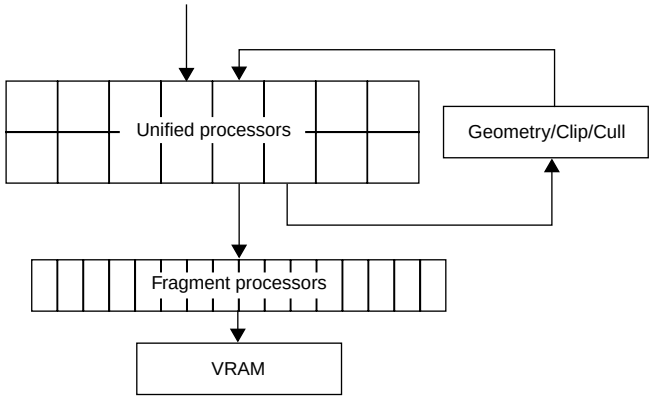


FIGURE 5. Schematic overview of the G80 GPU

This was a great success. The big advantage was that both the vertex and fragment stage could now access the whole pool, balancing the needs of the two, limiting the problem with computing bottlenecks.

For graphics, this optimized extreme situations where the balance between the difference stages is big. One such case is when there are very detailed models, with a lot of vertex computations, but little fragment processing, simple or no lighting effects etc. This is typical for 3D design situations, like CAD, where you need to see exactly how the model looks but there is no need for visual effects like Phong shading.

The other extreme is when you have less detailed models, but much computations at the pixel level, that is in the fragment shaders. This could be when there is a lot of visual effects, multiple light sources, bump mapping, ray marching per fragment, or even GPU computing in the fragment stage. The balancing problem is illustrated in Figure 6.

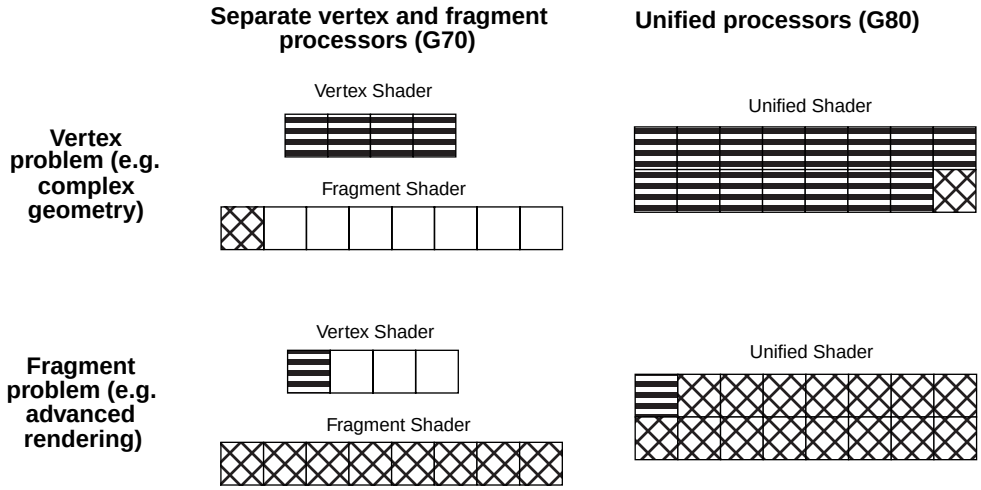


FIGURE 6. How G80 improved performance with load balancing

This new design was the first architecture that was truly suitable for GPU computing, with most of the chip dedicated to programmable computing.

4.4 SMs, SPs and shared memory

So let us have a closer look at the inside of the G80. What we find there is the design that all following GPUs are based on. See Figure 7.

We see how the processors of the G80 are grouped into eight TPCs, texture processing clusters. Each such cluster has two SMs, *streaming multiprocessors*, and hardware for texturing.

The SM concept is what we should care the most about. Inside each SM we find eight SPs, *stream processors*, which are the processing cores. However, they are not the processor cores we are used to. They are tightly coupled to a SIMD array, a vector processor, so they are essentially lanes in a single processor.

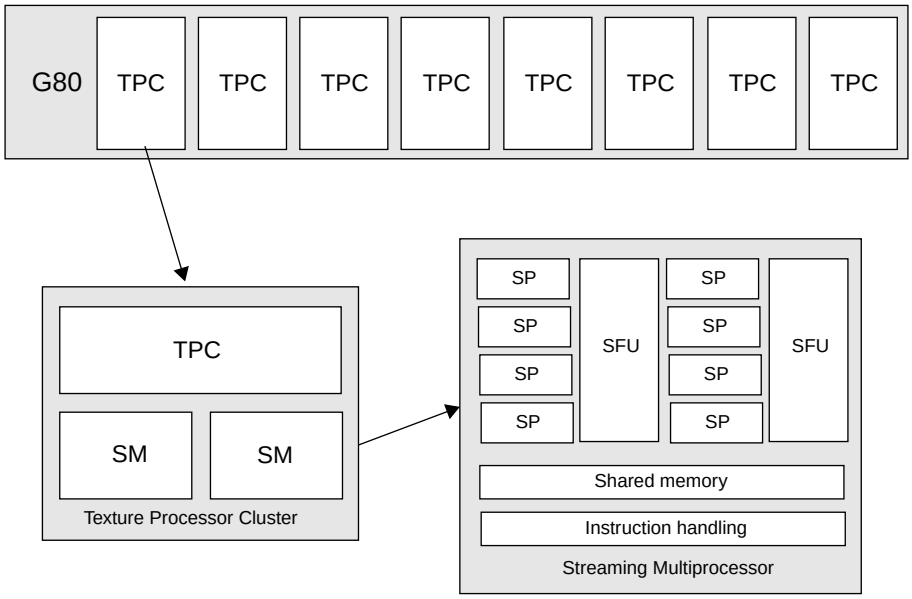


FIGURE 7. Vital components in the G80

We also find the SFUs, *special function units*, which corresponds to the ALU in a CPU, performing operations that we can't afford to provide separately for each SP. There is also instruction handling, register memory for each SP (not shown in the picture), and *shared memory.*

There is more inside the chip, of course. Perhaps most important to us is the thread management. Although the processing is performed as a vector processor, the computing from our point of view is made in threads, each with its own memory. These threads are managed in hardware, with automatic switching between groups of active threads. This thread switching is another key component to the efficiency. However, we don't see anything of this thread handling, we just use it and it seems totally seamless.

When doing shader programming, we see even less of the architecture. We see threads, and that is it. With CUDA, OpenCL etc., we see more. We don't care too much about the number of SMs, because the work is automatically queued over available SMs, but we do specify the split over number of threads per SM, and it is often important to make this split as optimal as possible, to distribute the work to keep the hardware busy, but also to avoid too much synchronization and passing data. We also need to plan the usage of the shared memory. More about that later.

Important note: The queuing of work over a number of SMs, which are not capable of communicating during processing other than by writing and reading global memory, and even then you can't rely on a specific block/work group being active since they are queued, is both effective, simplifying work balancing, as well as complicating many algorithms. Many seemingly simple algorithms are hard to parallelize due to the lack of communication.

So, the numbers we see in a G80 are 16 SMs and 8 cores (SPs) for each SM. These numbers are not magical in any way and change with each new architecture. The subsequent architecture from NVidia, the GT200, sports 10 SPs per SM and 30 SMs in 10 clusters, a straight upscaling.

The Fermi, in 2010, was the next major change in the hardware. Apart from even more upscaling, it surprised us by adding on-chip cache memory, the very thing that NVidia had earlier implied that was a weakness with CPUs. There was also a dramatic increase in double precision floating-point performance, 4x higher than before, implying that it was intended to strengthen the GPU computing use. Otherwise, it was business as usual with some more upscaling, 16 SMs with 32 SPs each, support for 24576 threads.

The Fermi has even had its own name, a Computing Graphics Processing Unit, CGPU. However, marketing-wise this generation was a failure for NVidia, because their strong emphasis on GPU computing let AMD take the lead, and for quite some time, AMD was the brand of choice for gamers. NVidia could not afford falling behind on their core market, so the next generation, Kepler, had more single precision support and NVidia was back on track. The following architectures, Maxwell and Pascal, has kept NVidia high on the price/performance scale again.

But don’t count out AMD. While NVidia was busy taking back what they had lost in the gaming market, AMD took the lead in GPU computing with the R9 series, which for a while was the chip of choice for mining crypto currencies!

I have not said anything about the GPU architectures from AMD, but generally, they follow the lead of NVidia, in order to match the same OpenGL and DirectX generations.

5. Hello World!

This chapter does something completely trivial in a non-trivial way, which is even pretty unique as far as I know: We will take the super simple “Hello World” problem and solve it in parallel, not once but several times, for different GPU computing platforms. Thereby we get a first introduction to all the GPU computing platforms that I intend to cover. As you will see, the solution requires very different amounts of setup, but in the end they use kernels that are quite similar.

Let me define the problem to be solved: “Hello World!” should produce the string “Hello World!” and nothing more. For our purposes, it has to be made in parallel. This is, for all examples in this chapter, made like this: Take the string “Hello ” and add, using one thread per character, the offsets 15, 10, 6, 0, -11, 1, to each character, thereby producing “World!”

This super simple problem, embarrassingly parallel and far too small for a parallel problem, thereby provides us with a first introduction to each platform. Naturally, it is a trivial problem, so we will go further with more interesting things, which is just what the original Hello World! is for. Get your first, simple program, running.

5.1 Hello CUDA

Most CUDA tutorials start with some simple example that is often dubbed “Hello World”, although that is usually an ignorant statement since the examples usually do not output “Hello World!” as their result. This is understandable, since it is not entirely obvious how to make an example of parallel computing which has the sole purpose of producing the string “Hello world!”. However, not without pride, I can present you with exactly that: A program that is short, simple, does perform parallel processing on the GPU using CUDA, and the result is indeed “Hello World!”!

So, here it is, the *real* “Hello world” for CUDA:

```
#include <stdio.h>

const int N = 16;
const int blocksize = 16;

__global__
```

```
void hello(char *a, int *b)
{
   a[threadIdx.x] += b[threadIdx.x];
}

int main()
{
   char a[N] = "Hello \0\0\0\0\0\0";
   int b[N] = {15, 10, 6, 0, -11, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

   char *ad;
   int *bd;
   const int csize = N*sizeof(char);
   const int isize = N*sizeof(int);

   printf("%s", a);

   cudaMalloc( (void**)&ad, csize );
   cudaMalloc( (void**)&bd, isize );
   cudaMemcpy( ad, a, csize, cudaMemcpyHostToDevice );
   cudaMemcpy( bd, b, isize, cudaMemcpyHostToDevice );

   dim3 dimBlock( blocksize, 1 );
   dim3 dimGrid( 1, 1 );
   hello<<<dimGrid, dimBlock>>>(ad, bd);
   cudaMemcpy( a, ad, csize, cudaMemcpyDeviceToHost );
   cudaFree( ad );
   cudaFree( bd );

   printf("%s\n", a);
   return EXIT_SUCCESS;
}
```

I hope the source itself explains what it is doing; it takes a string and an array of offsets to produce “World!” from “Hello “. But a few more clarifications are called for.

What you see here is, in one and the same file, both CPU and GPU code. This integration is very elegant. The amount of code to compile and launch the GPU kernel is extremely small.

The __global__ code is the kernel, executed on the GPU, in parallel. Note the threadIdx.x. That is the thread identifier, which must be used to calculate where in the data to operate. “Real” CUDA programs use both thread and block identifiers.

We allocate memory on the GPU from the CPU, using cudaMalloc. We can then upload and download data with cudaMemcpy, using the arguments cudaMemcpyDeviceToHost or cudaMemcpyHostToDevice to denote the copying direction. Finally, we can dispose of GPU memory using cudaFree.

One of the most challenging issues when you start with CUDA is the concepts of grid, block and thread. The grid is the whole computing, which is split into a number of blocks, which each contains a number of threads. This division scheme describes how the computing is distributed over the GPU.

The weird statement

```
hello<<<dimGrid, dimBlock>>>(ad, bd);
```

is the actual execution of the kernel.

This version of the Hello World for CUDA works on any CUDA version. It should be noted that a simpler version exists for newer CUDA versions, using managed memory. More about that later (chapter 9.6).

5.2 Hello OpenCL

Hello World for OpenCL is substantially longer. However, much of the complexity is the setup.

```
#include <stdio.h>
#include <math.h>
#ifdef __APPLE__
   #include <OpenCL/opencl.h>
#else
   #include <CL/cl.h>
#endif

const char *KernelSource = "\n" \
"__kernel void hello(              \n" \
"   __global char* a,          \n" \
"   __global char* b,          \n" \
"   __global char* c,          \n" \
"   const unsigned int count)  \n" \
"{                             \n" \
"   int i = get_global_id(0);  \n" \
"   if(i < count)              \n" \
"       c[i] = a[i] + b[i];   \n" \
"}                              \n" \
"\n";

#define DATA_SIZE (16)

int main(int argc, char** argv)
{
   int err;             // error code returned from api calls
   cl_device_id device_id; // compute device id
   cl_context context; // compute context
   cl_command_queue commands;  // compute command queue
    cl_program program;                // compute program
    cl_kernel kernel;                  // compute kernel
   cl_mem input;                    // device memory used for the input
array
    cl_mem input2;                     // device memory used for the
input array
    cl_mem output;                     // device memory used for the
output array
    size_t global;                     // global domain size for our
calculation
    size_t local;                     // local domain size for our cal-
culation
```

```
   int i;
    unsigned int count = DATA_SIZE;

   // Input data
   char a[DATA_SIZE] = "Hello \0\0\0\0\0\0";
   char b[DATA_SIZE] = {15, 10, 6, 0, -11, 1, 0, 0, 0, 0, 0, 0, 0, 0,
0};
   // Output data
   char c[DATA_SIZE];

   // Print original data
   printf("%s", a);

   cl_platform_id platform;
   unsigned int no_plat;
   err =  clGetPlatformIDs(1,&platform,&no_plat);

   // Where to run
   err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device_id,
NULL);
   if (err != CL_SUCCESS) return -1;
   context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
   if (!context) return -1;
   commands = clCreateCommandQueue(context, device_id, 0, &err);
   if (!commands) return -1;

   // What to run
   program = clCreateProgramWithSource(context, 1, (const char **) &
KernelSource, NULL, &err);
   if (!program) return -1;

   err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
   if (err != CL_SUCCESS) return -1;
   kernel = clCreateKernel(program, "hello", &err);
   if (!kernel || err != CL_SUCCESS) return -1;

   // Create space for data and copy a and b to device (note that we
could also use clEnqueueWriteBuffer to upload)
   input = clCreateBuffer(context,  CL_MEM_READ_ONLY |
CL_MEM_USE_HOST_PTR,  sizeof(char) * DATA_SIZE, a, NULL);
   input2 = clCreateBuffer(context,  CL_MEM_READ_ONLY |
CL_MEM_USE_HOST_PTR,  sizeof(char) * DATA_SIZE, b, NULL);
   output = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(char) *
DATA_SIZE, NULL, NULL);
   if (!input || !output) return -1;

   // Send data
   err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);
   err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &input2);
   err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &output);
   err |= clSetKernelArg(kernel, 3, sizeof(unsigned int), &count);
   if (err != CL_SUCCESS) return -1;

   local = DATA_SIZE;

   // Run kernel!
   global = DATA_SIZE; // count;
   err = clEnqueueNDRangeKernel(commands, kernel, 1, NULL, &global,
&local, 0, NULL, NULL);
```

```
   if (err != CL_SUCCESS) return -1;

   clFinish(commands);

   // Read result
   err = clEnqueueReadBuffer( commands, output, CL_TRUE, 0, sizeof(char)
* count, c, 0, NULL, NULL );
   if (err != CL_SUCCESS) return -1;

// Print result
      printf("%s\n", c);

   // Clean up
   clReleaseMemObject(input);
   clReleaseMemObject(output);
   clReleaseProgram(program);
   clReleaseKernel(kernel);
   clReleaseCommandQueue(commands);
   clReleaseContext(context);
   return 0;
}
```

In this case, the kernel is defined as a set of text strings at the top of the program. A real OpenCL program will rather put that in a separate file.

5.3 Hello GPGPU

In this section, we will have a look at how to write Hello World! for a fragment shader in the OpenGL pipeline. This means that the entire computation takes place by drawing bogus graphics and make computations per fragment, that is per pixel in the generated geometry.

This example exists in no less than three variants, one for old-style OpenGL, which I try to avoid, one with few dependencies, and one that uses my lab material for simplifying shader compilations and model/buffer handling (loadobj.c, GL_utilities.c and MicroGlut).

```
// Hello World in a shader.
// Kind of twisted, since it uses signed chars.
// Modern OpenGL, using my lab material for simplicity.

#include <stdio.h>
#include <OpenGL/gl3.h>
#include "MicroGlut.h"
#include "GL_utilities.h"
#include "loadobj.h"
#include <sys/times.h>
// uses framework Cocoa

// Add offset (texUnit2) to string (texUnit)
// Negative values end up as > 0.5, adjust them!
static const char *fragSource =
{
"#version 150\n"
"uniform sampler2D texUnit;"
"uniform sampler2D texUnit2;"
"out vec4 outColor;"
```

```
"in vec2 texCoord;"
"void main(void)"
"{"
"   vec4 texVal  = texture(texUnit, texCoord);"
"   vec4 texVal2  = texture(texUnit2, texCoord);"
"   if (texVal2.r > 0.5) texVal2.r -= 1.0;"
"   if (texVal2.g > 0.5) texVal2.g -= 1.0;"
"   if (texVal2.b > 0.5) texVal2.b -= 1.0;"
"   if (texVal2.a > 0.5) texVal2.a -= 1.0;"
"   outColor = texVal + texVal2;\n"
"}"
};

// Vertex shader, pass position and texcoord
char *vs =
{
"#version 150\n"
"in  vec3 inPosition;"
"in vec2 inTexCoord;"
"out vec2 texCoord;"
"void main()"
"{"
"  texCoord = inTexCoord;"
"  gl_Position = vec4(inPosition, 1.0);"
"}"
};

GLfloat vertices[] = {-1.0f,-1.0f,0.0f,
                      -1.0f,1.0f,0.0f,
                      1.0f,-1.0f,0.0f,
                      1.0f,1.0f,0.0f };

GLfloat texcoord[] = {0.0f, 1.0f,
                      0.0f, 0.0f,
                      1.0f, 1.0f,
                      1.0f, 0.0f};
unsigned int indices[] = {0,1,2, 2,1,3};

Model *m;
GLuint shader;

// declare texture size, the actual data will be a vector
// of size texSize*1*4 = N

#define N 16
// test data
char a[N] = "Hello \0\0\0\0\0\0\0\0\0\0";
char b[N] = {15, 10, 6, 0, -12, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
unsigned char c[N];
#define texSize 4

void display()
{
   DrawModel(m, shader, "inPosition", NULL, "inTexCoord");
   glutSwapBuffers();

   printf("%s",a);
   // and read back
   glReadPixels(0, 0, texSize, 1, GL_RGBA,GL_UNSIGNED_BYTE,c);
```

```
   // print out results
   printf("%s\n",c);
   exit(0);
}

// Not exported by GL_utilties:
GLuint compileShaders(const char *vs, const char *fs, const char *gs,
const char *tcs, const char *tes,
                        const char *vfn, const char *ffn, const char
*gfn, const char *tcfn, const char *tefn);

GLuint LoadTexture(unsigned char *a, GLuint texunit)
{
   GLuint tex;
   glActiveTexture(texunit);
   glGenTextures (1, &tex);
   glBindTexture(GL_TEXTURE_2D,tex);
   glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA,
               texSize,1,0,GL_RGBA,GL_UNSIGNED_BYTE, a);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
   return tex;
}

int main(int argc, char **argv)
{
// set up glut to get valid GL context and
// get extension entry points
    glutInit (&argc, argv);
   glutInitContextVersion(3, 2);
   glutInitWindowSize (4, 1);
    glutCreateWindow("TEST1");

// create string texture
    GLuint tex = LoadTexture(a, GL_TEXTURE0);
// create offset texture
    GLuint offtex = LoadTexture(b, GL_TEXTURE1);

// Compile shader
   shader = compileShaders(vs, fragSource, NULL, NULL, NULL, "vs", "fs",
NULL, NULL, NULL);

// Inform shader of texture units
   glUniform1i(glGetUniformLocation(shader, "texUnit1"), 0); // Texture
unit 0
   glUniform1i(glGetUniformLocation(shader, "texUnit2"), 1); // Texture
unit 1

   m = LoadDataToModel(vertices, NULL, texcoord, NULL, indices, 4, 6);

// Ask for a redraw
   glutDisplayFunc(display);
   glutMainLoop();
   exit(0);
}
```

Some notes: We are using all four channels of the texels, which is why the texture width is 1/4 of the data size. We also need to mess a bit with the data since we upload to unsigned chars. This problem disappears when using floating-point buffers. See chapter 11.

5.4 Hello Compute Shaders

OpenGL Compute Shaders is a relatively new development.

In this case, I chose to use MicroGlut for creating an OpenGL context. On the computer-graphics.se page, you can also find a stand-alone Linux version complete with context creation.

Also note that this code also contains a file loader, so the kernel can be in a separate file. This is desirable for the OpenCL code as well.

Like with the fragment shader version, I have skipped some code for compiling shaders, loading a file and printing out errors.

5.4.1 Main program

Here is the main program code. It sets up an OpenGL context, loads and compiles a shader (by standard code, not included).

```
int main(int argc, char **argv)
{
// Let GLUT create a GL context
   glutInit (&argc, argv);
    glutInitContextVersion(4, 4); // Failed with 4.5 on my PC. The com-
pute shader works even on old-style GL!
   glutCreateWindow("Hello");


// Load and compile the compute shader
    GLuint p =loadShader("hello.cs");

   GLuint ssbo, ssbo2; //Shader Storage Buffer Object

   // Some data
#define N 16
   char a[N] = "Hello \0\0\0\0\0\0";
   int ac[N];
   int b[N] = {15, 10, 6, 0, -11, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
   int *ptr;
   int i;

   printf("%s", a);

// PROBLEM: No bytes in shaders!
// I chose to package to int on the CPU.
// Convert string to int:
   for (i = 0; i < N; i++) ac[i]=a[i];

// Create buffer, upload data
   glGenBuffers(1, &ssbo);
   glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo);
   glBufferData(GL_SHADER_STORAGE_BUFFER, 16 * sizeof(int), &ac,
GL_STATIC_DRAW);

// Tell it where the input goes!
```

```
// The "5" matches a "layuot" number in the shader.
// (Can we ask the shader about the number? I must try that.)
   glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 5, ssbo);

// Same for the other buffer, offsets, ID 6
   glGenBuffers(1, &ssbo2);
   glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo2);
   glBufferData(GL_SHADER_STORAGE_BUFFER, 16 * sizeof(int), &b,
GL_STATIC_DRAW);
   glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 6, ssbo2);

// Get rolling!
    glDispatchCompute(1, 1, 1); //Work groups launch

// Get data back!
   glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo);
   ptr = (int *)glMapBuffer(GL_SHADER_STORAGE_BUFFER, GL_READ_ONLY);
// Convert int to string:
   for (i=0; i < 16; i++)
   {
      a[i] = ptr[i];
   }
   printf("%s\n", a);

}
```

The main program should be of most interest, the rest is reusable code. First we create an OpenGL context. We load and compile the shader. Above, we find code that will print out error messages from the compilation. Then we create buffers on the GPU and upload the data to them, and tell the shader about the buffers.

Then we are ready to run and call glDispatchCompute(). Finally, download the result.

5.4.2 Kernel

As for all other cases, the kernel itself is comfortably simple.

There are a few notable peculiarities here:

You may note that the work group size is defined by the compute shader, not by the main program. However, this is not a limitation, but rather a freedom, because we can also do that from the CPU.

A more disturbing limitation is that a compute shader does not allow byte-sized variables! Therefore, the CPU part converts the data to standard-sized integers. Given that, the shader itself is as simple as the earlier ones.

```
#version 430
#extension GL_ARB_compute_shader : enable
//#extension GL_ARB_shader_storage_buffer : enable
#define width 16
#define height 1

// Compute shader invocations in each work group
```

```
layout(std430, binding = 6) buffer offsbuf {int offs[];};
layout(std430, binding = 5) buffer strbuf {int str[];};
layout(local_size_x=width, local_size_y=height) in;

//Kernel Program
void main()
{
   int i = int(gl_GlobalInvocationID.x);
   str[i] = str[i] + offs[i];
}
```

5.5 Direct Compute and Vulkan

Although we do acknowledge Direct Compute and Vulkan to be significant frameworks for our purposes, they are left out in order not to make the focus too scattered and this chapter not too repetitive. See chapter 13 for a discussion and simple example of Direct Compute.

6. CUDA

Our first pick for learning GPU computing is CUDA, for the simple reason that it is the easiest starting point. It sports an integrated code model that makes simple programs very simple, which is, as you saw in chapter 5, not quite the case for the others. For bigger problems, the difference rapidly gets insignificant, but for an easy start, let us use CUDA.

CUDA is officially an acronym for “Compute Unified Device Architecture” (but see the cover for my interpretation). It is developed by NVidia and is only available on NVidia boards. A G80 or better GPU architecture is required, and as you may expect, the newer hardware, the newer CUDA version you can use. It is designed to hide the graphics heritage and add control and flexibility.

Since this means that computing is taken place outside the domain of your CPU, we can consider the following model for our computing:

- 1. Upload data to GPU
- 2. Execute kernel
- 3. Download result

The same holds for other platforms, like shader-based solutions and OpenCL However, a major difference to other platforms is that CUDA has integrated source, which means that the source of host and kernel code can be in the same source file! This makes the most difference for small examples, and that is also why we start with it.

Since CPU and GPU code can reside in the same file, CUDA uses special modifiers to identify kernel code. We will soon see how that looks.

Thus, CUDA is both an architecture (essentially the G80 architecture) and a C/C++ extension. The basic model is that we spawn a large number of threads. These threads will be ran in parallel, or rather *virtually* in parallel. They will not all be computed in parallel, they will be processed in batches, as much as the GPU can handle at a time. This is exactly what happens in graphics rendering as well; fragments and vertex computations not quite executed in parallel, but in batches. The difference is that in CUDA, these batches are more visible to you.

Compared to a graphics program, a CUDA program looks much more like an ordinary C program! Even though the hardware is made to process pixels, we don't see them any more, just arrays of whatever data we want to work with.

6.1 Simple CUDA example

Here follows a working, compileable example. This is as simple as Hello World!, but I believe that another simple example doesn't hurt, and it is even a little bit simpler since my goal was to make a truly minimal example. The central components remain the same.

```
#include <stdio.h>

const int N = 16;
const int blocksize = 16;

__global__
void simple(float *c)
{
   c[threadIdx.x] = threadIdx.x;
}

int main()
{
   int i;
   float *c = new float[N];
   float *cd;
   const int size = N*sizeof(float);

   cudaMalloc( (void**)&cd, size );
   dim3 dimBlock( blocksize, 1 );
   dim3 dimGrid( 1, 1 );
   simple<<<dimGrid, dimBlock>>>(cd);
   cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );
   cudaFree( cd );

   for (i = 0; i < N; i++)
      printf("%f ", c[i]);
   printf("\n");
   delete[] c;
   printf("done\n");
   return EXIT_SUCCESS;
}
```

In the code, you can easily spot the computing kernel, a thread identifier, allocation of GPU memory, specification of 1 block and 16 threads, the kernel call, the readback of data to CPU, and deallocations.

6.2 Modifiers for code

Since we are mixing CPU and GPU code, we must instruct the compiler on what is what. Three modifiers are built into CUDA to specify how code should be used:

__global__ executes on the GPU, invoked from the CPU. This is the entry point of the kernel.

__device__ is local to the GPU, not callable from the CPU. You use this for subroutines and methods used by the main kernel, the __global__.

__host__ is CPU code. This is superfluous, since it is the default. You may use it for making your code more readable.

The modifiers are illustrated in Figure 8.

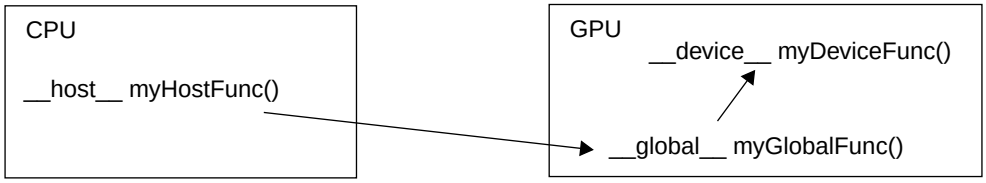


FIGURE 8. CUDA code modifiers

6.3 Memory management

The memory management calls are similar to the calls in the C libraries, where we have malloc(), calloc() and free(), but now they are calls done by the CPU to allocate and manage memory on the GPU.

```
cudaMalloc(ptr, datasize)
```

This allocates a chunk of memory on the GPU.

```
cudaFree(ptr)
```

This frees the memory allocated by cudaMalloc.

```
cudaMemCpy(dest, src, datasize, arg)
```

This copies data between CPU and GPU, datasize bytes from src to dest. This is rather peculiar, you must specify direction using these constants:

```
arg = cudaMemcpyDeviceToHost
or cudaMemcpyHostToDevice
```

These constants may seem unnecessary, but the dest and src pointers can not be identified as CPU or GPU memory so we need to keep track of that ourselves.

The easiest way to manage memory with CUDA, if you are on a fairly recent GPU (CUDA capability 6) is to take advantage of *unified memory*. This allows you to access the same memory from CPU and GPU. You must still allocate it with CUDA calls.

```
cudaMallocManaged(ptr, datasize)
```

For this kind of memory, no cudaMemCpy is needed, just pass the pointer.

To make matters even easier, you can declare a variable __managed__, which will then have the same capability. We will discuss this further in chapter 9.6.

6.4 Kernel execution

The kernel is executed by a call with very odd syntax:

```
simple<<<griddim, blockdim>>>(...)
```

That is KERNELNAME<<<argument1, argument2>>>(some other arguments)

The arguments in the parenthesis are the argument sent to the kernel entry point, the __global__. The ones within <<<>>> are something else, they specify the size of the computation, and how it should be split into blocks and threads.

The *griddim* argument specifies the number of blocks, and the *blockdim* argument specifies number of threads per block.

When working with blocks and threads in the kernel, you need to use the built-in variables *threadIdx*, *blockIdx*, *blockDim* and *gridDim*, which tells the thread what thread numer it has, in what block, and the dimensions of each (as specified above).

If you are used to OpenGL, you may note that no prefix is used for built-in variables, like GLSL does. We will look further into blocks and threads in chapter 6.7.

6.5 Compiling Cuda

If you start from the CUDA development kit, you will find that the CUDA examples are compiled by gigantic makefiles. Don't panic, they are just auto-generated makesfiles with a lot of unnecessary fluff. It all boils down to calling a single compiler, *nvcc*.

nvcc is nvidia's CUDA compiler. On Unix systems you will usually find it in

```
/usr/local/cuda/bin/nvcc
```

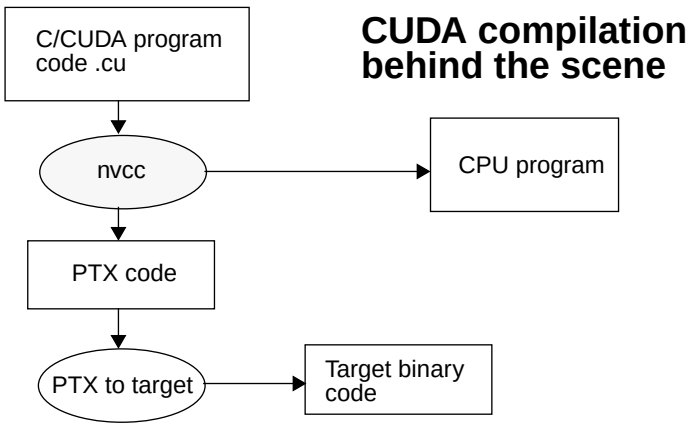


FIGURE 9. CUDA compilation

Source files are suffixed .cu. In order to compile a simple example like "simple" above, all you need to do is a simple command-line like this:

```
nvcc simple.cu -o simple
```

Of course it is not quite that simple for all cases. Like with all compilers, there are options for including libraries as well as other options, enabling/disabling language features etc.)

However, this simplicity hides a much more complex compilation than we are used to. What happens behind the scene is illustrated in Figure 9.

The nvcc compiler doesn't just compile a program, it splits the code into CPU and GPU parts, sends the CPU part to GCC/G++, and compiles the other to something called PTX code. This code is an intermediate code, which is compiled to your target GPU when executed. This two-step process is there to allow a compilation for any kind of (supported) GPU, while still allowing each GPU to have the instruction set it needs.

6.5.1 Compiling CUDA for larger applications

It may seem like your CUDA-using application should be written entirely as .cu files. Anyone involved in a large project that needs CUDA acceleration late in the process will realize that that is out of the question. Fortunately, this is not the situation, we do not need to port the rest of the project.

For instance, if your program is a C or C++ program, you can compile it with gcc as usual, and then link with the CUDA parts. You use nvcc for the .cu files and gcc for .c/.cpp files. You can mix in any language that produces code that can be linked with C/C++. You may do the final linking with gcc or g++. In any event, the final linking must include C++ runtime libs.

This gives us one little hint: .cu files are really C++. Indeed, .cu allows us to use C++ classes, even in the kernel!

6.5.2 Example of multi-unit compilation

Thus, multi-unit compilation is quite easy. Here follows a simple example with one .c file and one .cu file. They are called cudademokernel.cu and cudademo.c.

I compile them with

```
nvcc cudademokernel.cu -o cudademokernel.o -c
gcc -c cudademo.c -o cudademo.o -I/usr/local/cuda/include
```

Then I link them. To make it simple, I use g++ to include the C++ runtime.

```
g++ cudademo.o cudademokernel.o -o cudademo -L/usr/local/cuda/lib -lcuda
-lcudart -lm
```

I also included some more common linking options, like the location of the CUDA runtime library and the math library. You will need them in just about any nontrivial example.

6.5.3 Compiling for MacOSX

Compiling for MacOSX is a bit different than Linux. Since you add most libraries as frameworks, with the -framework option, which nvcc doesn't have, this information must be added as linker options.

Thus, a compilation line may look something like this:

```
/usr/local/cuda/bin/nvcc program.cu -L /usr/local/cuda/lib -lcudart -o
program -Xlinker -framework,GLUT,-framework,OpenGL
```

for a program "program.cu" that uses the frameworks GLUT and OpenGL.

6.6 Executing a Cuda program

CUDA programs are executed like any other program. From the command-line, the program simple is launched with

```
./simple
```

Often, this is all you need. However, this depends on your system, your OS as well as CUDA version. On some Linux installations, you may need to set environment variable to find Cuda renting.

```
export DYLD_LIBRARY_PATH=/usr/local/cuda/lib:$DYLD_LIBRARY_PATH
```

This may look a little different between different systems and CUDA versions.

6.7 Computing with CUDA

Let us now look a bit closer to the internal organization of CUDA. We have touched upon blocks and threads. The overall processing organization can be summarized as follows:

- 1 warp = 32 threads
- 1 kernel - 1 grid
- 1 grid - many blocks
- 1 block - 1 SM
- 1 block - many threads

6.8 Warps

A *warp* is the minimum number of data items/threads that will actually be processed in parallel by a CUDA capable device. This number may vary with different GPUs but has been surprisingly stable at 32.

We usually don't care so much about warps but rather discuss threads and blocks, but warps are useful to take into account when optimizing.

6.9 Kernel

The kernel is the GPU program. We usually consider one at a time, although that is not a necessary limitation in modern GPUs. The kernel is mapped to a computing *grid*.

6.10 Grid, blocks and threads

A *grid* is the top level of the computing structure. It consists of a number of *blocks*. Any running block is mapped to one SM. However, there may be many more blocks than SMs. It is recommended that you use more blocks than SMs, so they can be organized in a queue, and processed as SMs get freed up.

Every block consists of a number of *threads*. It is recommended than the number of threads in a block is a multiple of 32.

You should use many threads and many blocks! More than 200 blocks are recommended, but it is virtually unlimited. The number of threads is more limited, but you should use plenty of them. About 256 tends to be optimal.

Thus, grid, blocks and threads form a hierarcical model, illustrated in Figure 10.

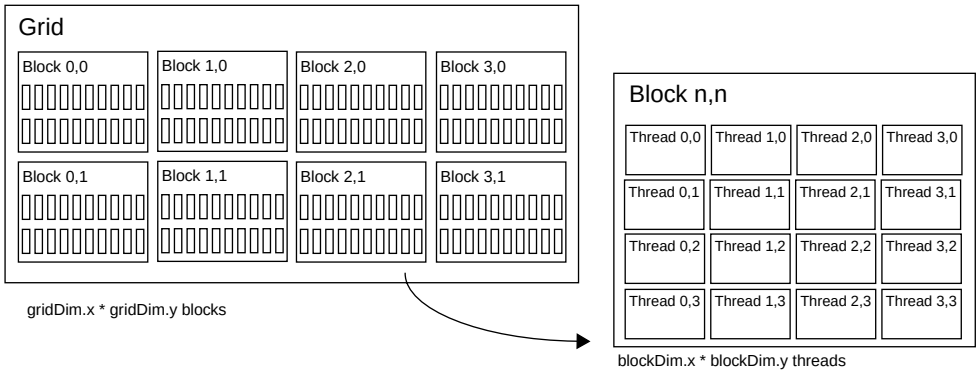


FIGURE 10. Hierarchy of grid, blocks and threads

In the figure, it looks like the grids and blocks are organized in 2 dimensions. This is not strictly true, but also not strictly false. It is legal to use up to three dimensions, but the range in the third dimension is more limited than for the two first, so you will often just use two.

6.11 Indexing data with thread/block IDs

In order to know which thread that is running, your kernel code should inspect the built-in variables blockIdx, blockDim and threadIdx and compute a n index from them, indicating what data it should process. Here is another simple kernel, which does this in one dimension, supporting multi-block computing by calculating an index from both thread and block numbers.

```
// Kernel that executes on the CUDA device
__global__ void square_array(float *a, int N)
{
   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   if (idx<N) a[idx] = a[idx] * a[idx];
}
```

The host part of this program looks like this:

```
// main routine that executes on the host
int main(int argc, char *argv[])
{
   float *a_h, *a_d;// Pointer to host and device arrays
   const int N = 10;// Number of elements in arrays
   size_t size = N * sizeof(float);
   a_h = (float *)malloc(size);
   cudaMalloc((void **) &a_d, size);   // Allocate array on device
// Initialize host array and copy it to CUDA device
   for (int i=0; i<N; i++) a_h[i] = (float)i;
   cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
// Do calculation on device:
   int block_size = 4;
   int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);
   square_array <<< n_blocks, block_size >>> (a_d, N);
// Retrieve result from device and store it in host array
   cudaMemcpy(a_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
// Print results and cleanup
   for (int i=0; i<N; i++) printf("%d %f\n", i, a_h[i]);
   free(a_h); cudaFree(a_d);
}
```

Note that there is now a variable number of blocks. We may also note that the block size is only four, which is much too small for a good computation.

The vital part here is the index calculation, using the block and thread numbers. In this case, we only use one dimension. For bigger problems, you primarily use two, X and Y.

6.12 Julia example

The Julia set is a family of fractals based on iterating of complex functions. It is, together with the Mandelbrot, one of the most famous fractals. It was described in Volume 1, so here I will only briefly cite its definition. It is created by iterating the function

$$z_{k+1} = z_k^2 + L$$

where L is a constant. Every pixel is scaled and defines the starting z, and the output is the number of iterations until z is outside a certain radius.

Thus, each thread will compute a single pixel. We should assume that the image can be at least a million pixels, possibly more. A million threads or more? Yes, and that is no problem at all for the GPU.

This is a bigger problem than before, processing a number of iterations for each pixel in an entire image. Addressing calculation should be done in 2D. Not only do we avoid running out of allowed range for the X coordinate, it is also very convenient since each pixel comes in 2D.

The problem requires considerable computations for every pixel. However, each computation is independent, which makes this trivial to implement in parallel, and we can easily get full performance out of the GPU. We say that the problem is *embarrassingly parallel*. Not only is the problem easy to implement in parallel, the amount of computations for each thread also means that the problem is not memory limited.

Furthermore, the problem is somewhat tricky when performed with a small number of threads, on a multi-core CPU. In that case, it is important to do proper load balancing to utilize each core optimally. On the GPU, however, this problem is must smaller. The massive parallelism makes the problem so fine-grained that it practically balances itself. There is a certain waste in areas where deep iterations are processed together (in the same warp - more about that later) with areas that diverge quickly, but for most of the fractal, we can finish entire warps quickly and spend more time on the expensive parts automatically.

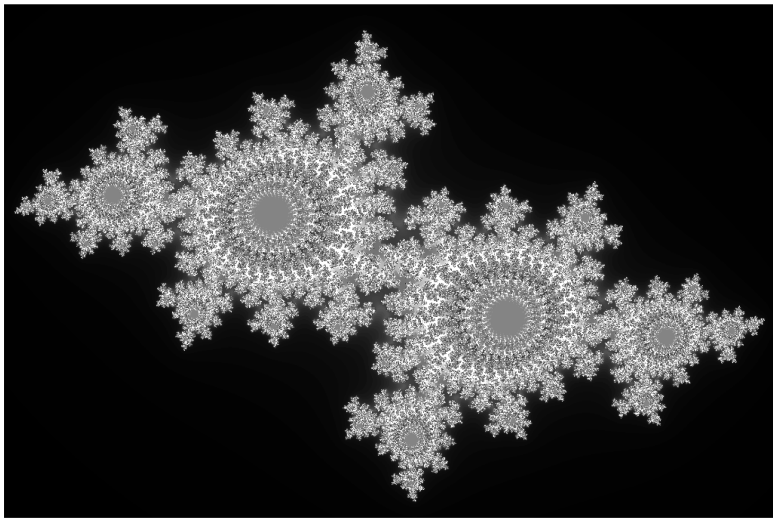


FIGURE 11. The Julia fractal, rendered in real time

The result is shown in Figure 11. We can hardly wish for a nicer problem for showing off the computational power of the GPU!

For this demo, we use a simple OpenGL output. This will waste some performance in passing data back and forth, but we will still get amazing performance. See further chapter 15.2.

I will only include the kernel and the julia() subroutine here. The kernel calls the function julia() which performs the calculation. Notice how the __device__ modifier is used to identify GPU code.

```
__device__ int julia( int x, int y, float r, float im)
```

```
{
    const float scale = 1.5;
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);

    cuComplex c(r, im);
    cuComplex a(jx, jy);

    int i = 0;
    for (i=0; i<200; i++)
    {
        a = a * a + c;
        if (a.magnitude2() > 1000)
            return i;
    }

    return i;
}

__global__ void kernel( unsigned char *ptr, float r, float im)
{
    // map from blockIdx to pixel position
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    int offset = x + y * DIM;

    // now calculate the value at that position
    int juliaValue = julia( x, y, r, im );
--- calculate colors ---
    ptr[offset*4 + 0] = red;
    ptr[offset*4 + 0] = green;
    ptr[offset*4 + 0] = blue;
    ptr[offset*4 + 3] = 255;
}
```

We conclude that the Julia demo uses many blocks as well as many treads in each block. This distribution should be made in a way that makes sure all hardware is in use as much as possible. We also see an index calculation by thread and block. We have made it to the first example that makes significant computations with large data output.

Let us also draw some conclusions about indexing: Every thread does its own calculation for indexing memory! This may seem wasteful but this part of code is expected so the GPU can optimize it well. We use blockIdx, blockDim, threadIdx in 1, 2 or 3 dimensions, with 2 dimensions being the typical choice.

7. Memory access

The problem of memory access is vital in GPU computing. Memory access is all too often the bottleneck of a parallel computation, so we need to do what we can to handle it. There are several memory types and memory access paths in a GPU, more than you may expect.

In this chapter, we will mainly deal with global and shared memory. A later chapter will discuss more memory access, including the important topic of coalescing.

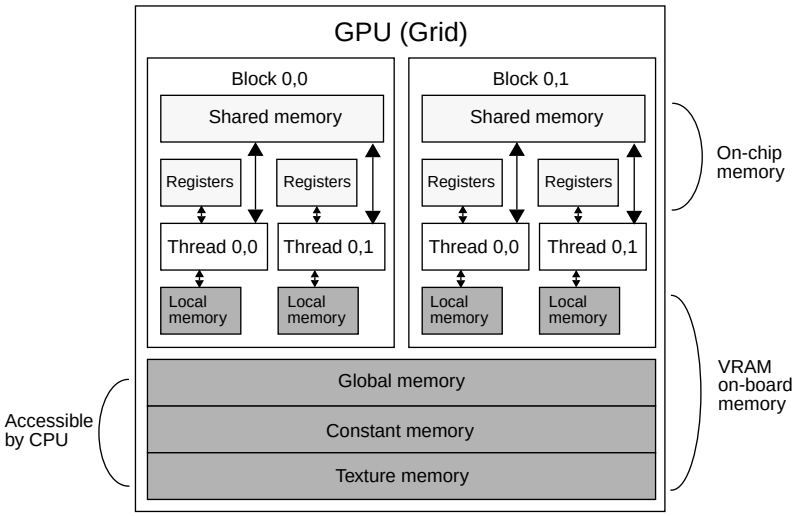


FIGURE 12. Memory model for CUDA

We can identify the following memory types/access paths:

- Global
- Shared
- Constant (read only)
- Texture cache (read only)
- Local
- Registers

When writing a CUDA program, you usually don't care so much about these to begin with, but the difference is so big that you will soon want to take at least some steps to optimize. The first step is to learn about shared memory.

7.1 Global memory

Global memory is plentiful, but you should expect it to be slow, much slower than most of the system. A global memory access has a whopping 400-600 cycles latency! The bandwidth is very good but the response time for a single access is not as impressive,

Due to this latency, it often pays to use *shared memory* fast temporary storage, as a kind of "manual cache". You should also make sure global memory access is ordered properly, with *coalescing* for the memory accesses! That means continuous addresses, aligned on a power of 2 boundary, addressed in order of thread numbers... See chapter 9.1.

7.2 Shared memory

Shared memory is a small memory bank, local to each SM, and thereby to each block. The amount of data is not extremely small, but with something like 48k-96k per block in a modern GPU, it clearly isn't a big data buffer but rather a small buffer for temporary use. But this use is important!

This memory is roughly 10 times faster to access than global memory. Let us consider the case where the global memory has a latency of 440 cycles (actual numbers from the Tesla K20/Kepler GPU) and the shared memory 48 cycles (same GPU). This means that if you need to access a certain data item 8 times, if you do it directly in global memory, it will take 3520 cycles, but if we read the item to shared memory, it will only take 824 cycles!

This is disregarding cache and queuing effects in the GPU, so the number is not exact, but it can give you a rough estimate of the expected effect.

In CUDA, shared memory is declared *__shared__*:

```
__shared__ float a[SOMESIZE];
```

In OpenCL, shared memory is called *local memory* and is declared *__local*:

```
__local float a[SOMESIZE];
```

In OpenGL compute shaders, shared memory is declared *shared*:

```
shared float a[SOMESIZE];
```

7.3 Example: Matrix multiplication

The task of multiplying two large matrices is one of the most common examples in GPU computing. You will find it in the CUDA NVidia programming guide [3], and for a reason. It gives us a pure and (relatively) simple case where shared memory matters a lot.

To multiply two N*N matrices, every item will have to be accessed N times! A naive implementation uses $2N^3$ global memory accesses! How can we improve on that? The computations are simple, so it is likely that we have a memory bound operation.

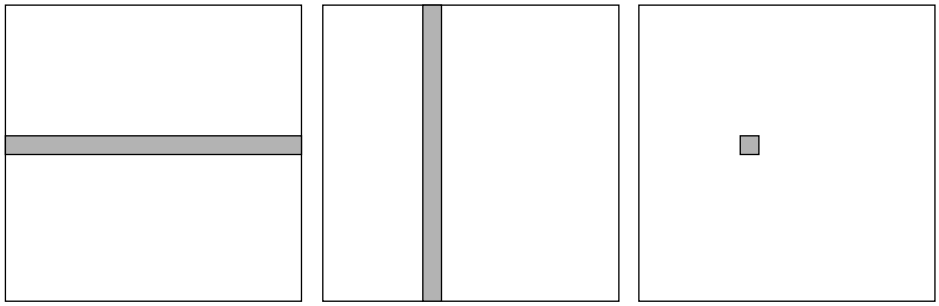


FIGURE 13. Matrix multiplication, output is the dot product of a row tiles a column

7.3.1 Matrix multiplication on CPU

This is a simple triple “for” loop. This is so wonderfully simple that you want to believe that it is good, even optimal. Even on the CPU, we can do better (i.e. split into multiple threads), but our goal is to make a massively parallel version.

```
void MatrixMultCPU(float *a, float *b, float *c, int theSize)
{
   int sum, i, j, k;

   // For every destination element
   for(i = 0; i < theSize; i++)
      for(j = 0; j < theSize; j++)
      {
         sum = 0;
         // Sum along a row in a and a column in b
         for(k = 0; k < theSize; k++)
            sum = sum + (a[i*theSize + k]*b[k*theSize + j]);
         c[i*theSize + j] = sum;
      }
}
```

7.3.2 Naive GPU version

The typical first try when porting a CPU program to the GPU is to replace outer loops by thread indices. So, let us do just that.

```
__global__ void MatrixMultNaive(float *a, float *b, float *c, int the-
Size)
{
   int sum, i, j, k;

   i = blockIdx.x * blockDim.x + threadIdx.x;
   j = blockIdx.y * blockDim.y + threadIdx.y;

   // For every destination element
   sum = 0;
   // Sum along a row in a and a column in b
```

```
    for(k = 0; k < theSize; k++)
        sum = sum + (a[i*theSize + k]*b[k*theSize + j]);
    c[i*theSize + j] = sum;
}
```

Looks good, right? Short and easy to understand. Yes, but every thread makes 2N global memory accesses! This can be significantly reduced using shared memory.

7.3.3 Optimized GPU version

In order to optimize, we can first make this observation: If we want to compute a group of output pixels, they will share some data. Every pixel in the same row will share the same input row in the left input matrix, and the same for pixels in the same column and the right matrix. See Figure 14.

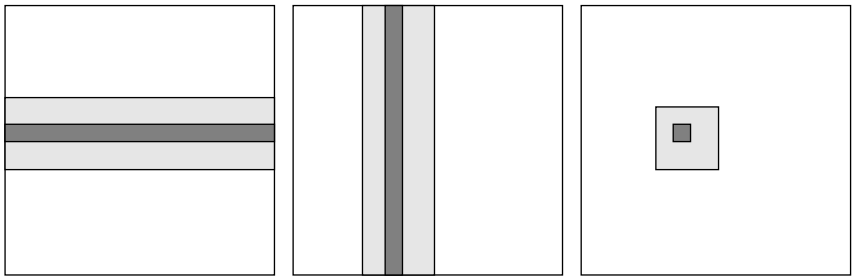


FIGURE 14. Strategy: For a patch of output, a limited number of rows and columns contribute

We can take advantage of this. What we need to do is to store data that will be used multiple times in shared memory. Shared memory is local to the SM, and thereby much faster.

However, assuming that the matrices can be very large, we must limit ourselves to a part of the input data that we know fits in shared memory. We can handle that by reading a part of the input data at a time. See Figure 15.

Our strategy is like this:

1) We split the input (A, B) and output (C) matrices into patches of equal size that will fit in shared memory. This split is for a specific output patch (C) that will be produced by one single block.

2) Each thread in the block is now responsible for reading *one single item* of A and B. Note that this does not have to me an item that the tread itself needs! For some problems, it may be more convenient to have each thread reading a part of the data, but the general principle is to split the reading to several threads.

3) Synchronize! All threads in the block must have read its data before continuing.

4) Compute and accumulate the dot product result for this part of the matrix only.

5) Synchronize again, continue with the next two input patches.

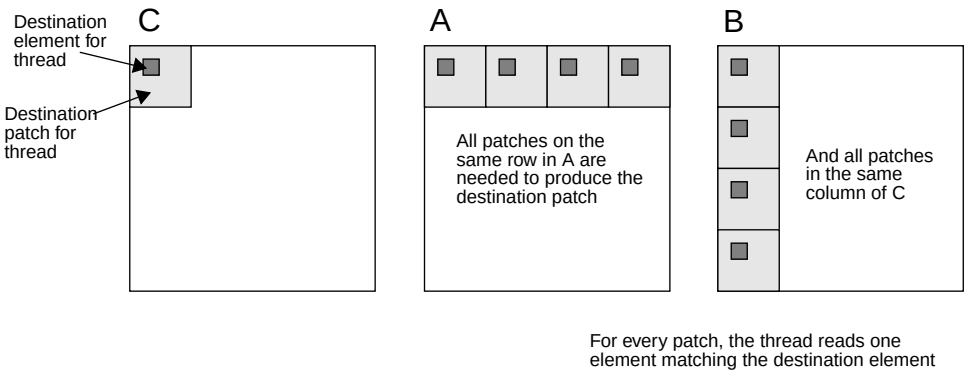


FIGURE 15. GPU implementation, using a sequence of input patches

For each patch that is processed, the whole (shaded) area is loaded into shared memory, and then we calculate the contribution to the dot product that should end up in the output item for the thread in question. See Figure 16.

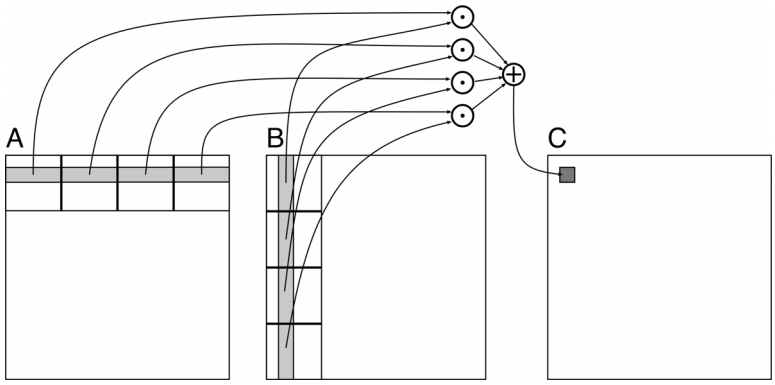


FIGURE 16. Output is accumulated as the dot product of sub-rows and sub-columns of parts of the matrices

The optimized GPU version follows here.

```
__global__ void MatrixMultOptimized( float* A, float* B, float* C, int
theSize)
{
   int i, j, k, b, ii, jj;

// Global index for thread
   i = blockIdx.x * blockDim.x + threadIdx.x;
   j = blockIdx.y * blockDim.y + threadIdx.y;

   float sum = 0.0;
   // for all source patches
   for (b = 0; b < gridDim.x; b++)
   {
      __shared__ float As[BLOCKSIZE*BLOCKSIZE];
      __shared__ float Bs[BLOCKSIZE*BLOCKSIZE];

      // Index locked to patch
      ii = b * blockDim.x + threadIdx.x;
```

```
      jj = b * blockDim.y + threadIdx.y;

      As[threadIdx.y*blockDim.x + threadIdx.x] = A[ii*theSize + j];
      Bs[threadIdx.y*blockDim.x + threadIdx.x] = B[i*theSize + jj];

      __syncthreads(); // Synchronize to make sure all data is loaded

      // Loop, perform computations in patch
      for (k = 0; k < blockDim.x; ++k)
         sum += As[threadIdx.y*blockDim.x + k]
         * Bs[k*blockDim.x + threadIdx.x];

      __syncthreads(); // Synch so nobody starts next pass prematurely
   }

   C[i*theSize + j] = sum;
}
```

We can find these parts of the code:

- Allocate shared memory
- Calculate indices both local to the patch and globally
- Copy one element to shared memory
- Loop over row/column in patch, compute, accumulate result for one element
- Write result to global memory

The result is something like 25-30 times faster on my computer! So what did I do? First, I must use a decent number of threads and blocks, so the computation is reasonably balanced. I use shared memory for temporary storage to reduce global memory access. Note that all threads read *one* item, but use many! Finally, I synchronize after any stage where the threads depend on each other.

There is, however, one weakness in my measure: I compare with single-thread CPU. For a fair comparison, I should split the CPU version to multiple cores.

7.4 Modified computing model

I suggested a simple computing model in chapter 6. Now we can expand that a little bit, as follows.

- Upload data to global GPU memory
- For a number of parts, do:
- Upload partial data to shared memory
- Process partial data
- Write partial data to global memory
- Download result to host

This gives us a better view on the structure of a typical CUDA/GPU computing solution.

8. More language features

In this chapter, I will discuss some features and details, mainly in CUDA.

8.1 Synchronization

As soon as you do something where one part of a computation depends on a result from another thread, you must synchronize!

In a thread, local to a group, this is done with this call:

```
__syncthreads()
```

When using shared memory, this will typically work like this:

- Read to shared memory
- __syncthreads()
- Process shared memory
- __synchthreads()
- Write result to global memory

This seems simple, even trivial. Do we need so synchronize when everybody are doing the same thing anyway? However, since not all threads are running at the same time, we need to wait sometimes. Synchronization simply means “wait until everybody are done with this part”.

However, this is local to a block only. We return to global synchronization in the next section.

Although we have a SIMD/SIMT machine, note that deadlocks can still occur! CUDA allows loops, so if you set up a locking mechanism with semaphores, you can get stuck in a loop that never unlocks. If you try to set up semaphores to handle dependencies between different blocks, I can almost guarantee that you get deadlocks. More about that below.

8.1.1 Global synchronization

There is a big limitation in synchronization with __synchthreads(): It can only be done within a block! No synchronization is possible between blocks!

Why is this a necessary limitation? That is because all blocks are not active at the same time! We may have more blocks than SMs, so blocks are queued until an SM is free! So don't even think about having one block waiting until another block has finished. If that other block waits to start executing, you have a solid deadlock on your hands.

But what if my algorithm *must* synchronize globally? There are many such cases, where a part of an algorithm depends on the result of totally different parts of the computation.

The answer is simple: You run one iteration in one kernel run, and then you finish that kernel run. Once all blocks have finished, you can launch a new iteration.

And you need to wait until the previous kernel run finishes, otherwise you may have several runs overlapping, getting conflicts over accessing the same data.

Thus, there are three synchronization calls, one for the kernel code and two for the host:

```
__syncthreads()
cudaDeviceSynchronize()
cudaStreamSynchronize()
```

__syncthreads() is used inside a kernel, and affects the current block. Stop thread until all threads *in the block* reach the location!

cudaDeviceSynchronize() is used from the host. Wait until all current kernels finish.

cudaStreamSynchronize() waits until all kernels in a stream finish. We have not talked about streams yet, though.

8.2 Error checking

So far, we have basically expected our programs to work flawlessly. As we all know, large programs without bugs is a dream. We must check for errors.

Most CUDA function calls from the host return error codes. However, kernel launches do not, so we check for errors there with separate calls. The main calls are simple: cudaGetLastError(), which gets the latest error and removes it from the list of errors, and cudaPeekLastError(), which looks at the latest error without removing it,

But note that many errors do not occur at the time of the function call. These are asynchronous errors. They happen after the call is made. For those, you call cudaDeviceSynchronize() and check its returned error code.

8.3 Query devices

You can't trust all devices to have the same, or even similar, data. The number of SMs vary a lot, and, more importantly, the amount of shared memory, registers and maximum number of threads in a block vary between different GPU generations. You may design for the current boards and a few generations back (all the way back to G80 if you are ambitious), but the boards arriving in the future may have totally different data. You can not assume that everything grows.

What your program can do is to query CUDA for a list of features. This is made using cudaGetDeviceProperties().

Here follows two examples. Both are a bit small, few SMs, because both are portable chips. Here is the query result from my old laptop (9400M):

```
---- Information for GeForce 9400M ----
Compute capability:  1.1
Total global memory (VRAM):  259712 kB
Total constant Mem:  64 kB
Number of Streaming Multiprocessors (SM): 2
Shared mem per SM:  16 kB
Registers per SM:  8192
Threads in warp:  32
Max threads per block:  512
Max thread dimensions:  (512, 512, 64)
Max grid dimensions:  (65535, 65535, 1)
```

My newer laptop has a more modern GPU, not the absolutely latest but at least a pretty capable Kepler (GT 650M). Here is the query result for that:

```
---- Information for GeForce GT 650M ----
Compute capability:  3.0
Total global memory/VRAM:  523968 kB
Total constant Mem:  64 kB
Number of Streaming Multiprocessors (SM):  2
Shared mem per SM:  48 kB
Registers per SM:  65536
Threads in warp:  32
Max threads per block:  1024
Max thread dimensions:  (1024, 1024, 64)
Max grid dimensions:  (2147483647, 65535, 65535)
```

That is quite a bit of information. So, what is important to us?

They have different *compute capability*. This basically means what generation of GPU architecture we have. The question for this is whether your program use features from a later chip, so this tells whether your program has any chance at all to work on this chip.

The amount of shared memory is maybe the most important piece of information. Here you can see whether your assumptions of available shared memory holds, if your program will fit in the memory.

Then we have the maximum number of threads and the maximum dimensions. Again, you can check whether we fit in the hardware.

We have the number of threads in warp. This rarely change but some day it might. You will often ignore this, unless you use warp-based tricks to avoid synchronizations.

The number of SMs basically gives you the lower bound for the number of blocks you should use. If you try fewer than this, part of the GPU will idle.

8.4 Compute capability

Let us look more closely at the *compute capability* (CC). This is essentially a CUDA/ architecture version number. Here is a list of compute capabilities:

- 1.0: Original release.
- 1.1: Mapped memory, atomic operations.
- 1.3: Double support.
- 2.0: Fermi.
- 3.0: Kepler.
- 5.0: Maxwell.
- 6.0: Pascal.

For full details see the CUDA C programming guide [3], the compute capability appendix late in the document. Here, let me cite some details of interest.

Compute capabilities 1.0 to 1.3 are the G80/Tesla architecture. It is now being phased out, so starting with CUDA 7.0 it is no longer supported.

Atomic functions were enhanced with many new features in CC 2.0. See chapter 9.3.

Unified (managed) memory was introduced with CC 3.0. See chapter 9.6.

Half-precision floating-point operations were introduced with CC 5.3. This is a somewhat surprising move, since half-precision floating point has been in the GPUs for a long time.

Maximum dimension of blocks in x-dimension was 64k up to CC 2.x, then it was increased to 2^{31}-1.

So, should you care about Compute capability? While learning CUDA, I say not so much. Stick to the basics, it works everywhere. But if you write professional CUDA code, that is a different situation. You need to optimize more, and then you may get dependent on new features.

8.5 Timing and profiling

Since GPU computing is all about performance, we must be able to measure the effect of what we do. There are a few ways to do this.

Concerning timing, I could say that there are two ways of timing GPU programs. You can use a CPU timer, or built in events/timing functionality.

8.5.1 CPU timers

An easy timing method that I find pretty reliable is to use the built-in timers of the OS. Although it can not deliver the same precision as calls built into the computation framework, it makes it possible to use the same measurement for multiple platforms. Under UNIX-like systems I use the function gettimeofday() and calculate the time from that. This is quite easy. Here is a function from my timing code “milli.c”:

```
double GetSeconds()
{
   struct timeval tv;

   gettimeofday(&tv, NULL);
   if (!hasStart)
   {
      hasStart = 1;
      timeStart = tv;
   }
   return (double)(tv.tv_usec - timeStart.tv_usec) / 1000000.0 + (dou-
ble)(tv.tv_sec - timeStart.tv_sec);
}
```

The important part is the calculation of time, in this case the difference from one time to another. There is another call, ResetMilli, which simply stores the current time in the timeStart variable.

To get correct timing with this call, we must synchronize properly! Synchronizing after computation is necessary. If you had other computations running before the one you wish to time, you may also need to synchronize before measuring. Thus, the usage may look like this:

```
cudaThreadSynchronize();
ResetMilli();
my_kernel<<<dimGrid, dimBlock>>>(arguments);
cudaDeviceSynchronize();
t = GetSeconds();
```

The first synchronization may be skipped if no other computation is running, or if you synchronized after the last.

8.5.2 CUDA Events

CUDA events is maybe not exactly what you expect. They are for two things: Knowing that a task has completed, and timing computations. The advantage over CPU timing is

quite clear; CUDA events work inside the CUDA framework, closer to the computation, and is therefore likely to be more exact.

Since CUDA runs asynchronously, you need to synchronize. When using CPU, you should synchronize both at start and finish, while for CUDA events, you only need that at the end.

The CUDA Events API contains the following calls:

cudaEventCreate(): initialize an event variable

cudaEventRecord(): place a marker in the queue

cudaEventSynchronize(): wait until all markers have received values

cudaEventElapsedTime(): get the time difference between two events

cudaEventDestroy(): Dispose an event variable.

That is about it. The usage looks something like this:

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);
my_kernel<<<dimGrid, dimBlock>>>(arguments);
cudaEventRecord(stop);
cudaDeviceSynchronize();
cudaEventSynchronize(stop);
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
```

The API is slightly more complex than the CPU timing, but if timing CUDA is all you want to do (that is, not comparing to other platforms), CUDA events are recommended.

8.6 CUDA streams and overlapping data transfers

CUDA processes commands in *streams*. Our basic examples only use the default stream, which makes the stream concept invisible to us. However, we can create additional streams. An important reason to do this is to optimize data transfers. This section is based on material by Harris [26].

Each stream has its own CUDA Events, so we can use them to determine when a computation in a specific stream has finished.

We must here introduce a new memory mode, even though we are not in a memory access chapter: It is called *pinned memory* or *page-locked memory.* We shall see how it can be used to boost performance for memory transfers.

So far we have used malloc() and cudaMalloc(). Pinned memory is allocated with a new call: cudaHostAlloc(). This allocates a page-locked memory. This means that it has a fixed physical location! This sound practical, but page-locked memory is a limited resource.

For non-pinned memory, CUDA copies it internally to page-locked memory, then DMA to GPU. Transfer time goes up! By using pinned memory from the start, we can optimize this a bit.

However, the most interesting application is probably for *overlapping computations*. Then, it is no longer just a slight speedup of data transfer, but may provide a significant boost.

We need a new data copying call: cudaMemCpyAsynch(). This can copy locked memory asynchronously.

8.6.1 Multiple streams

CUDA commands are placed in a queue, a stream. These are the same queues as you can post CUDA events to. We usually only use the default CUDA stream.

Multiple CUDA streams can be used to overlap work - especially computing and data transfers!

With single stream computations, the kernel can not run until the data is transferred. See Figure 17. For this example, 2/3 data transfer, 1/3 computation

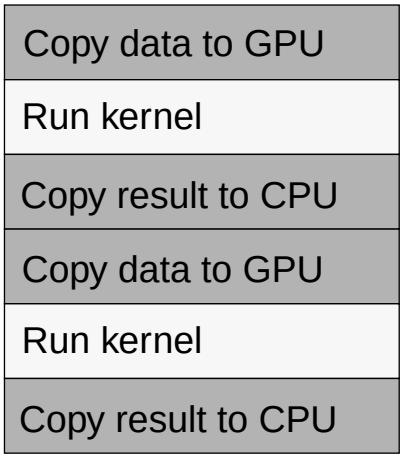


FIGURE 17. Single stream computation

With more than one stream active, we can gain flexibility. While one stream runs a kernel, the other stream performs data copying. See Figure 18. The amount of time free for computing goes up. In the figures, the dual stream example has the kernels running 1/2 of the time instead of 1/3, a most respectable speedup.

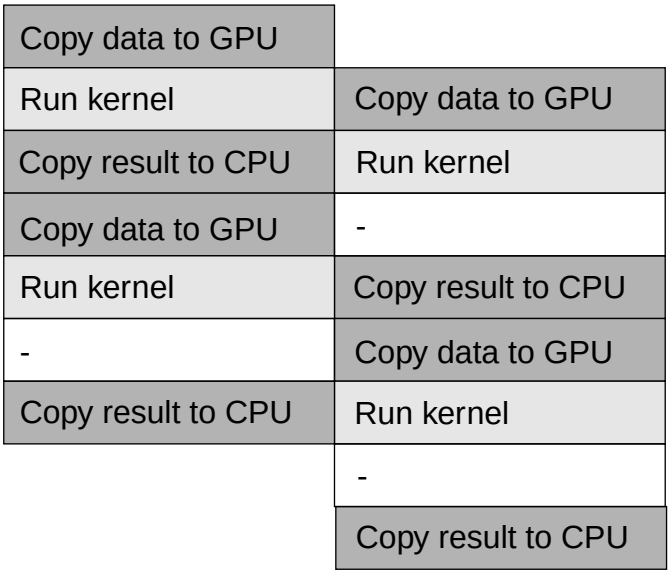


FIGURE 18. Dual stream computation

However, not all devices support this. Asynchronous data copying as well as concurrent execution is not guaranteed. We should make a device query to make sure that it does, and of not, switch to the simpler, less efficient solutions. Try the following queries:

CU_DEVICE_ATTRIBUTE_ASYNCH_ENGINE_COUNT: Can we copy memory asynch?

CU_DEVICE_ATTRIBUTE_CONCURRENT_KERNELS: Can we run multiple kernels?

9. Memory access part 2

Earlier, we have discussed primary memory and shared memory. However, there are a few more concepts, memory types and memory access types left to cover. I will start with the most vital concept, coalescing, and continue with faster access of shared memory as well as texture and constant memory.

9.1 Coalescing

Coalescing is a technique for optimizing the caching of the GPU when doing global memory accesses. I would claim that the manual ([3], section G.3.2) is quite confusing and incomplete on this matter so I will try to sort it out to something that we can somewhat easier apply on our code.

You are advised always to access global memory in order since nearby accesses will help caching. This should be made in order of thread numbers. Thus, note that the “access in order” does *not* refer to consecutive accesses by one thread, but simultaneous accesses by neighbor threads!

“In order” does not mean that it has to be strictly in order, but nearby. More precisely, memory accesses by threads in the same warp will be organized to fewer memory accesses. This is made in chunks of 128 bytes, the size of a cache line.

Maybe the best example of coalesced versus non-coalesced access is the access of a 2D array (e.g. image or matrix). When reading a row at a time, your accesses will typically be coalesced, but if you work column-wise, you will get a jump in memory of a whole row at a time. Also, note that if you are reading an RGBA image one channel at a time, so each thread first reads R, then G, then B and finally A, with four separate accesses, this also will not help coalescing.

As noted by Harris[15], making jumps in memory, strided memory access, will rapidly reduce memory bandwidth. Harris reports top performance when accessing items that are immediately following each other, with performance rapidly dropping until coalescing effects disappear completely with a stride of around 12 to 16.

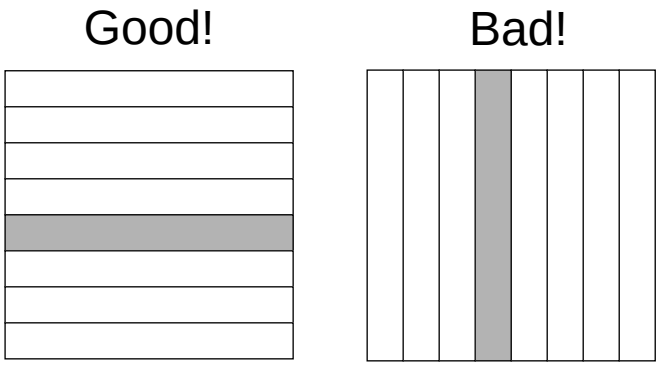


FIGURE 19. Coalesced memory access for matrices.

So line up your memory access. Pure memory transfers can be 10x faster by taking advantage of memory coalescing. I end this discussion with a citation from the documentation:

“Perhaps the single most important performance consideration... is coalescing of global memory accesses.” (CUDA C Best Practices Guide 2018) [16]

9.1.1 Matrix transpose example

A good, and common, example of the importance of coalescing is the problem of transposing a large matrix. It is a very simple problem, just flip an array of data over the diagonal. There are only memory accesses, no computations at all. A naive implementation would look like this:

```
__global__ void transpose_naive(float *odata, float* idata, int width,
int height)
{
   unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
   unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;

   if (xIndex < width && yIndex < height)
   {
       unsigned int index_in  = xIndex + width * yIndex;
       unsigned int index_out = yIndex + height * xIndex;
       odata[index_out] = idata[index_in];
   }
}
```

How can this be bad? The problem is the access pattern, it is not coalesced. It is reading row by row, that is coalesced, but it is writing column by column, as in Figure 20.

The trick to get this right is, again, to use shared memory. We read coalesced from shared memory, write to shared memory in any order. (This is not strictly true, see chapter 9.2, but for many cases it will work out fine.) Then we read from shared memory in an order that is transposed to when we wrote to it, and write coalesced to global memory.

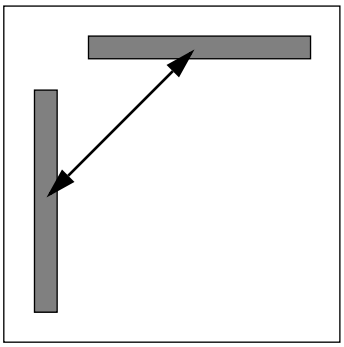


FIGURE 20. Transposing means swapping rows for columns, causing non-coalesced access

Furthermore, for large matrices we must split the matrix into patches so it fits in shared memory. This gives us a three-step solution as in Figure 21.

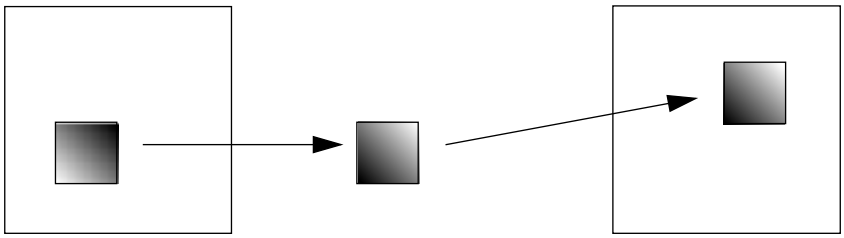


FIGURE 21. Faster transpose by temporarily going through shared memory

Better CUDA matrix transpose kernel

```
__global__ void transpose(float *odata, float *idata, int width, int
height)
{
   __shared__ float block[BLOCK_DIM][BLOCK_DIM+1];

   // read the matrix tile into shared memory
   unsigned int xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
   unsigned int yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;
   if((xIndex < width) && (yIndex < height))
   {
      unsigned int index_in = yIndex * width + xIndex;
      block[threadIdx.y][threadIdx.x] = idata[index_in];
   }

   __syncthreads();

   // write the transposed matrix tile to global memory
   xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
   yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;
   if((xIndex < height) && (yIndex < width))
   {
      unsigned int index_out = yIndex * height + xIndex;
      odata[index_out] = block[threadIdx.x][threadIdx.y];
   }
}
```

Coalescing rules of thumb

- The data block should start on a multiple of 64
- It should be accessed in order (by thread number) or close to it
- It is allowed to have threads skipping their item but if you skip a lot of data, like every other item, you lose bandwidth
- Data should be in blocks of 4, 8 or 16 bytes

9.2 Optimizing shared memory

Shared memory is fast, but even there we can optimize the access. The memory is split into multiple memory banks, 32 ones for Compute Capability 2 to recent (6).

Shared memory access is fastest if you access different banks with each thread. This will often happen effortlessly, but you should be aware of the problem when optimizing.

The memory banks are interleaved in 32 bit chunks. With Kepler (Compute Capability 3) it was configureable, but this feature did not make a very big difference so it went away later. Thus, if you are addressing in 32-bit steps, you will get the best performance, but the most important thing is not to have a stride of 128 bytes (32 bits = 4 bytes, 32 banks), which would make all accesses go to the same memory bank, linearizing the accesses.

This called a *bank conflict*. The number of threads that access the same memory bank simultaneously is called the *degree* of bank conflict, which is how far the memory accesses will me linearized.

Some algorithms, like FFT, are very likely to have bank conflicts if they are implemented in a straight forward manner. Bank conflicts are avoided by introducing an offset, a *padding*, that changes the memory access pattern. This will add some arithmetic operations to the memory access, but the gain in memory access performance will easily be bigger.

This means that shared memory, like global memory, benefits from accessing data in order of thread numbers as in Figure 22, but for a different reason.

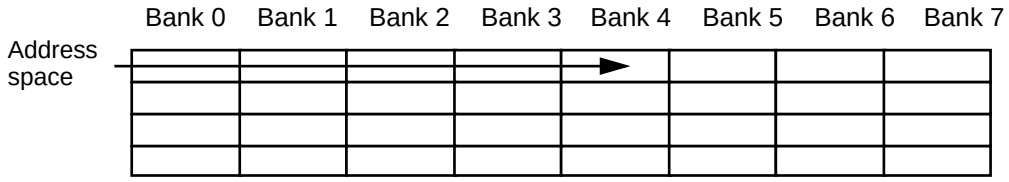


FIGURE 22. Memory banks in shared memory

9.3 Atomic functions

Atomic functions, or *atomics*, are operations that are guaranteed to be racing-free, so any memory accesses caused by it can not be intercepted by some other operation by another

thread. Typically they do a read-modify-write as a single operation. This is very useful for guaranteeing a correct result in some parallel algorithms.

GPUs support atomics. Here follows a fairly complete list of available functions:

atomicAdd(), atomicSub(), atomicExch(), atomicMin(), atomicMax(), atomicInc(), atomicDec(), atomicCAS(), atomicAnd(), atomicOr(), atomicXor()

Already a G80, the very first CUDA capable hardware, had atomics, but some of the functions come on 64-bit versions that are only supported on later architectures (Compute Capabililty3.5 and up).

Although some algorithms are very easy to rewrite using atomics, they should not be overused. For example, the rank sort algorithm (chapter 17.2) is very easy to write using atomics, but since that serializes the operations, makes all operations queue up, waiting for each other, performance will be poor. Thus, atomics can perform well when the number of conflicts is expected to be low, but not when all threads are fighting for the same memory.

Let us take two examples of usage of atomic functions. First one that works well: Histograms. This is simple method for gathering statistics about a set of data. Much data in, little out. It is common in image processing. A sequential implementation looks like this:

```
for all elements i in a[]
   h[a[i]] +! 1
```

given an input array a and an output h, the histogram. For example, the histogram of the Lenna test image is shown in Figure 23.

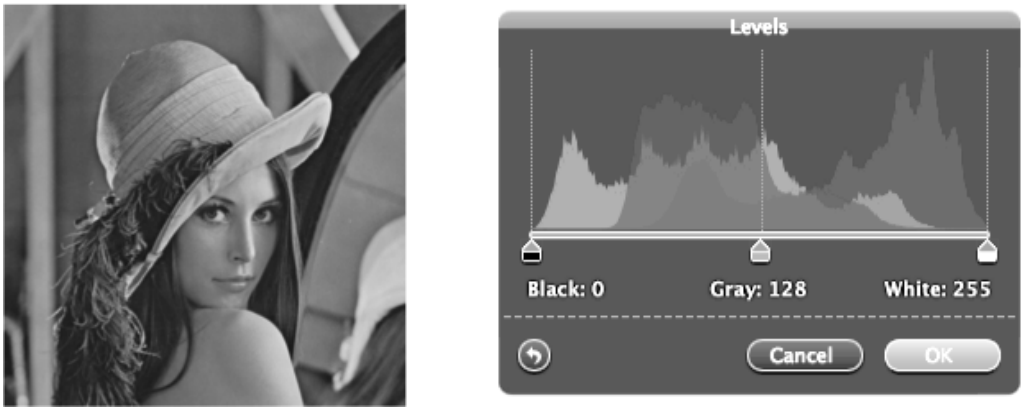


FIGURE 23. The Lenna test image with histogram

If you try to parallellize this operation, multiple threads will write simultaneously at the same item, you will get *racing*. Non-atomic operations will read h[a[i]], add 1, and write back. See Figure 24.

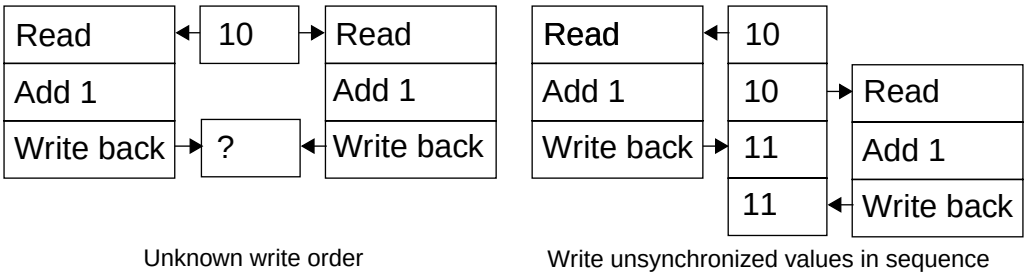


FIGURE 24. Memory access conflicts in the histogram example.

This is quite conveniently solved with atomics. They can read, modify and write in one operation which is then guaranteed not to be subject to racing

This is a pretty good solution for histograms, since any non-trivial data set will have varying values and therefore not cause conflicts very often. Thus, the atomics are not likely to linearize accesses very often, and will guarantee a correct result for a low cost.

However, there are also algorithms where atomics will cost more than the gain. Our next example is very bad for atomics:

```
for all elements i in a[]
   maxValue = max(maxValue, a[i])
```

It is a very simple algorithm, and it will work with atomics. However, is it fast in its parallel version? No, it will be slower than a sequential implementation! The problem is that all threads write to the same memory element. Thus, we should not use atomics for this problem. Solution: Use reduction instead! (See chapter 15.)

To conclude, atomic operations simplify some operations, but it serializes conflicting operations, so it can hurt performance! Use them wisely.

9.4 Constant memory

Constant memory isn't as trivial and boring as it sounds. The big point with it is somewhat similar to coalescing; it enables broadcasting over several threads. However, while coalescing detects nearby accesses to make fewer global memory reads, constant memory optimizes the case where many threads read the *same* data.

As the name says, it is read-only, that is for the kernels. More specifically, it is data that *does not change during a kernel execution*. You mark it with the __constant__ modifier. You use for input data, obviously, so it is writable from the CPU.

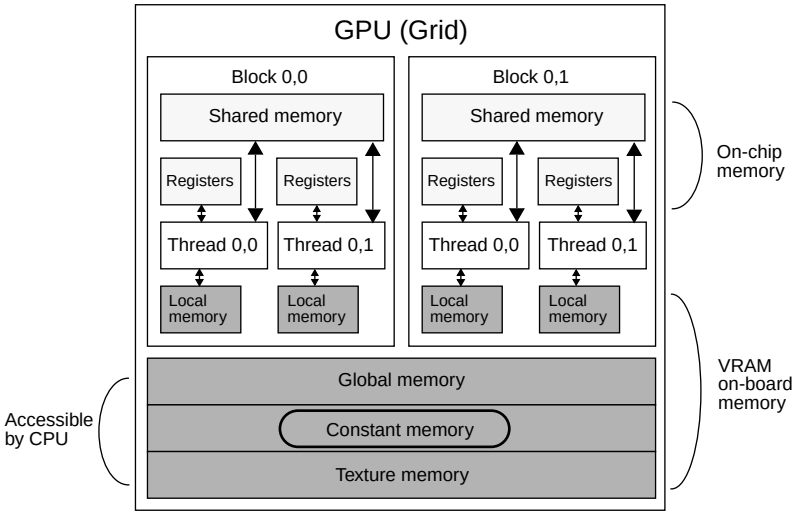


FIGURE 25. Memory overview

As we can see in Figure 25 (variant of Figure 12), constant memory is “global” in the sense that it doesn’t belong to any block. Its size is limited; we have seen numbers like 64k, plus an 8k cache per SM.

The biggest benefit of constant memory is the performance, the ability to broadcast data when several threads read the same data. This is limited to a *half warp*, but one access rather than 16 still matters.

Another advantage, which, however, has lost some importance over time, is the fact that no cudaMemcpy needed for constant memory! You just declare it __constant__, write from the CPU and use it from the kernel. This was a great advantage before managed memory arrived. With GPUs/CUDA installations that support managed memory, we have one more path that has this advantage. See chapter 9.6.

So, we get faster access when all threads are reading the same data *at the same time*. One read can be broadcast to all *nearby* threads, same half-warp, that is 16 threads. But it does not help if the threads are reading different data!

9.4.1 Ray-caster example

Let us look at a case where constant memory matters: a ray-caster! Ray-casting is basically the first step of a ray tracer. (See Volume 1.) It does not support reflections and refractions, but it casts ray from the camera into the scene. This is a less impressing way to render an image, but more suited for a demo. This simple ray-caster will inspect all parts of the scene for every ray. This is not a good way to write raycaster, and even less so a ray-tracer, but it creates a situation where all threads read the same data.

The example below is a simplified raycaster, with an arbitrary number of spheres (I use 100) and one checkered plane as floor.

This demo was inspired by a demo in “CUDA by example” [7] but is rewritten from scratch to allow additional features, like the checkered plane and real-time camera movement.

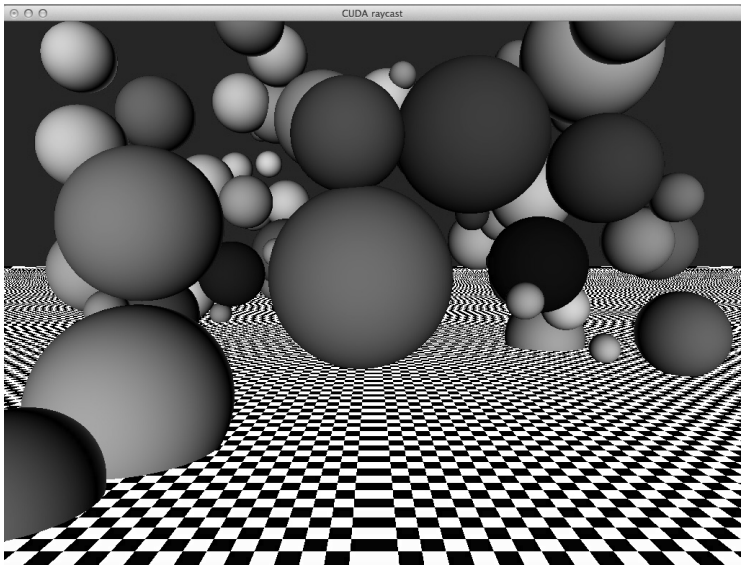


FIGURE 26. Raycaster demo

I can not justify printing the entire code here, but it is/will be available at computer-graphics.se. I am skipping over the vec3 struct and utility functions like dot products.

In the raycaster, every thread renders *one* pixel. Every thread loops through all spheres, and finds the closest with intersection.

The scene is described by an array t of spheres defined as follows:

```
#define THINGCOUNT 100
typedef struct Thing // Spheres
{
   float x, y, z, radius;
   float r, g, b;
} Thing;

Thing t[THINGCOUNT] =
{
   // x, y, z,  radius,  r, g, b
{0.78,-6.15,-16.86,1.14,0.60,0.86,0.54},
{5.56,-5.24,-13.47,0.95,0.38,0.44,0.30},
...
{-7.29,-7.00,-19.30,0.79,0.08,0.47,0.52},
{6.12,-3.88,-11.14,0.88,0.41,0.96,0.17},
};
```

For the ray-casting, the function for calculating the intersection with a sphere is vital. This is straight out of [1].

```
__device__ float intersectSphere(vec3 p, vec3 v, vec3 c, float radius)
{
```

```
   vec3 a = c - p;
   float av = dot(a, v);
   float arg = sqrt(av*av - dot(a, a) + radius * radius);
   if (arg > 0)
      return av - arg;
   else
      return -1;
}
```

The function render() is a straight-forward raycaster. Since we only have one plane, it is hard-coded. Note that the array t appears here under the name dev_things.

```
__device__ vec3 render( Thing *dev_things, int x, int y, float gx, float
gy )
{
    const float scale = 0.7;
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);

   vec3 ray = normalize(vec3(jx, jy, -1));
   float m;
   float bestm;
   vec3 color = vec3(0,0,0.5);
   vec3 p = vec3 (gx, gy ,0);
// Intersect spheres
   bestm = 10000;
   for (int i = 0; i < THINGCOUNT; i++)
   {
      vec3 c = vec3(dev_things[i].x, dev_things[i].y, dev_things[i].z);
      m = intersectSphere(p, ray, c, dev_things[i].radius);
      if (m > 0 && m < bestm)
      {
         bestm = m;
         vec3 intersection = p + ray * m;
         vec3 n = normalize(intersection - c);
         color = vec3(dev_things[i].r, dev_things[i].g, dev_things[i].b)
* abs(n.z);
      }
   }
// Checkered plane
   {
      vec3 n = vec3(0,1,0);
      float d = -1;
      m = - (dot(n, p) + d)/dot(n, ray);
      if (m < bestm && m > 0)
      {
         vec3 intersection = p + ray * m;
         // Find pattern
         int dx = (int)(intersection.x * 5);
         int dz = (int)(intersection.z * 5);
         int bw = abs(dx + dz + 1000) % 2;
         color = vec3(bw, bw, bw);
      }
   }

    return color;
}
```

The kernel entry point, kernel(), just calculates global thread numbers and passes that as well as dev_things to render(). (We also pass gx and gy, which refer to the mouse position that controls the camera.)

```
__global__ void kernel( unsigned char *ptr, Thing *dev_things, float gx,
float gy )
{
    // map from blockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * gridDim.x * blockDim.x;

    // Use gx, gy for light source?

    // now calculate the value at that position
    vec3 color = render( dev_things, x, y, gx, gy );
    ptr[offset*4 + 0] = 255 * color.x;
    ptr[offset*4 + 1] = 255 * color.y;
    ptr[offset*4 + 2] = 255 * color.z;
    ptr[offset*4 + 3] = 255;
}
```

We skip most of the main program, but note that a cudaMalloc() and cudaMemCpy is used to pass the array t to the GPU.

```
   err = cudaMalloc( (void**)&dev_things, THINGCOUNT*sizeof(Thing) );
   cudaMemcpy( dev_things, t, THINGCOUNT*sizeof(Thing), cudaMemcpyHost-
ToDevice );

   kernel<<<grids,threads>>>( dev_bitmap, dev_things, gX, gY );
```

The rest is similar to the Julia demo.

On my 650M, this runs in around 38 ms for rendering a frame.

This was the straight forward version with the array t in global memory. Note that the array is read in the same order by all threads. This is important and is why constant memory can help. (Shared memory could also help but that is another matter.)

With constant memory, the array t is now declared __constant__:

```
__constant__ Thing t[THINGCOUNT] =
{
   // x, y, z,  radius,  r, g, b
{0.78,-6.15,-16.86,1.14,0.60,0.86,0.54},
{5.56,-5.24,-13.47,0.95,0.38,0.44,0.30},
...
{6.12,-3.88,-11.14,0.88,0.41,0.96,0.17},
};
```

In the function render(), we no longer have dev_things but refer directly to t:

```
// Intersect spheres
   bestm = 10000;
   for (int i = 0; i < THINGCOUNT; i++)
   {
      vec3 c = vec3(t[i].x, t[i].y, t[i].z);
```

```
        m = intersectSphere(p, ray, c, t[i].radius);
        if (m > 0 && m < bestm)
        {
           bestm = m;
           vec3 intersection = p + ray * m;
           vec3 n = normalize(intersection - c);
           color = vec3(t[i].r, t[i].g, t[i].b) * abs(n.z);
        }
    }
```

In the main program, t is no longer copies to dev_things.

```
kernel<<<grids,threads>>>( dev_bitmap, gX, gY );
```

This version produces the same results, but only takes 30 ms on my 650M. Thus, the performance improved, for something that simplified the code!

We conclude that constant memory gives relatively fast memory access, but only for the case when all threads (or groups of threads) read the same memory s*imultaneously.* It is not something we use for everything.

9.5 Texture memory/ Texture units

Texture memory is a kind of memory/memory access that clearly shows the graphics heritage. It is also relatively complicated to use. We can see it among the global areas in Figure 25. The texture memory itself is nothing but VRAM, so what is the point with it? The answer is that it provides a few extra features that may help you to optimize your computations. These features are provided for graphics but by making texture memory available to GPU computing platforms, we can use it for other purposes.

Texture memory is read-only, although writable using “surface objects”. The memory has its own cache, which can make it fast if the data access patterns are good.

More importantly, the texture access goes though texture units that provide texture filtering, that is linear interpolation between data items, and automatic edge handling.

You can expect it to be especially good for handling 4 floats at a time (float4), since that is what is used for a pixel. Still, CUDA hides this and we can see the texture as an array of floats.

9.5.1 Texture memory for graphics

In graphics, texture data is, as the name says, mostly for rendering textures, but even in graphics it is often used for other purposes like bump mapping, gloss mapping, noise data and more.

Much of the power of texture memory comes from the features needed by graphics. The border checks (clamp/repeat) enable repeated textures as well as decals, and the interpolation is vital to limit aliasing effects when scaling and rotating images (as in Figure 27).

That means that an access in non-integer coordinates will access 4 neighbor pixels (or more when using mip-mapping, see [1]).

FIGURE 27. Texture access in graphics

These features give texture memory considerable power compared to the normal memory access. Let me stress the most remarkable features:

- Ability to access on non-integer addresses, with hardware interpolation.
- Automatic border checks with options like clamp and repeat.

These features go far beyond images and it would be bad to ignore them.

9.5.2 Using texture memory in CUDA

The CUDA C Programming Guide describes to APIs for using texture memory, the “Texture Object API” and the “Texture Reference API”.

Here follows an example of using texture memory, using the “Texture Object API”. It is based on sample code by NVidia, but stripped down to a minimum. I generate an array of alternating 1’s and 0’s, and zoom that data to 4x more, thereby using the interpolation feature.

As we will see, the texture API in CUDA is not exactly intuitive. I find it easier to use textures through OpenGL.

First, we must have a texture reference:

```
texture<float, 2, cudaReadModeElementType> tex;
```

Here is the kernel. It will scale the input to the output.

```
__global__ void kernel(float* output,
            int inwidth, int inheight, int width, int height)
{
```

```
   unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
   unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;
    output[y * width + x] = tex2D(tex, (float)x*inwidth/width+0.5,
                        (float)y*inheight/height+0.5);
}
```

Here I define the size of the input and output.

```
#define inwidth 16
#define inheight 16
#define insize (inwidth*inheight*sizeof(float))

#define width 64
#define height 64
#define size (width*height*sizeof(float))

float indata[inwidth * inheight];
```

The host code should also include some error checks but they are removed to increase readability.

```
// Host code
int main()
{
   // Input is a sequence of 0 and 1.
   for (int i = 0; i < inwidth * inheight; i++)
      indata[i] = (float)(i & 1);

   // Allocate CUDA array in device memory
   cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc(32, 0, 0,
0, cudaChannelFormatKindFloat);
   cudaArray *cuArray;
   int err = cudaMallocArray(&cuArray, &channelDesc, inwidth, inheight);

   // Copy to device memory some data located at address indata in host
memory
   cudaMemcpyToArray(cuArray, 0, 0, indata, insize, cudaMemcpyHostToDe-
vice);
```

Here we set the texture parameters. Important! See below for alternative settings.

```
    // Set texture parameters
    tex.addressMode[0] = cudaAddressModeWrap;
    tex.addressMode[1] = cudaAddressModeWrap;
    tex.filterMode = cudaFilterModeLinear;
    tex.normalized = false;    // access with unnormalized texture coor-
dinates

   // Bind the array to the texture
    cudaBindTextureToArray(tex, cuArray, channelDesc);

    // Allocate device memory for result
    float *deviceoutput = NULL;
    cudaMalloc((void **) &deviceoutput, size);

   // Host output
    float *hostoutput = (float *) malloc(size);

   // Invoke kernel
```

```
   dim3 dimBlock(16, 16);
   dim3 dimGrid((width  + dimBlock.x - 1) / dimBlock.x,
             (height + dimBlock.y - 1) / dimBlock.y);
    kernel<<<dimGrid, dimBlock>>>(deviceoutput, inwidth, inheight,
width, height);
   cudaThreadSynchronize();

   // Get the ouput data.
   err = cudaMemcpy((void *)hostoutput, (void *)deviceoutput, size,
cudaMemcpyDeviceToHost);
   if (err != 0)
   {
      printf("Copy back failed\n");
   }
   for (int i = 0; i < 16; i++)
      printf("%f\n", hostoutput[i]);

   // Free device memory
   cudaFreeArray(cuArray);
   cudaFree(deviceoutput);
   free(hostoutput);

   return 0;
}
```

This program produces an output like this, which is a small sample of the actual result:

```
--- texobjdemo starts ---
0.000000
0.250000
0.500000
0.750000
1.000000
0.750000
0.500000
0.250000
0.000000
0.250000
0.500000
0.750000
1.000000
0.750000
0.500000
0.250000
--- texobjdemo finished ---
```

Thus, it makes linear interpolation between each pair of 1 and 0.

We see that the example uses cudaMallocArray() and cudaMemcpyToArray to create the texture. For the output, I use the standard buffers.

The texture is set up with the parameters for interpolation and clamp/repeat. We bind to texture unit using cudaBindTextureToArray()

We read from data using tex2D(), with the additional freedom to skip border checks and address between integer positions.

Being used to OpenGL, I am not very happy about the API. The access is simple, just like in OpenGL, but I find the setup to be more complicated.

9.5.3 Clamp and repeat

Texture access needs no boundary checks, and this is supported by the hardware so you get the checks for free! No access violations, you are always inside your buffer! The only question is whether you want the access to clamp to edge or repeat.

Actually, CUDA gives as much as four edge behaviors, *clamp*, *border*, *wrap* and *mirror.*

cudaAddressModeBorder

cudaAddressModeClamp

cudaAddressModeWrap

cudaAddressModeMirror

These constants are written into the texture description record, in the dressmaker field. Note that the field stores three values, one for each coordinate axis.

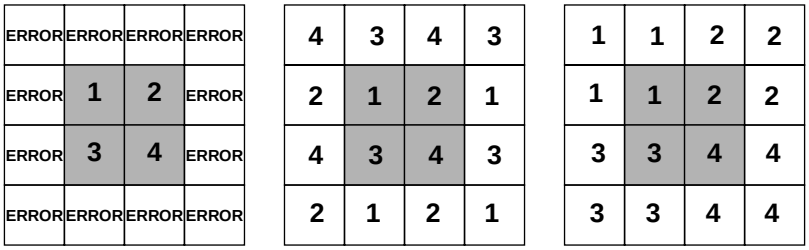


FIGURE 28. Texture memory access protects you from illegal access with repeat (middle) and clamp (right).

9.5.4 Interpolation

Another interesting feature of texture access is interpolation. You are allowed to access textures not only on integer coordinates, but on non-integer coordinates, between the data entries! What happens then is that the data is interpolated with linear interpolation, as much as trilinear interpolation, and this, too, is done in hardware!

This can be useful as computation trick when optimizing. One example when it can be useful is when implementing filters, although only ones with positive weights like the low-pass filters in chapter 18. Other applications include simulation of liquids.

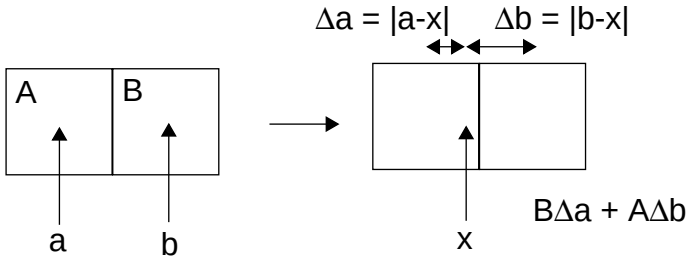


FIGURE 29. Automatic linear interpolation by accessing between pixels

However, you need to check how good interpolation your GPU performs. It is meant for interpolating between texels, a visual effect, and the presence of small errors is not a problem then! Thus, you may find that some GPUs have precision. I have seen one that only interpolated in 10 steps. That reduced its usefulness significantly.

9.6 Managed/unified memory

Managed/unified memory was briefly introduced before, in chapter 5. Let me describe in more details how it works. It does not take much detail though. From the programmer's side, it is purely a simplification.

What managed memory gives us is a shared memory space, where CPU and GPU can work in the same memory without any explicit memory transfers. No more cudaMemcpy calls!

In order to enable this, instead of allocating memory with cudaMalloc() (on the GPU) or malloc() (on the CPU) you use cudaMallocManaged(). You should also declare your pointers __managed__.

For small examples, like Hello World!, this simplifies the trivial into something close to ridiculous. However, for larger, more interesting problems, the difference quickly becomes less important. So, let us revisit Hello World!

```
#include <stdio.h>

const int N = 16;
const int blocksize = 16;

__global__
void hello(char *a, int *b)
{
   a[threadIdx.x] += b[threadIdx.x];
}

__managed__ char a[N] = "Hello \0\0\0\0\0\0";
__managed__ int b[N] = {15, 10, 6, 0, -11, 1, 0,0,0,0,0,0,0,0,0,0};

int main()
{
   printf("%s", a);
   dim3 dimBlock( blocksize, 1 );
```

```
    dim3 dimGrid( 1, 1 );
    hello<<<dimGrid, dimBlock>>>(a, b);
    cudaDeviceSynchronize();
    printf("%s\n", a);
    return EXIT_SUCCESS;
}
```

The small and simple was reduced to just a handful of lines! So, why don’t we use this all the time? Well, we are getting there but at the time of writing this, there are still active platforms where it can not be used.

This simplification is not isolated to CUDA. OpenCL, for example, has a related concept: Zero Copy Memory. Memory is allocated with the modified CL_MEM_ALLOC_HOST_PTR.

So, how about performance? Does it cost to use managed memory? The general answer seems to be a vague “maybe”, because hardware capabilities and implementations may vary. However, it is notable that Intel claims that managed memory will *always* be faster! [25] It seems your mileage can vary but the general message is not to shy way from it.

10. OpenCL

Besides CUDA, the most famous GPU computing platform is probably OpenCL. OpenCL stands for “open compute language”. Like OpenGL, it is an open standard, and *open specification*, which means that anyone can make their own implementation. Also, like OpenGL, it is managed by the Khronos Group.

OpenCL was released 2009. The motivation for it was clearly the lack of open alternatives for GPU computing. CUDA and Direct Compute are both commercial systems limited to the platforms that the vendors choose to support. A driving force behind OpenCL was Apple Computer, but there are many providers, and many supported architectures.

From an idealistic perspective, we could say that the dream is an API for *all* architectures. We can immediately realize that this is a dream, one system can not be optimal for everybody. Still, the list of supported architectures is considerable. In includes, but is not limited to: GPUs (NVidia, AMD, Intel), Intel compatible CPUs (Intel, AMD), ARM, FPGA, CELL, Intel Xeon Phi...

Who decides? Any company making its own OpenCL implementation! It is an open specification, so if you are missing a platform, feel free to support it. But that also means that there is not *one* OpenCL library but several. This also means that some implementations may be lagging behind, not getting updated with new features. Ironically, one such implementation is Apple’s, which is odd since they were a driving force behind the creation of OpenCL!

There is no such thing as a free lunch, the model does not fit all architectures. It is a one size fits all solution, which makes platform dependent optimizations hard to do. But for GPU computing, it fits right in and you can easily see that the GPU platform is its main focus.

10.1 OpenCL for GPU Computing

OpenCL is strikingly similar to CUDA both in architecture and performance. The big difference is the setup, which is a lot more complicated, but once you have a working setup, it is not really a big problem. Computing kernels are quite similar to CUDA. To some

extent, performance is similar, but it is easier for NVidia to be first with new features. For OpenCL, many parties must agree on additions, and then everybody must add support for the new features. This often makes OpenCL lag behind.

10.2 OpenCL vs. CUDA terminology

Unfortunately, the terminology differs substantially between different platforms, and CUDA, being first, is the one that is the most different.

OpenCL	CUDA
compute unit	multiprocessor (SM)
work item	thread
work group	block
local memory	shared memory
private memory	registers

Most notable, OpenCL local memory = CUDA shared memory, while CUDA local memory is a vague concept which causes confusion. However, NVidia's manual is fairly clear on the subject:

"Local memory space resides in device memory, so local memory accesses have the same high latency and low bandwidth as global memory accesses and are subject to the same requirements for memory coalescing..." [3]

Which means that CUDA local memory is a slow memory, used when thread local variables ca not be placed in registers, while OpenCL local memory is fast, the same as CUDA shared memory.

Due to this confusion, I will primarily use the term shared memory since it is unique.

- CUDA local memory = global memory accessible only by one thread (like registers but slower)
- CUDA shared memory = OpenCL local memory = memory local inside the SM, shared within block work group

OpenCL local memory is declared *__local*:

```
__local float a[SOMESIZE];
```

See chapter 17.2 for an OpenCL example using local memory.

10.3 OpenCL memory and thread model

On the subject of computing with OpenCL, let us not waste space and time repeating what I said in chapter 6. Most of it maps straight over, work groups (blocks) map onto SMs, each work group consists of work items (threads). The same rules apply.

One notable difference is the thread and block identification. While CUDA gives you block number and thread number within the block, OpenCL is somewhat more convenient. The built-in calls get_global_id() and get_local_id() returns identification indices on a local or global scale, which means that a part of the thread ID calculation has been done for you. Both calls take an integer as parameter which specifies the dimension that you wish the id for. There is also get_work_dim() for getting the total size, and get_local_size() for block/workgroup size.

10.4 Heterogeneous

Apart from the superficial syntax differences, the more significant differences from CUDA comes from OpenCL being designed for heterogeneous systems, not only heterogeneous as in a GPU controlled from a CPU, but beyond that, for any kind of mix, where the same task can be assigned totally different hardware. Several devices may be active at once and contribute to the same computation.

This will mainly affect the setup, where you can specify which device to launch a task to, and select different OpenCl implementations. This gives us some overhead compared to CUDA, but the reward is flexibility.

10.5 Language

The OpenCL kernel language is based on C99, but there are some differences.

- No function pointers
- No pointers to pointers in function calls
- (=> no multi-dimensional arrays)
- No recursion
- No arrays with dynamical length
- No bitfields

Some features are optional:

- Pointers with length <32 bit
- Writing support for 3D images
- Double and half types
- Atomic functions

So far, I have mainly listed limitations, but there are also some strong points worth mentioning:

- Integrated functions for reading / writing 2D images and
- reading 3D images
- Converting functions incl. explicit rounding and saturation

- math.h, all functions with different precisions
- Vector support (2-, 3- and 4-dimensional)

Available primitive datatypes: Bool, char, int, long, float, size_t, void, plus their unsigned versions.

Mix of OpenCL and OpenGL possible, so OpenCL can share data structures and variables (without copying). More about that in chapter 15.2.

When talking about features, there is one more thing I would like to mention: In early versions, there was no possibility to call a kernel from another kernel, but that feature arrived in OpenCL 2.0.

10.6 Walk through the Hello CL example code

We can identify two major parts of the code, the setup and the computing. For a small example, the setup is pretty big. It follows these steps:

- 1) Get a list of platforms
- 2) Choose a platform
- 3) Get a list of devices
- 4) Choose a device
- 5) Create a context
- 6) Load and compile kernel code

Then we can start working. Here are the computing steps:

- 7) Allocate memory
- 8) Copy data to device
- 9) Run kernel
- 10) Wait for kernel to complete
- 11) Read data from device
- 12) Free resources

1-5: Where to run

```
cl_platform_id platform;
unsigned int no_plat;
err =  clGetPlatformIDs(1,&platform,&no_plat);

// Where to run
err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
if (err != CL_SUCCESS) return -1;

context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
if (!context) return -1;
```

```
commands = clCreateCommandQueue(context, device_id, 0, &err);
if (!commands) return -1;
```

6: Kernel; What to run

```
program = clCreateProgramWithSource(context, 1, (const char **) & Ker-
nelSource, NULL, &err);
if (!program) return -1;

err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
if (err != CL_SUCCESS) return -1;
kernel = clCreateKernel(program, “hello”, &err);
if (!kernel || err != CL_SUCCESS) return -1;
const char *KernelSource = “\n” \
“__kernel void hello(              \n” \
“   __global char* a,           \n” \
“   __global char* b,           \n” \
“   __global char* c,           \n” \
“   const unsigned int count)  \n” \
“{                              \n” \
“   int i = get_global_id(0);  \n” \
“   if(i < count)               \n” \
“       c[i] = a[i] + b[i];   \n” \
“}                              \n” \
“\n”;
```

Most programs load kernels from files. The Hello World example does not, it stores the kernel as an inline string. I would not do that for bigger kernels.

7-8: Get the data in there

```
   // Create space for data and copy a and b to device (note that we
could also use clEnqueueWriteBuffer to upload)
   input = clCreateBuffer(context,  CL_MEM_READ_ONLY |
CL_MEM_USE_HOST_PTR,  sizeof(char) * DATA_SIZE, a, NULL);
   input2 = clCreateBuffer(context,  CL_MEM_READ_ONLY |
CL_MEM_USE_HOST_PTR,  sizeof(char) * DATA_SIZE, b, NULL);
   output = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(char) *
DATA_SIZE, NULL, NULL);
   if (!input || !output) return -1;

   // Send data
   err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);
   err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &input2);
   err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &output);
   err |= clSetKernelArg(kernel, 3, sizeof(unsigned int), &count);
   if (err != CL_SUCCESS) return -1;
```

9-10: Run kernel, wait for completion

```
// Run kernel!
err = clEnqueueNDRangeKernel(commands, kernel, 1, NULL, &global, &local,
0, NULL, NULL);

if (err != CL_SUCCESS) return -1;

clFinish(commands);
```

11-12: Read back data, release

```
// Read result
err = clEnqueueReadBuffer( commands, output, CL_TRUE, 0, sizeof(char) *
count, c, 0, NULL, NULL );
if (err != CL_SUCCESS) return -1;

// Print result
printf("%s\n", c);

// Clean up
clReleaseMemObject(input);
clReleaseMemObject(output);
clReleaseProgram(program);
clReleaseKernel(kernel);
clReleaseCommandQueue(commands);
clReleaseContext(context);
```

10.7 The Julia example in OpenCL

As one more introductory example, let us look at the Julia example again. The code will be highly similar to the CUDA version.

From the CPU part, I will only include the part closest to the kernel call, setting up parameters, transferring data. Note that this is a simplified version that doesn't use any OpenGL bindings.

```
float theReal, theImag;
cl_kernel k;

int computeFractal(unsigned int *data, unsigned int length)
{
  cl_int ciErrNum = CL_SUCCESS;
  size_t localWorkSize, globalWorkSize;
  cl_mem in_data, out_data;

  in_data = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, length * sizeof(unsigned int), data, &ciErrNum);
  out_data = clCreateBuffer(cxGPUContext, CL_MEM_READ_WRITE, length *
sizeof(unsigned int), NULL, &ciErrNum);

    localWorkSize  = 512;
    globalWorkSize = length;

    // set the args values
    int width = dataWidth;
    ciErrNum = clSetKernelArg(k, 0, sizeof(cl_mem), (void *) &out_data);
    ciErrNum |= clSetKernelArg(k, 1, sizeof(cl_uint), (void *) &length);
    ciErrNum |= clSetKernelArg(k, 2, sizeof(cl_uint), (void *) &width);
   ciErrNum |= clSetKernelArg(k, 3, sizeof(cl_float),(void *) &theReal);
   ciErrNum |= clSetKernelArg(k, 4, sizeof(cl_float),(void *) &theImag);

    cl_event event;
    ciErrNum = clEnqueueNDRangeKernel(commandQueue, k, 1, NULL, &global-
WorkSize, &localWorkSize, 0, NULL, &event);

    clWaitForEvents(1, &event); // Synch
    printCLError(ciErrNum);
```

```
   ciErrNum = clEnqueueReadBuffer(commandQueue, out_data, CL_TRUE, 0,
length * sizeof(unsigned int), data, 0, NULL, NULL);
   printCLError(ciErrNum);

   clReleaseMemObject(in_data);
   clReleaseMemObject(out_data);
   return ciErrNum;
}
```

Some error checks are omitted. Apart from this, there is code for setting up OpenCL as well as providing visual output and user interaction. Note that the localWorkSize may need to be tuned for optimal performance.

As expected, the kernel is relatively simple:

```
int julia( int x, int y, float r, float im, int DIM);

int julia( int x, int y, float r, float im, int DIM)
{
   const float scale = 1.5;
   float jx = scale * (float)(DIM/2 - x)/(DIM/2);
   float jy = scale * (float)(DIM/2 - y)/(DIM/2);

   int i = 0;
   for (i=0; i<50; i++)
   {
      float jx2 = jx * jx - jy * jy + r;
      jy = 2*jy * jx + im;
      jx = jx2;

      if (jx*jx + jy*jy > 1000)
         return i;
   }

   return i;
}

__kernel void juliaKernel(__global unsigned int *outdata,
      const unsigned int length,
      const unsigned int width,
      const float r,
      const float im)
{
   unsigned int pos = 0;
   unsigned int i;
   unsigned int val;

   pos = get_global_id(0);

   val = julia(pos % width, pos / width, r, im, width);


   outdata[pos]= val * 20 + (val * 10 << 8) + (val * 5 << 16);
}
```

At the time of writing this, there are major OpenCL implementations are still at OpenCL 1.2 so that is what I use here. Unlike the CUDA case, I don’t use C++ extensions. These

are available in OpenCL since version 2.1. Even though that version was announced in 2015, as of 2018 it still has not propagated to all major platforms so 1.2 is still of interest.

I chose to make the thread numbering one-dimensional which is kind of questionable when you are working with 2-dimensional output. That is why I do % and / calculations for the position sent to the julia function. I also use different scaling factors on each component in order to get simple colorizing, and I didn't bother with overflows that may affect the next channel. You may fix that as a simple exercise.

We will see more OpenCL examples later, e.g. the rank sorting in chapter 17.

10.8 Some more notes on OpenCL

Let me say a few words about the concepts *platform* and *device*. A platform is an OpenCL *implementation*, i.e. a library. Not OpenCL itself, but one specific implementation. Several can co-exist, supporting different devices. A *device* in OpenCL is therefore a chip which a platform supports. (Otherwise, I usually use the word platform for a specific API, like CUDA/OpenCL/Compute shaders.)

Then we have the question of language freedom. OpenCL is both good and bad in this case. From the CPU side, OpenCL is very easy to call from any language! Anything that can call into a C API can call OpenCL. I have used it from other languages. The kernel code, however, is only C-style. It is theoretically possible for a specific implementation may choose to support more, but the code you write for it will lose portability.

Finally, we have the question of performance. Investigations report remarkably small differences, but the differences vary a lot. We have conducted several investigations ourselves, and the difference can be that one platform is 2x faster than the other. In most cases, CUDA is faster, but not all the time. We have seen OpenCL winning, we have seen them going side by side with no significant differences.

It is very hard to compare, due to multiple OpenCL implementations, and we can only compare NVidia implementations if we want to compare to CUDA.

10.9 Synchronization in OpenCL

As expected, synchronization in OpenCL is largely the same as in CUDA. In OpenCL, synchronization is made with *barriers*. Most importantly, kernels can synchronize within a work group like this:

```
barrier(CLK_LOCAL_MEM_FENCE)
```

Like we said above, there is no synchronization between work groups, so these must be made with multiple kernel runs.

See chapter 17.2 for an example that uses OpenCL barriers.

In OpenCL, the barriers are focused on synchronizing memory access. You choose which kind of memory access to synchronize (global, local).

The host (CPU) can synchronize on global level. This is available for:

- tasks (e.g. clEnqueueNDRangeKernel)
- Memory(e.g.clEnqueueReadBuffer)
- events (e.g. clWaitforEvents)

10.10 Queries in OpenCL

OpenCL can provide information with clDeviceInfo(); Among the options are:

```
CL_DEVICE_LOCAL_MEM_SIZE
CL_DEVICE_LOCAL_MEM_TYPE
CL_DEVICE_MAX_COMPUTE_UNITS
CL_DEVICE_WORK_GROUP_SIZE
CL_DEVICE_MAX_WORK_ITEM_SIZES
```

10.11 OpenCL events

OpenCL has events similar to CUDA events. The options and functions include:

CL_PROFILING_COMMMAND_SUBMIT

```
cl_event
clWaitForEvents
clFinish
clGetEventProfilingInfo
```

10.12 Conclusions on OpenCL

Don’t fear the complex setup phase! The rest is similar to CUDA.

Performance tend to be on par with CUDA or almost.

The speciality of OpenCL is heterogeneous systems.

One big problem burdens the platform: Apple and NVidia lagging behind. OpenCL 2.0 has major enhancements, but Apple and NVidia seem not to be interested! But we are hoping for a change.

11. Fragment shaders

This section is partially based on a chapter in Volume 2.

General-purpose computing with fragment shaders is the “classic GPGPU”, the original GPU computing approach. Here, we use graphics shaders, so the graphics heritage is highly visible. We adapt data and computing to fit the graphics pipeline.

This technique was hot until CUDA arrived. Since then, it is overshadowed by the other platforms, CUDA and OpenCL in particular. However, I do not want to ignore this path, for a few reasons that I will discuss. There are a few arguments:

- Highly suited to all problems dealing with images, computer vision, image coding etc.
- Parallellization, “comes natural”, you can’t avoid it and good speedups are likely. I would argue that there are fewer pitfalls.
- Highly optimized (for graphics performance).
- Compatibility is vastly superior!
- Very much easier to install!

This sounds good, but certainly there are some important weaknesses.

- You must map data to image data
- Computing controlled by pixels in output image
- No shared memory access

Out of these, the lack of shared memory access is probably the biggest weakness. However, OpenGL 4 adds much flexibility, moves closer to CUDA and (especially) OpenCL. We now have new features like writable textures, atomics and synchronization. With this added flexibility, fragment shaders have taken some important steps ahead to make them viable for GPU computing again.

We saw the OpenGL pipeline in Figure 3 on page 20. It consists of multiple stages. Out of these, three are programmable, but only one creates easily accessible output data, the fragment processing stage.

The typical OpenGL situation works with complex geometry, many transformations, perspective projection, lighting and material calculations for the surfaces, and many texture accesses for interpolation and supersampling.

Typical GPU Computing with fragment shaders is vastly different. However, it is not all that alien, since it is also used in filtering in graphics. For this case, we render to a *single rectangle* covering the entire image buffer, we use FBOs for effective feedback, floating-point buffers, and ping-ponging, many pass with different shaders

Thus, the GPGPU model can be summarized as follows:

- We have an array of *input data*. This is put in a *texture* (or several).
- We produce an array of *output data*. This arrives in the *frame buffer.*
- This is produced by a *computing kernel*, which is a *fragment shader.* It is invoked by *drawing graphics*.
- The computation is done by one or several *rendering passes*.
- When we need several passes, the output is rerouted/copied to the input data of the next pass, usually using FBOs, framebuffer objects.

I will now continue with an example, that makes a trivial computation in many iterations using FBOs and ping-ponging.

11.1 Input and output

You load your input data from CPU/RAM to GPU/VRAM, usually using glTexImage2D.

If our input data is a one-dimensional array of floating-point values, they will end up distributed over the four channels of the texture if you use glTexImage2D the standard way. It is, however, possible to upload into a monochrome texture.

A texture is allocated like this:

```
   glGenTextures (1, &tex1);
   glBindTexture(GL_TEXTURE_2D,tex1);
// set texture parameters
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
   // define texture with floating point format
```

We use two textures so we can switch between them. We assign one of the two textures initial data, in our case just zeroes.

```
   float* data = (float*)malloc(4*texSize*texSize*sizeof(float));
   float* result = (float*)malloc(4*texSize*texSize*sizeof(float));
   for (int i=0; i<texSize*texSize*4; i++)
      data[i] = 0.0;

   glBindTexture(GL_TEXTURE_2D, tex1);
   glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA32F,
      texSize,texSize,0,GL_RGBA,GL_FLOAT, data);
```

Note that the texture size is texSize*texSize, but since we have four color channels, the total data size is texSize*texSize*4.

Getting the data out afterwards is not harder. It is copied to CPU/RAM from GPU/VRAM using glReadPixels();

```
   glReadBuffer(GL_COLOR_ATTACHMENT0);
   glReadPixels(0, 0, texSize, texSize,GL_RGBA,GL_FLOAT,result);

// print out results
   printf("Data before computation:\n");
   printf("%f\n",data[texSize*texSize*4-1]);
   printf("Data after computation:\n");
   printf("%f\n",result[texSize*texSize*4-1]);
```

11.2 The computation kernel = the shader

The shaders are read and compiled to one or several program objects. A GPGPU application may have several shaders loaded. We only have one in our example.

Activate the desired shader as needed using glUseProgram[1](); Our shaders are very simple. The fragment shader adds a small number to all elements of the data:

```
#version 150
uniform sampler2D texUnit;
out vec4 outColor;
in vec2 texCoord;
void main(void)
{
   vec4 texVal  = texture(texUnit, texCoord);
   outColor = texVal + vec4(0.001, 0.001, 0.001, 0.001);
}
```

Modern OpenGL requires a vertex shader too. All we need is a trivial pass-through shader:

```
#version 150
in vec3 inPosition;
in vec2 inTexCoord;
out vec2 texCoord;
void main(void)
{
   texCoord = inTexCoord;
   gl_Position = vec4(inPosition, 1.0);
}
```

The geometry is, again, a single polygon. I set it up so it matches the viewport (-1 to 1 in all directions) and with texture coordinates that will match every pixel.

```
GLfloat vertices[] = {-1.0f,-1.0f,0.0f,
                      -1.0f,1.0f,0.0f,
                      1.0f,1.0f,0.0f,
                      1.0f,-1.0f,0.0f};
GLfloat texcoord[] = {0.0f, 1.0f,
                  0.0f, 0.0f,
```

1. glUseProgramObjectARB for older SDKs

```
                  1.0f, 0.0f,
                  1.0f, 1.0f};
GLuint indices[] = {0,1,3, 3,1,2};
```

This allows the vertex shader to be a pure pass-through, which is what we want. We load the geometry to the GPU using vertex buffers. In our demo, we hide that in a Model structure provided by our lab code loadobj.c.

```
// Upload geometry to the GPU:
   m = LoadDataToModel(vertices, NULL, texcoord, NULL, indices, 4, 6);
```

For more details on vertex buffers and vertex arrays, see Volume 1.

11.3 Feedback

In Volume 2, I described how shaders can work in several passes, by using the output from one iteration as input to the next. This kind of feedback is ever-present in GPGPU applications. The bandwidth over the bus to the CPU is limited, so the more that can be done before passing back the data to the CPU, the more efficient will the processing be.

This is most efficiently done by rendering to textures, which is done using framebuffer objects. We create multiple textures, use one or more as input and others as output for each stage, and they can switch roles as needed. This technique is called “ping-ponging”. We discussed this for graphics in Volume 2. Let us do it for a general computing perspective.

First of all, we need two FBOs, created like this:

```
   glGenFramebuffers(1, &fbo1); // frame buffer id
   glBindFramebuffer(GL_FRAMEBUFFER, fbo1);
   glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
GL_TEXTURE_2D, tex1, 0);
```

Now we can run multiple iterations like this:

```
   for (int loop = 0; loop < loopCount; loop++)
   {
      // Ping-pong between fbo1 and fbo2
      if ((loop & 1) == 0)
      {
         glBindFramebuffer(GL_FRAMEBUFFER, fbo2);
         glBindTexture(GL_TEXTURE_2D, tex1);
      }
      else
      {
         glBindFramebuffer(GL_FRAMEBUFFER, fbo1);
         glBindTexture(GL_TEXTURE_2D, tex2);
      }
      DrawModel(m, shader, "inPosition", NULL, "inTexCoord");
      glFlush();
   }
```

If all is well, we will get a number out that matches the number of iterations.

11.4 Image filter in fragment shader

The shader can, for example, look as follows. This particular shader comes from a demo that I believe was an introductory example at GPGPU.org. This is a Laplacian filter for detecting high frequencies, shown in Figure 30. Edges will give a high response, but even more so will local maxima and noise.

-1	-1	-1
-1	8	-1
-1	-1	-1

FIGURE 30. A simple 3x3 Laplacian filter for detecting high frequencies

If this filter had been properly normalized, it should divide the result by 8, but that would make the resulting signal too low to view.

```
uniform sampler2D texUnit;
void main(void)
{
   const float offset = 1.0 / 512.0;"
   vec2 texCoord = gl_TexCoord[0].xy;"
   vec4 c  = texture(texUnit, texCoord);"
   vec4 bl = texture(texUnit, texCoord + vec2(-offset, -offset));"
   vec4 l  = texture(texUnit, texCoord + vec2(-offset, 0.0));"
   vec4 tl = texture(texUnit, texCoord + vec2(-offset,  offset));"
   vec4 t  = texture(texUnit, texCoord + vec2(0.0,  offset));"
   vec4 ur = texture(texUnit, texCoord + vec2( offset,  offset));"
   vec4 r  = texture(texUnit, texCoord + vec2( offset, 0.0));"
   vec4 br = texture(texUnit, texCoord + vec2( offset,  offset));"
   vec4 b  = texture(texUnit, texCoord + vec2(0.0, -offset));"
   outColor = -8.0 * (c + -0.125 * (bl + l + tl + t + ur + r + br + b));
}
```

See Figure 31 below for an example of the effect of this filter.

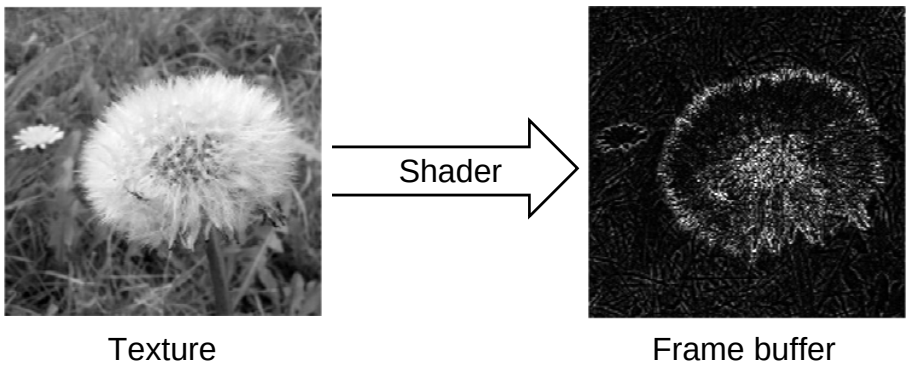


FIGURE 31. High-pass image filter performed in fragment shader.

11.5 Reduction in fragment shaders

Reduction, discussed in chapter 15, is slightly different when using fragment shaders, but only slightly. We can work pretty much the same way. However, a few things are different.

With 4-channel data we should take that into account. GLSL has a max() function that works on a vec4, producing the maximum of each channel. This suggests that we should handle each channel separately until the data is sufficiently small.

The focus on images makes it very tempting to work with 2D images. 3D is also possible for very large data, but harder to handle. It is pretty natural to reduce the data equally along every axis. That makes Figure 36 illustrate a typical reduction for this case. Each kernel run is now one rendering pass, and the number of threads is controlled by the size of the drawn geometry (quad).

12. OpenGL Compute shaders and Vulkan

It is impossible to cover all possible frameworks for GPU computing, but I will here introduce the ones that I judge as the most important ones beside CUDA and OpenCL. The one I consider most important is OpenGL compute shaders, so that is the one that will get most space here, but we should not forget Direct Compute and Vulkan.

12.1 OpenGL Compute shaders

The compute shader concept originally appeared in Microsoft's Direct Compute (chapter 12.8), but the same concept was later added to OpenGL, since OpenGL 4.3. This is not the latest version, but still a bit of "bleeding edge" since 4.3 is not fully universal. We are waiting for some major players to get up to date.

So, why should we consider compute shaders instead of CUDA or OpenCL? I have a few arguments:

- Better integration with OpenGL
- No extra installation!
- Easier to configure than OpenCL
- Not NVidia specific like CUDA
- If you know GLSL, Compute Shaders are (fairly) easy!

This is pretty good! So what is talking against it? Not very much, actually.

- Higher hardware demands than OpenCL and CUDA: needs a "Kepler generation" board or better and OpenGL 4.3. This is not much of a problem today.
- Some new concepts. Not much of a problem either since all important things are there.
- No support for 8-bit integers.

- Not part of the main graphics pipeline like fragment shaders. But still much closer than CUDA or OpenCL!
- Some vendors (Apple) are lagging behind. This may be the biggest problem. I can not run compute shaders on my MacBook Pro, despite modern hardware!

Compute shaders run alone, not compiled together with others, but being part of OpenGL, it has direct access to much of the OpenGL features. See Figure 32.

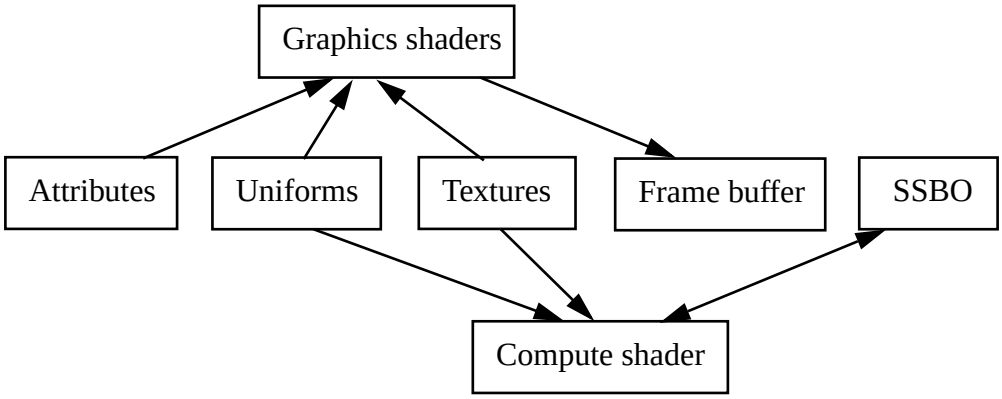


FIGURE 32. Data access for compute shaders and other shaders

In the figure, I have lumped together all the graphics shaders (vertex, fragment, geometry, tessellation) into one. For our purposes we mostly consider fragment shaders, but in this case I rather consider most kinds of shader input, including attributes to vertex shaders.

Since we have no geometry, there can be no attributes per vertex, Uniforms, however, work nicely, and so do textures despite not being able to write them to the frame buffer. So what you see in the upper part of the figure is a data view of the OpenGL pipeline, while the lower shows the more general purpose paths with arbitrary access to and from SSBOs.

So how do I use it? If you know OpenGL, it isn't very hard to get it running. Compilation is just like other shaders, you just don't compile it together with others the way you do with vertex, fragment etc. All you need to do is a trivial change from the usual shader loader/compilation code that you are bound to have already. You just need to compile as GL_COMPUTE_SHADER.

Many things are just like you are used to. You can send uniforms to the shader, just like in other shaders, and you access textures the same way. That makes OpenGL integration far superior than any other solution.

But of course there are a few differences to attend to. We no longer have one thread per fragment, since there are no fragments (output pixels), and thereby no pre-determined output either. The thread number is set to what you want. Input and output data is even more special.

12.2 Shader Storage Buffer Objects

Of course, input data can be textures, but then we are back to the sometimes a bit awkward situation of having to fit data into textures. As a complement, we now have “Shader Storage Buffer Objects”. This is a general buffer type for arbitrary data. We can declare it as an array of structures, which gives us great freedom in what data we want to use. These buffers are read and written freely by Compute Shaders.

Upload input data to SSBO works like this:

```
glGenBuffers(1, &ssbo);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo);
glBufferData(GL_SHADER_STORAGE_BUFFER, size, ptr, GL_STATIC_DRAW);
```

We also need to tell the shader about it. We do that with this call on the CPU:

```
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, id, ssbo);
```

which matches this line in the compute shader;

```
layout(std430, binding = id, buffer x {type y[];};
```

The number of blocks (work groups) is controlled from the host. The number of threads per block can be set either from inside the shade or from the host. In the shader, this is made by another layout call:

```
layout(local_size_x = width, local_size_y = height)
```

In the shader, the thread number is similar to OpenCL, but in this case given as predeclared “in” variables: gl_GlobalInvocation and gl_LocalInvocation, together with the equally intuitive gl_NumWorkGroups, gl_WorkGroupID and gl_WorkGroupSize. Note that these output vec3’s, that is a 3-component vector.

There is also gl_LocalInvocationIndex, which in an integer giving a unique number of each work item in one work group.

Thus, in the simplest cases we can access data using gl_GlobalInvocation and each thread will get a unique item.

You execute the kernel like this:

```
glUseProgram(program);
glDispatchCompute(sizex, sizey, sizez);
```

There is also shared memory. In compute shaders, shared memory is declared *shared*:

```
shared float a[SOMESIZE];
```

The arguments to glDispatchProgram set the number of blocks / workgroups. The number of threads (work items) per block can be set by the shader as above,

In order to access the output data, you use the following calls:

```
glBindBuffer(GL_SHADER_STORAGE, ssbo);
```

```
ptr = (int *) glMapBuffer(GL_SHADER_STORAGE, GL_READ_ONLY);
```

Then read from ptr[i]. When you are done, release the data like this:

```
glUnmapBuffer(GL_SHADER_STORAGE);
```

12.3 Example code

Here follows the complete main program:

```
int main(int argc, char **argv)
{
  glutInit (&argc, argv);
  glutCreateWindow("TEST1");

// Load and compile the compute shader
  GLuint p =loadShader("cs.csh");

  GLuint ssbo; //Shader Storage Buffer Object

  // Some data
  int buf[16] = {1, 2, -3, 4, 5, -6, 7, 8, 9,
                 10, 11, 12, 13, 14, 15, 16};
  int *ptr;

// Create buffer, upload data
  glGenBuffers(1, &ssbo);
  glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo);
  glBufferData(GL_SHADER_STORAGE_BUFFER,
      16 * sizeof(int), &buf, GL_STATIC_DRAW);
// Tell it where the input goes!
// "5" matches "layuot" in the shader.

  glBindBufferBase(GL_SHADER_STORAGE_BUFFER,
         5, ssbo);

// Get rolling!
    glDispatchCompute(16, 1, 1);

// Get data back!
  glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo);
  ptr = (int *)glMapBuffer(
         GL_SHADER_STORAGE_BUFFER,
         GL_READ_ONLY);
  for (int i=0; i < 16; i++)
  {
    printf("%d\n", ptr[i]);
  }
}
```

Here follows a simple Compute Shader:

```
#version 430
#define width 16
#define height 16

// Compute shader invocations in each work group

layout(std430, binding = 5) buffer bbs {int bs[];};
```

```
layout(local_size_x=width, local_size_y=height) in;

//Kernel Program
void main()
{
  int i = int(gl_LocalInvocationID.x * 2);
  bs[gl_LocalInvocationID.x] = -bs[gl_LocalInvocationID.x];
}
```

Note: In this example there are too many threads for data (16*16*16)

12.4 Synchronization in OpenGL compute shaders

Compute shaders has a similar synchronization mechanism. Much of it is shared with the rest of OpenGL. These include:

barrier(); synchronizes the execution. Other barrier calls relate to the memory access:

groupMemoryBarrier(); synchronizes within a work group.

See also: memoryBarrier(); memoryBarrierShared(); memoryBarrierImage(); memoryBarrierBuffer();

There are also synchronization commands from the CPU, e.g. glMemoryBarrier().

12.5 Compute shader timing with query objects

With compute shaders, timing is made with *query objects*. Start timing with

```
glBeginQuery(GL_TIME_ELAPSED, myQuery);
```

and end with

```
glEndQuery(GL_TIME_ELAPSED);
```

Check if it has finished with

```
glGetQueryObjectiv(myQuery, GL_QUERY_RESULT_AVAILABLE, &query_done);
```

Finally, get the time with

```
glGetQueryObjecti64v(myQuery, GL_QUERY_RESULT, &elapsed_time);
```

12.6 Queries in compute shaders

In OpenGL, thereby available both to compute shader and fragment shader solutions, you can use glGetIntegerv/glGetBooleanv/glGetFloatv and glGetInteger64v with various parameters. Perhaps the most vital ones are:

```
MAX_COMPUTE_SHARED_MEMORY_SIZE
MAX_COMPUTE_WORK_GROUP_COUNT
MAX_COMPUTE_WORK_GROUP_SIZE
```

12.7 Conclusions on Compute Shaders

OpenGL Compute Shaders are not only available for stationary computers. They were originally, but since 2014 they are also supported in GLES 3.1 (OpenGL for embedded systems). That was also the time when MESA (the open source OpenGL implementation, e.g. for Linux) became available for Intel GPUs (Haswell). So the support improved over time. Alas, as of 2018 they are still not supported by Apple.

So, do Compute Shaders provide an important alternative? They provide good portability between different GPUs and OSes. They are supported on most GPUs that are of interest today, graphics integration can't be easier, it has the vital features to be competitive. Thus, I only see advantages with them.

12.8 Vulkan

At the time of writing this, Vulkan is still the new player on the field. It was released in 2016. It has been called the new OpenGL, but it is also a new open parallel computing platform.

Will it step in and take over? Well, I would not say that it has so far, but it has made considerable success in the gaming arena. It is cross-platform, it is built for both graphics and general-purpose computations. This is true for OpenGL as well, but Vulkan solves problems that has surfaced in OpenGL over time.

To be precise, the speciality of Vulkan is that it is designed for *multi-threaded host applications*. OpenGL is single-threaded on the host. In many cases I would consider this a minor problem, since many of the big problems run best on the GPU anyway, but there are still many problems of more sequential nature that are best computed on the CPU.

The big challenge in Vulkan is to set up the CPU environment. The compute shaders in Vulkan are *identical to the ones in OpenGL*. Thus, Vulkan as GPU computing platform is little more than one more vote for the importance for OpenGL Compute Shaders.

13. Direct Compute

I don’t think it would be serious of me to ignore Microsoft’s Direct Compute in a volume like this. Direct Compute is part of Microsoft’s Direct X, and first appeared in Direct X 11, but also supports Direct X 10. It first appeared in 2009.

Direct Compute is based on its own kind of compute shaders, and predates the OpenGL compute shaders, which have been in core OpenGL since 2012. In Direct Compute, the shaders are written in HLSL, MicroSoft’s shader language. However, the differences are not major between HLSL and GLSL.

In most ways, Direct Compute is similar to OpenGL compute shaders or OpenCL. You compile with the built-in HLSL compiler, you load data to buffers, you execute the kernel, you download the result.

An unusual concept in Direct Compute is the *resource views*. This is a “view into buffers”, access paths into buffers, that allow hardware acceleration of format conversions as well as (less surprising) hardware accelerated filtering when sampling data.

Otherwise, much is as usual, and again we need to translate some concepts. A *thread group* is the same as what we know as a work group or block. A *thread vector* is a warp. But let’s not dig deeper into that.

Here follows a simple demo. It is based on MicroSoft’s “BasicCompute11” demo but simplified further. Reusable calls (which are very convenient to keep the complexity down) are not listed.

All the code is now doing is to load an array of floating--point numbers, and computing the square root of each element. The problem is embarrassingly parallel, no dependencies. Here follows the shader:

```
// Floating point numbers in a raw

ByteAddressBuffer Buffer0 : register(t0);
RWByteAddressBuffer BufferOut : register(u0);

[numthreads(1, 1, 1)]
```

```
void CSMain( uint3 DTid : SV_DispatchThreadID )
{
    float f0 = asfloat( Buffer0.Load( DTid.x*4 ) );
    BufferOut.Store( DTid.x*4, asuint(sqrt(f0)) );
}
```

Note that the number of threads is specified from within the shader, just like it is in OpenGL Compute Shaders.

The access to the input and output buffers is a bit different than before, but otherwise the differences are not major.

Here follows the CPU program (main program only, reusable code, error checks and declarations excluded):

```
int __cdecl main()
{
    CreateComputeDevice( &g_pDevice, &g_pContext, false );
    CreateComputeShader( L"SimpleDCa.hlsl", "CSMain", g_pDevice, &g_pCS
);
   printf( "Creating buffer and fill it with initial data..." );
    for ( int i = 0; i < NUM_ELEMENTS; ++i )
        g_vBuf0[i] = (float)i;

    CreateRawBuffer( g_pDevice, NUM_ELEMENTS * sizeof(float),
&g_vBuf0[0], &g_pBuf0 );
    CreateRawBuffer( g_pDevice, NUM_ELEMENTS * sizeof(float), nullptr,
&g_pBufResult );

    printf( "Creating buffer views..." );
    CreateBufferSRV( g_pDevice, g_pBuf0, &g_pBuf0SRV );
    CreateBufferUAV( g_pDevice, g_pBufResult, &g_pBufResultUAV );

    printf( "Running Compute Shader..." );
    ID3D11ShaderResourceView* aRViews[1] = { g_pBuf0SRV };
   RunComputeShader( g_pContext, g_pCS, 1, aRViews, nullptr, nullptr, 0,
g_pBufResultUAV, NUM_ELEMENTS, 1, 1 );
    printf( "done\n" );

    // Read back the result from GPU
   ID3D11Buffer* debugbuf = CreateAndCopyToDebugBuf(g_pDevice,
g_pContext, g_pBufResult);
   D3D11_MAPPED_SUBRESOURCE MappedResource;
   float *p;
   g_pContext->Map(debugbuf, 0, D3D11_MAP_READ, 0, &MappedResource);
   p = (float*)MappedResource.pData;

   // Verify that if Compute Shader has done right
   printf("Printing some output...");
   for (int i = 0; i < min(NUM_ELEMENTS, 10); ++i)
      printf("%f %f\n", g_vBuf0[i], p[i]);

    g_pContext->Unmap( debugbuf, 0 );

    debugbuf->Release();

    printf( "Cleaning up...\n" );
    g_pBuf0SRV->Release();
```

```
    g_pBufResultUAV->Release();
    g_pBuf0->Release();
    g_pBufResult->Release();
    g_pCS->Release();
    g_pContext->Release();
    g_pDevice->Release();

   printf("Press key to finish.\n");
   getch();

    return 0;
}
```

Is it mostly similar to OpenGL Compute Shaders or OpenCL? You can be the judge off that, but I conclude that it reminds mostly of those two, with a considerable setup, and support for all important GPU computing needs.

I am not an expert on Direct Compute, so there may be superfluous code, simplifications that I don’t see. Corrections are welcome.

13.1 Shared memory

In Direct Compute, shared memory is called... shared memory! Or more precisely, Thread Group Shared Memory (TGSM), but we feel rather at home with it anyway, don’t we? Just like shared memory in CUDA, it is local to one block/thread group. It is declared like this (in the shader, obviously):

```
groupshared float2 myArray[N];
```

The usual rules apply, using it as temporary storage, and avoiding bank conflicts.

13.2 Synchronization

The second vital detail to add is that of synchronization. Within a group, in shader code, synchronization is made with

```
GroupMemoryBarrierWithGroupSync();
```

Synchronization on CPU level is implicit in the Dispatch() call, which launches the compute shader (according to the more advanced demos from Microsoft)

There is always more to say and more code to show, but for our needs I believe that this is a sufficient introduction to Direct Compute.

14. Comparisons of the platforms

Performance is important, so we should ask ourselves if these different frameworks perform equally well. I have been involved in several such investigations. There are, of course, also many elsewhere, but I will here refer to the ones we have made locally.

One early project, shown in Figure 33, was made by Marco Fratarcangeli, building large mass-spring systems, comparing CPU, CUDA, GLSL and OpenCL. This was originally made as a project in my PhD course in GPU computing, but has since then been published. [18]

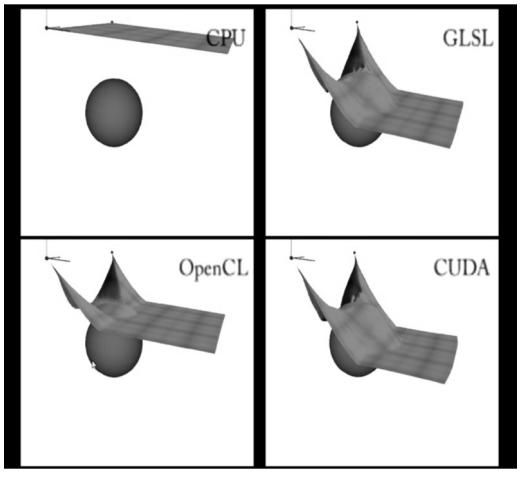


FIGURE 33. Cloth simulation performance comparison

In his investigation, the CPU quickly fell behind. OpenCL was noticeably behind GLSL and CUDA but still a lot faster than the CPU. CUDA and GLSL were almost side-by-side, with a slight advantage to CUDA. On the largest sizes, however, OpenCL either failed completely or had very poor performance.

In 2016, Torbjörn Sörman made the “FFT everywhere” project as his diploma thesis [19]. The Fast Fourier Transform was implemented, in comparable and reasonably optimized implementations. These were then measured for different data sizes, and the result for one of the test cases is shown in Figure 34.

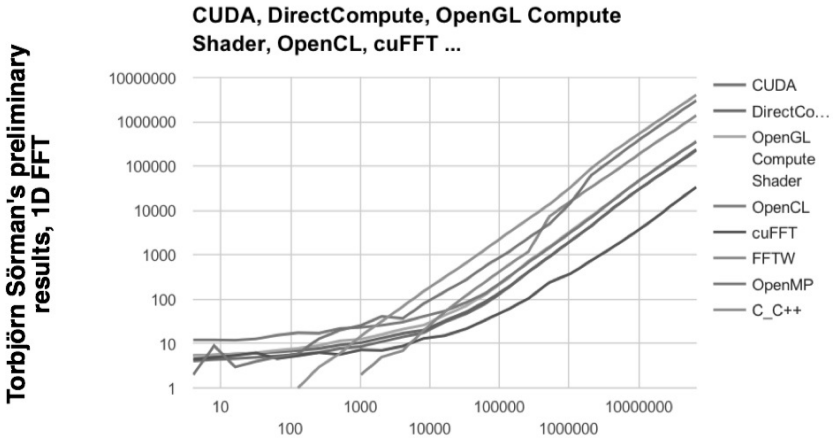


FIGURE 34. Example results from “FFT Everywhere”

Apart from Sörman’s implementation, NVidia’s cuFFT and the FFTW implementations were also used. It is clear from the graph that cuFFT is extremely well optimized, and outperformed everything else by an order of magnitude.

However, the main focus of the investigation was to see how a comparable implementation would perform. For that, we can see that CUDA, Direct Compute and OpenCL were the winners. There was also a test on AMD, where Direct Compute, OpenCL and OpenGL Compute Shaders ran side-by-side.

There are many if’s and but’s in a comparison like this, but there were two clear conclusions:

- Hard optimization (cuFFT and FFTW) pays, and not just by a little.
- OpenCL and Compute Shaders tend to be very close.

An even more recent study (2018) by Adam Söderström reports a different picture. In his study, OpenCL and Direct Compute clearly beat CUDA.

These studies are scarily inconclusive, pointing in different directions, but I believe they tell us one thing: No platform stands out as clear winner every time. There are often a clear winner for your application, and it can pay to try one more platform than whatever you find the most obvious one.

15. Reduction

An interesting class of problems for GPU computing is reduction problems, where a small amount of data is extracted from a larger set. This is a problem of limited parallel nature which still lends itself to parallel implementation.

Examples of reduction algorithms include finding maximum or minimum, calculating median or average, and histograms. These are all common problems.

They are often sequentially trivial, you just loop through the data and find the answer. You add, takes min or max, you accumulate results, you may compute an intermediate histogram to find the answer. But these solutions fit badly in massive parallelism!

A typical solution is to use a tree-based approach, as illustrated in Figure 35, where the maximum of a dataset is calculated. The figure only shows a trivial sized problem, for which a GPU implementation is irrelevant. It is just an illustration of a problem which is rather applied on millions of items.

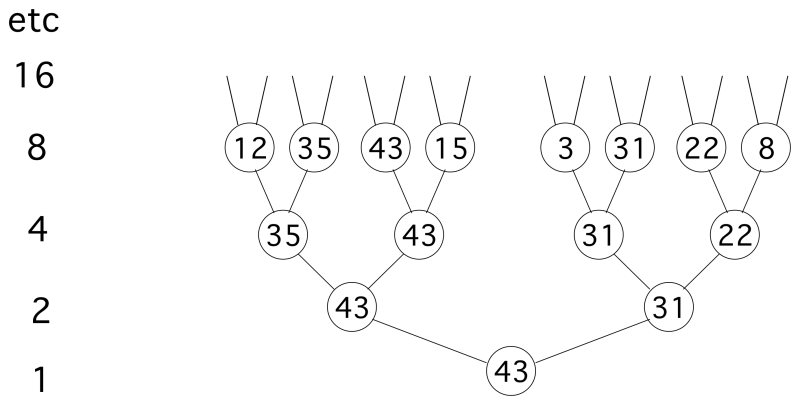


FIGURE 35. Multi-level reduction

With a tree, each comparison is independent of the others, making every level a trivially parallel problem which is split to a lather number of threads. It doesn't have to be 2-to-1. Rather, it is often beneficial to have each thread calculating the max or 4 or 8 items. With a 2D arrangement, we can reduce 4 to 1 as shown in Figure 36.

The parallelism is reduced for each level, as the dataset shrinks to smaller and smaller size. Thus, the computing need to be reorganized to a smaller number of threads. This is done by launching multiple kernel runs with gradually fewer threads.

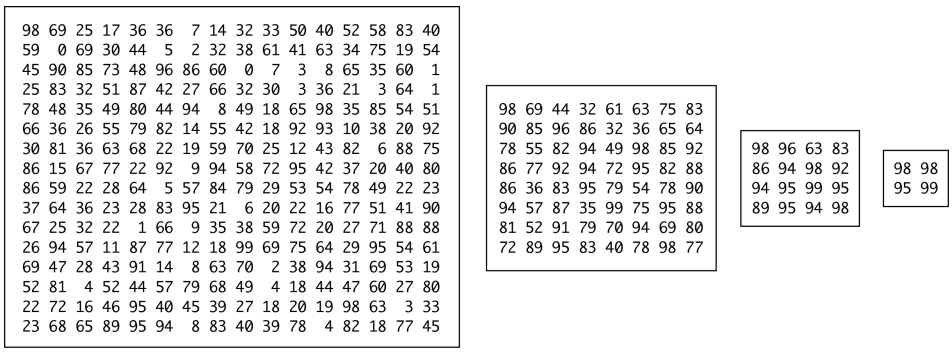


FIGURE 36. 2D reduction finding max

Essentially, we do this:

```
For n = k downto 0 do
   Launch 2n kernels
```

There is a certain overhead in launching a kernel, which is why we should explore how much work to do in one run. We can merge multiple levels into one, but there is a limit where we lose too much parallelism. There is a balance between enabling parallelism and avoiding overhead.

This also means that we should *not* run the entire algorithm on the GPU. Under a certain problem size, there will be no gain in launching a GPU kernel and the rest of the problem should be computed on the CPU.

See Figure 37. Several levels of a 2-to-1 reduction pyramid is performed in each level (kernel run). The final layer, in the bottom, should be performed on the CPU.

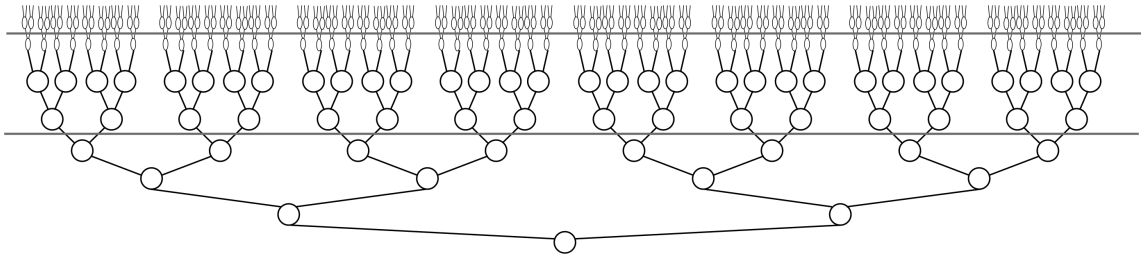


FIGURE 37. Reduction with multiple levels per kernel

Like so often before, we must remember that we can not synchronize between blocks. The multi-level approach helps us to handle this.

15.1 Optimization of reduction

As reported by Harris [17], reduction is a great example of how far optimization can go. These optimizations include:

- Avoid “if” statements, divergent branches
- Coalesced memory access
- Avoid bank conflicts in shared memory
- Optimizing the number of data items handles per thread
- Loop unrolling to avoid loop overhead (classic old-style optimization!)

Harris reports huge speed differences. He reports 30x speedup from naive implementation to optimized. We must expect the number to change with different hardware, but it still illustrates the importance of optimization. The “standard” question of coalescing is only part of the optimization.

We can also note that shared memory has no relevance for the problem. If every item is only read once, we have nothing to gain of temporary storages.

15.2 Parallel prefix sum on GPU

A particularly important technique for much parallel computing, and not least GPU computing, is the *parallel prefix sum* or *scan*. This operation is essential for many algorithms where a parallel version may seem undoable. It is a key operations in many algorithms, for example for implementing QuickSort on GPU (chapter 17).

Parallel prefix sums are not only useful for GPUs. It is just as relevant for e.g. parallel algorithms on CPU. Here, I will describe the principle and some specific details relevant for GPU. The text is based on texts by Harris [23] and Blelloch [24].

Parallel prefix sum is quite related to reduction algorithm, although the output is not reduced to a single data item or small set, but a cumulative sum of the whole data set. For example, the list (1, 2, 3, 4, 5) would be converted to (1, 3, 6, 10, 15). This is, like reduction, an operation that is trivial to perform sequentially, but requires some tricks to compute in parallel.

Sequentially, it is trivially performed like this:

```
a: array[0..max] of Integer;

for i := 1 to max do
   a[i] := a[i] + a[i-1]
```

This is called an *inclusive scan*, where the number produced is the sum of all elements to the left, including the element itself. For an *exclusive* scan, the result is the sum of all elements to the left, excluding the element itself. Thus, our example above, (1, 2, 3, 4, 5) results in (0, 1, 3, 6, 10). The result is simply shifted one step to the right.

In order to compute this efficiently in parallel, we use the parallel method by Blelloch [24]. It is similar to reduction, but computed in two passes, one called the *up-sweep* and then the *down-sweep*. The up-sweep is a straight-forward reduction sum, except for saving all intermediate sums at the rightmost element of each subset. This data is then used by the down-sweep, for creating the final cumulative sums.

The following illustrations are based on figures by Harris [23]. These are stated as based on Blelloch [24], but Harris' pictures are in my opinion clearer and easier to follow. Each figure shows the structure left and an example to the right. The algorithm as illustrated performs an *exclusive scan*.

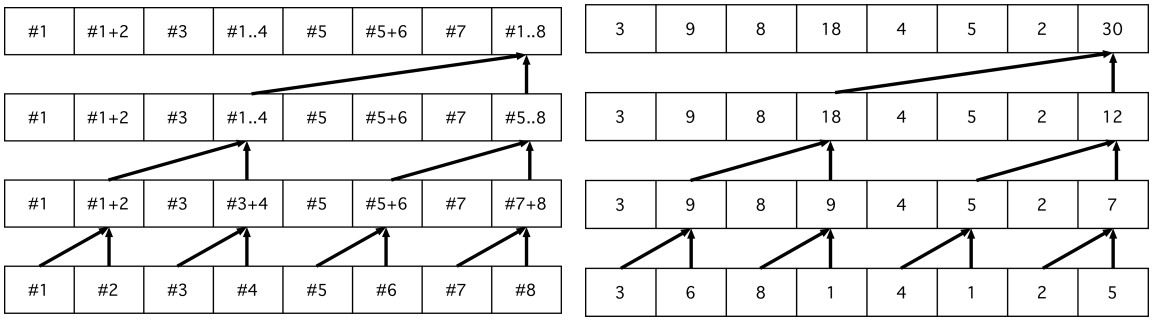


FIGURE 38. Up-sweep

The up-sweep (Figure 38) performs a straight-forward reduction sum. The last stage is actually not needed. Then follows the down-sweep (Figure 39) which is less intuitive, but if you study the structure, you will see that it will eventually store a cumulative sum in each element.

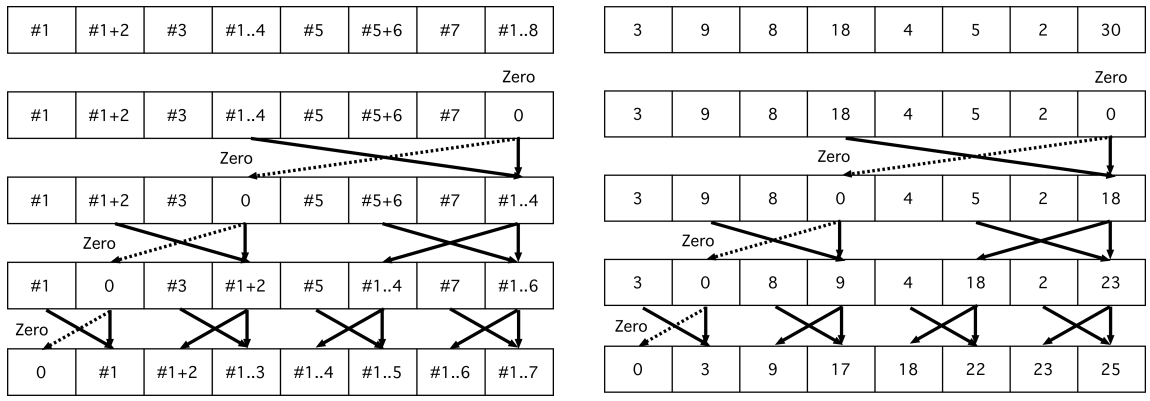


FIGURE 39. Down-sweep

On the GPU, we must perform this as a number of kernel runs, just like the reduction methods above. Most of the optimizations described by Harris [17] (chapter 15.1) apply.

16. OpenGL Interoperability

We have already seen an example that visualize its results with OpenGL, namely the Julia fractal example. There, we did it the simplest, but not fastest way possible: Download output from the GPU computing to the CPU, then upload it to an OpenGL texture. This works, but moving the whole result over the bus just to upload it again is wasteful.

Figure 40 suggests three different scenarios: No visualization, visualization by downloading and uploading again, and finally OpenGL interoperability, where the data stays on the GPU and goes directly from GPU Computing (e.g. CUDA) to OpenGL.

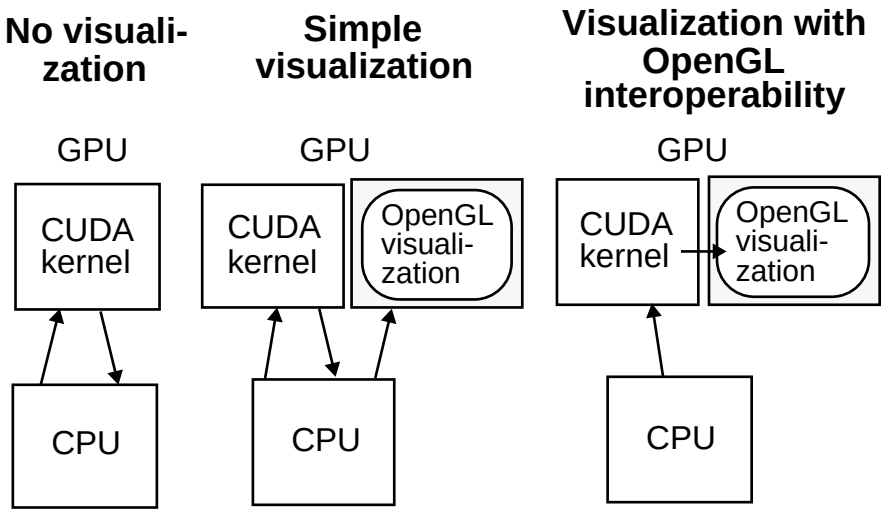


FIGURE 40. CUDA and visualization

16.1 CUDA-OpenGL Interoperability

Let us start looking at how this works in CUDA. This requires a setup that is a bit more complex. We must decide beforehand what data should be shared with OpenGL, and allo-

cate that from the OpenGL side. Then we tell CUDA about it, we register with CUDA, and map a buffer to give CUDA a pointer to the data. Then we can compute, and have OpenGL visualizing the result. Here follows code snippets from our demo of this process:

• Allocate with OpenGL

• Register with CUDA

• Allocate VBO (vertex buffer)

```
glGenBuffers(1, &positionsVBO);
glBindBuffer(GL_ARRAY_BUFFER, positionsVBO);
unsigned int size = NUM_VERTS * 4 * sizeof(float);
glBufferData(GL_ARRAY_BUFFER, size, NULL, GL_DYNAMIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, 0);

cudaGraphicsGLRegisterBuffer(&positionsVBO_CUDA, positionsVBO,
cudaGraphicsMapFlagsWriteDiscard);
```

• Map buffer to get CUDA pointer

• Pass pointer to CUDA kernel

• Release pointer

```
cudaGraphicsMapResources(1, &positionsVBO_CUDA, 0);
size_t num_bytes;
cudaGraphicsResourceGetMappedPointer((void**)&positions, &num_bytes,
positionsVBO_CUDA);printError(NULL, err);

// Execute kernel
dim3 dimBlock(16, 1, 1);
dim3 dimGrid(NUM_VERTS / dimBlock.x, 1, 1);
createVertices<<<dimGrid, dimBlock>>>(positions, anim, NUM_VERTS);

// Unmap buffer object
cudaGraphicsUnmapResources(1, &positionsVBO_CUDA, 0);
```

Simple CUDA kernel for producing vertices for graphics

```
// CUDA vertex kernel
__global__ void createVertices(float4* positions, float time, unsigned
int num)
{
   unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;

   positions[x].w = 1.0;
   positions[x].z = 0.0;
   positions[x].x = 0.5*sin(kVarv * (time + x * 2 * 3.14 / num)) * x/num;
   positions[x].y = 0.5*cos(kVarv * (time + x * 2 * 3.14 / num)) * x/num;
}
```

The result of this example is shown in Figure 41. Our example is based on NVidia's example "SimpleGL", but simplified to significantly briefer code.

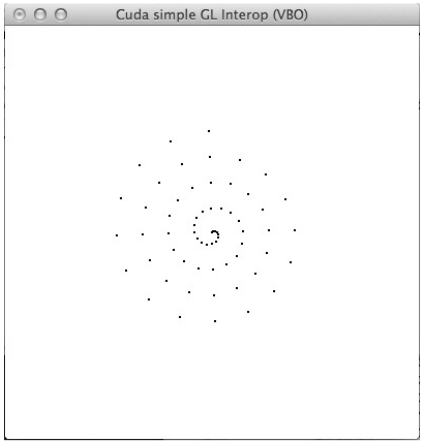


FIGURE 41. Interoperability example

This example only produces a set of vertices, but we are not limited to that. You can draw surfaces, compute textures etc.

However, we should still ask ourselves if we should use CUDA with OpenGL. It is great for visualizing, and faster than going over CPU, but do we really have a problem that benefits from CUDA or would it be better to do it all in OpenGL? Also, consider that OpenGL has CUDA-like functionality built-in, namely Compute Shaders.

We conclude that CUDA can be coupled closer to OpenGL than the simple way we have done before. Moving data back and forth is wasteful, there is performance to gain.

16.2 OpenCL and OpenGL

Interoperability between OpenCL and OpenGL is more complicated, since it requires a changes in the setup phase, the context creation, where you need to add additional properties, and these properties are platform dependent.

With a modified context, the question is what information to share between OpenGL and OpenCL. There are several options: Texture sharing, pixel buffer objects, map buffers with glMapBuffer, and vertex buffer sharing. According to Shevtsov [22], texture sharing is the most efficient choice so it should be preferred.

For doing this, you set up a texture as you always do with OpenGL, except that it should be using GL_NEAREST as filter parameter. Then we can create a reference to it for OpenCL with clCreateFromGLTexture().

It is notable that before you start working on the texture with OpenCL, you should call glFinish() so OpenGL is no longer doing anything with it. Likewise, after OpenCL is done, call clFinish() to be sure, to synchronize.

The calls glEnqueueAquireGLObjects() and glEnqueueReleaseGLObjects() are used to give OpenCL control over a buffer.

Thus, the computation is bracketed with calls like this:

```
glFinish();
glEnqueueAquireGLObjects()
--- do your OpenCL work here ---
clFinish()
glEnqueueReleaseGLObjects()
```

16.3 OpenGL, Compute Shaders and fragment shaders

Finally, let us have a look at the platforms where OpenGL interoperability is by far easiest: Compute shaders and fragment shaders. Since we are working in the OpenGL context all the time, interoperability is not an issue at all. The only real question is how to write to data, like textures, but that is a minor issue. For writing to a texture that you also read from, you use the ImageStore() call. When using fragment shaders, you typically use FBOs so you are writing to textures all the time. You can also consider transform feedback for modifying vertex buffers.

These are all standard operations in OpenGL. Therefore, I choose not to go further on the subject, as being relatively trivial. Just let me stress once more: OpenGL interoperability is by far easiest for these platforms!

17. Sorting on GPUs

Sorting is an important problem that is a challenge to do efficiently on GPUs. I will here present simple but not efficient approaches, the popular bitonic sort and the challenging but efficient QuickSort.

An important aspect when considering a sorting algorithm for parallel implementation is whether the algorithm has *data driven execution* or *data independent execution*.

In data driven execution, the computing pattern depends on the data. This makes it harder to parallellize. One such algorithm is QuickSort.

With data independent execution, the computing pattern is data independent and therefore always the same. One such algorithm is Bitonic sort.

17.1 Bubble sort

Can the classic Bubble sort be made in parallel? We loop through data and compare neighbors. It is extremely sequential in its usual form, and known to be inefficient.

However, a parallel variant of it is very easy to make. For that, we may use a two-phase method called “odd-even sort“. We use an “odd phase” and an “even phase” where we compare even indexed items to either the higher or lower neighbor. This is fully sorted after n phases.

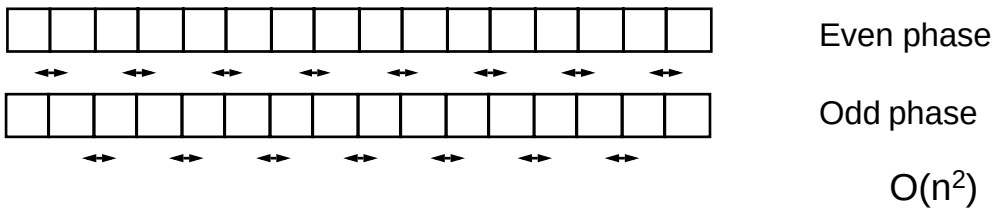


FIGURE 42. Two-phase Bubble sort variant, odd-even sort

This isn’t as bad as it first sounded. It is data independent and has excellent data locality. If we have all data available in one chunk, this will work perfectly. However, on a GPU that

implies that it must fit in one single block. To overcome this, we would need to run multiple kernel runs. And that is where it fails.

17.2 Rank sort

Rank sort is a simple and in its sequential form inefficient sorting algorithm. It requires a set of items to sort where all values are different, or an extension to expand multiple values to multiple locations.

The algorithm works like this: For each item, count the number of items that are smaller. Then you know the position of that item in the sorted data.

This is very easy to parallellize. You need one thread per item. Each thread loops through the entire data set. When the thread is done, it stores its value in a destination array, using the count of smaller items as index.

Being an $O(n^2)$ algorithm, doesn't it perform badly? It is not a top performer, but it is not as bad as it seems at first look. It is data independent, which makes it highly suitable for parallel implementation. It also has excellent locality, with multiple threads accessing the same item simultaneously. This makes it especially good for broadcasting (e.g. constant memory), but also suitable for acceleration using shared memory.

Let us look at how this can be implemented. For this example, I choose to use OpenCL.

The CPU part looks essentially the same as for all OpenCL programs (computing part only):

```
int gpu_Sort(unsigned int *data, unsigned int length)
{
  cl_int ciErrNum = CL_SUCCESS;
  size_t localWorkSize, globalWorkSize;
  cl_mem in_data, out_data;

  in_data = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, length * sizeof(unsigned int), data, &ciErrNum);
  out_data = clCreateBuffer(cxGPUContext, CL_MEM_READ_WRITE, length *
sizeof(unsigned int), NULL, &ciErrNum);

  if (ciErrNum != CL_SUCCESS)
  {
    printf("Error: Failed to allocate memory on the device\n");
    return ciErrNum;
  }

    localWorkSize  = 128; // Can be adjusted for best balance
    globalWorkSize = length;

    // set the args values
    ciErrNum  = clSetKernelArg(gpgpuSort, 0, sizeof(cl_mem),  (void *)
&in_data);
    ciErrNum  = clSetKernelArg(gpgpuSort, 1, sizeof(cl_mem),  (void *)
&out_data);
```

```
    ciErrNum |= clSetKernelArg(gpgpuSort, 2, sizeof(cl_uint), (void *)
&length);

    if (ciErrNum !=CL_SUCCESS)
    {
      printf("Error: clSetKernelArg failed");
      return -1;
    }

    gettimeofday(&t_s_gpu, NULL);

    cl_event event;
    ciErrNum = clEnqueueNDRangeKernel(commandQueue, gpgpuSort, 1, NULL,
&globalWorkSize, &localWorkSize, 0, NULL, &event);

    clWaitForEvents(1, &event); // Synch
    gettimeofday(&t_e_gpu, NULL);
    printCLError(ciErrNum);

   ciErrNum = clEnqueueReadBuffer(commandQueue, out_data, CL_TRUE, 0,
length * sizeof(unsigned int), data, 0, NULL, NULL);
   printCLError(ciErrNum);

   clReleaseMemObject(in_data);
   clReleaseMemObject(out_data);
   return ciErrNum;
}
```

Most of the CPU code is similar to any OpenCL code. One detail deserves mentioning: The localWorkSize is set to 128 here. With 1024 or more threads per block (work group) a larger number can make better use of the hardware.

Here is simple kernel for the problem. All threads are independent and perform their task.

```
__kernel void sort(__global unsigned int *data, __global unsigned int
*outdata, const unsigned int length)
{
  unsigned int pos = 0;
  unsigned int i;
  unsigned int val;

  //find out how many values are smaller
  for (i = 0; i < get_global_size(0); i++)
    if (data[get_global_id(0)] > data[i])
      pos++;

  val = data[get_global_id(0)];
  outdata[pos]=val;
}
```

As mentioned above, this can be optimized using constant or shared memory. Here is a kernel that uses shared memory:

```
__kernel void sort(__global unsigned int *data, __global unsigned int
*outdata, const unsigned int length)
{
   unsigned int pos = 0;
   unsigned int i, b;
```

```
   unsigned int val;
   unsigned int this;

   unsigned int __local buf[128];

   // loop until all data is covered

   this = data[get_global_id(0)];

   for (b = 0; b < length; b += 128)
   {
      // Get data
      buf[get_local_id(0)] = data[get_local_id(0) + b];

      // Synch
      barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);

      //find out how many values are smaller
      for (i = 0; i < 128; i++)
         if (this > buf[i]) // data[b + i])
            pos++;

      // Synch
      barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
   }

   outdata[pos] = this;
}
```

Note how the shared memory (__local) is used to read blocks of data, just a single item per thread, which is then read by all threads. Also, synchronization (barrier) is now essential.

What is interesting here is really the performance of such a simple algorithm, and the impact of the optimization. With 16384 items, I measured the following times (single-threaded on CPU);

- CPU: 1303 ms
- Not optimized: 71 ms
- Optimized: 61 ms

This was computed on a MacBook Pro with a 2.3 GHz Intel Core i7 CPU and an NVIDIA GeForce GT650M with 512 MB.

However, 16k items is a rather small problem for a GPU. With 65536 items (64k) we see a considerable difference:

- CPU: 28857 ms
- Not optimized: 535 ms
- Optimized: 272 ms

As the startup cost of the GPU computing gets less and less significant, the acceleration compared to the CPU gets bigger, 21 times at 16k, 106 times at 64k. At the same time, the impact of the optimization gets more and more significant.

17.3 Bitonic sort

Bitonic sort, also known as *Bitonic Merge Sort*, or *Batcher's Bitonic Sort*, is a sorting method based on the properties of bitonic sets. A bitonic set is defined as a range of numbers consisting of two monotonic sub-parts, varying in different direction, one increasing and the other decreasing. See Figure 43.

Ken Batcher, who invented the algorithm [21], states (not exact quote):

"Let a be a bitonic set with a maximum at k, consisting of two monotonic parts, one increasing, a- (from item 1 to k) and one decreasing, a+ (k+1 to n)

Then two new sets can be constructed as

$a' = min(a_1, a_{k+1}), min(a_2, a_{k+2})\ldots$

$a'' = max(a_1, a_{k+1}), max(a_2, a_{k+2})\ldots$

These two sets are also bitonic and $max(a') \leq min(a'')$*."*

The algorithm is based on this property. It lets us split the data set in exact halves. See Figure 44. Since these halves are also bitonic, they can also be split in half and eventually the entire data set is sorted.

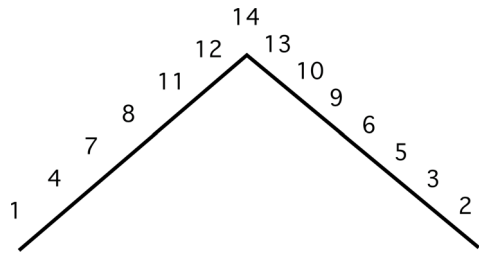


FIGURE 43. A bitonic data set is simply a "pyramid", two sorted halves, sorted in different direction

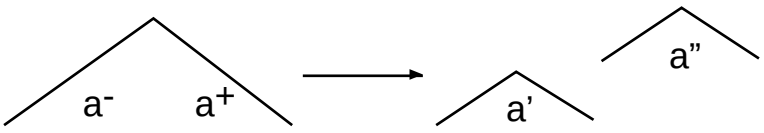


FIGURE 44. By comparing each side with the other, two new bitonic sets are created.

In order to do this, we just have to get to the first bitonic set. But this can, too, be made by bitonic sort! Start by sorting very small parts (pairs) to make small bitonic sets. These are sorted alternating increasing or decreasing, so each pair of subsequences form a bitonic set

The full algorithm is illustrated in Figure 45. Note the grouping. Each part sorts the data to a certain size, and then the following stages can use that data for sorting to parts of double size of the previous level. Figure 46 shows the algorithm with example data.

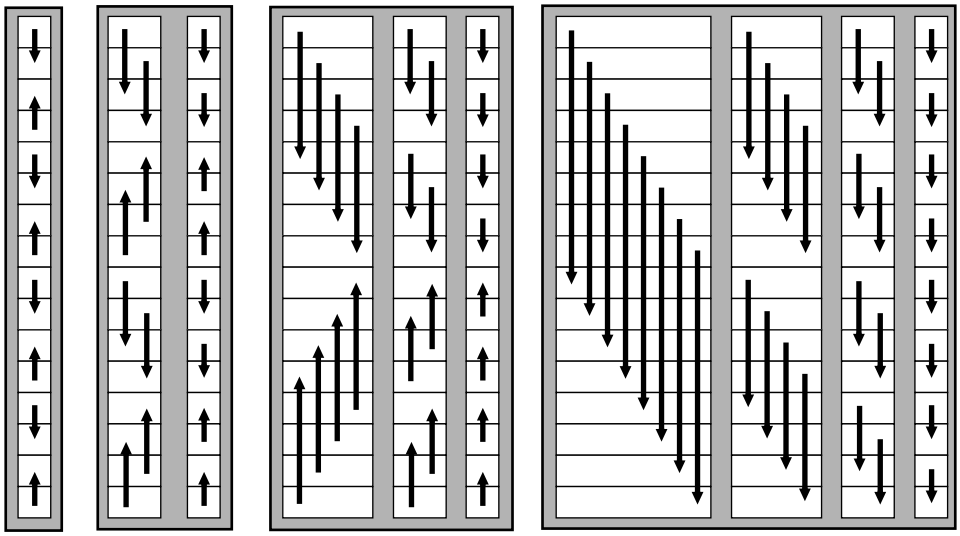


FIGURE 45. Bitonic sort. Phases marked, each producing sorted data for a part of the data.

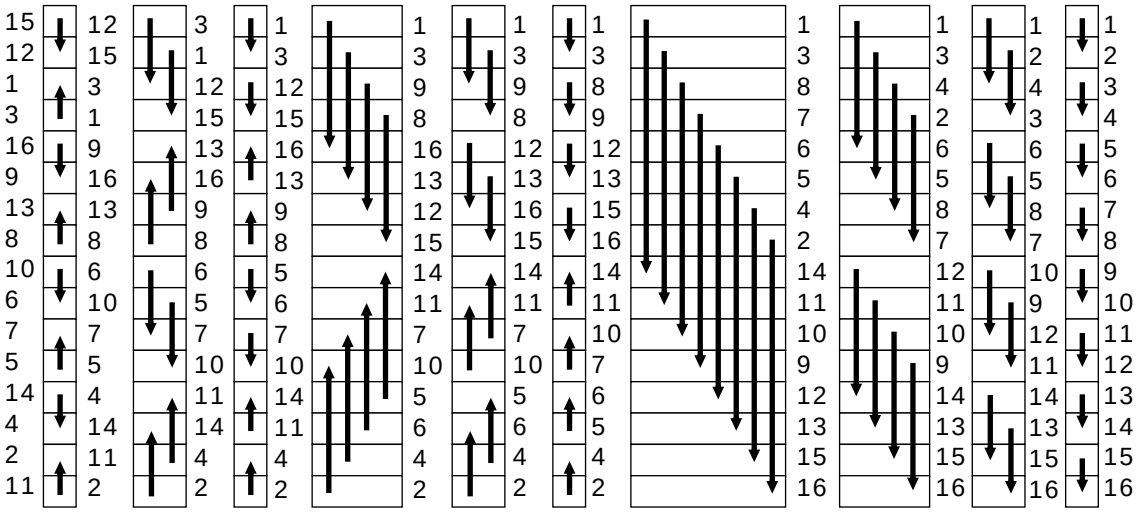


FIGURE 46. Bitonic sort with example data.

I found the code below on-line. If I understand the comments right, it was written by Nikos Pitsianis. [20] Since this implementation relies on nested for loops, it is very suitable for converting to GPU code. Most other examples that I find are written as recursive code, which is not only likely to be less efficient on the CPU but also much harder to parallellize.

```
static void exchange(unsigned int *i, unsigned int *j)
{
   int k;
   k = *i;
   *i = *j;
   *j = k;
}

void bitonic_cpu(unsigned int *data, int N)
```

```
{
  unsigned int i,j,k;

  printf("CPU sorting.\n");

  for (k=2;k<=N;k=2*k) // Outer loop, double size for each step
  {
    for (j=k>>1;j>0;j=j>>1) // Inner loop, half size for each step
    {
      for (i=0;i<N;i++) // Loop over data
      {
        int ixj=i^j; // Calculate indexing!
        if ((ixj)>i)
        {
          if ((i&k)==0 && data[i]>data[ixj])
            exchange(&data[i],&data[ixj]);
          if ((i&k)!=0 && data[i]<data[ixj])
            exchange(&data[i],&data[ixj]);
        }
      }
    }
  }
}
```

From the same source, I could also find a recursive implementation. That code is bigger, most likely slower due to too many function calls, and irrelevant for us since it is much harder to rewrite to GPU code.

The code above is rather so compact that it is hard to understand. The “magic” is to get those steps right, which is made with a few simple calculations to figure out whether we are in the upper or lower part of a pair that should be compared, and what direction it should go. This is calculated from the stage number and stage length.

Bitonic sort is *data independent*. There is no worst case, it always runs in the same time for a certain size. It is pretty fast, $O(n\log^2 n)$ (Why?) However, this is higher complexity than the fastest algorithms, which run in $O(n\log n)$.

A critical question is that of locality. It has good locality in some parts, namely the parts where the subpart being sorted fits in a single block. However, if this is not the case, we need to make multiple kernel runs. This can be elaborated on quite a bit. We can handle several comparisons in a single thread in order to handle more data in a single block. That way we can run rather large parts of the algorithm in each block, but we should avoid to not leave SMs idle just to avoid multiple kernel runs. We need to find the best balance to optimize the problem.

17.4 QuickSort

QuickSort is a very popular algorithm for sorting. It is very efficient for sequential implementations. It can be summarized as follows (see Figure 47):

- Choose a pivot item
- Compare to pivot, form two subsets.

- For each subset, sort subset with QuickSort.

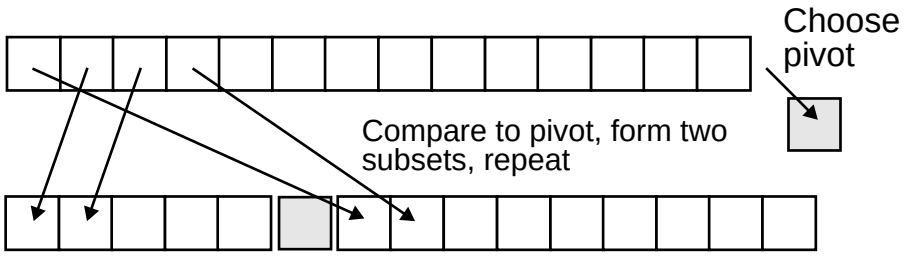


FIGURE 47. QuickSort; Choose pivot and split

The algorithm is data driven; it makes a data dependent reorganization. The execution and data sizes are non-uniform, the recursion goes to different depth in different parts.

QuickSort is fast: $O(n \cdot \log n)$ in typical cases, but it is $O(n^2)$ in the worst case.

It has a fancy name - nobody expect QuickSort to be nothing but optimal. It is indeed good, but not perfect. The data dependent execution makes it less suited for parallel implementation. QuickSort on GPU was initially ignored as impractical, but CUDA implementations exist. This can be motivated by GPUs becoming increasingly flexible, but the GPU implementations are actually perfectly suited even for early CUDA capable GPUs.

The description below is loosely inspired by Cederman & Tsigas [11].

To make a parallel QuickSort, we should consider each stage:

- Pivot selection.
- Partitioning
- Concatenate result

17.4.1 Pivot selection

Usually just grab one. There is little parallelism in this stage, more than that it needs to be performed for each current subsection.

17.4.2 Comparisons

These can easily be run in parallel. On thread for a single comparison works well on a GPU. What is more important is where to put the result. Use an intermediate array of booleans. After each comparison, store the result in the array. Note that we can not immediately put the data items densely packed in two arrays because that would cause racing problems. If we try doing it with atomics, we would serialize the access and performance would suffer.

17.4.3 Partitioning

This is the critical stage, the one that is hardest to run in parallel. For doing that, we can use a parallel prefix sum (see chapter 15.2).

Once we have filled the array of booleans above, we can do the partitioning using the array. In order to do that, we need to perform a parallel prefix sum, making a cumulative sum of the number of zeroes (false) and one for the number of 1's (true) in the array of booleans. This means that the prefix sum gets sums that tell exactly how many zeros och ones that exist to the left of an item. This number tells a thread exactly where to store an item.

Thus, the problem turns into a binary parallel prefix sum. Figure 48 shows an example based on the more general parallel prefix sum in chapter 15.2.

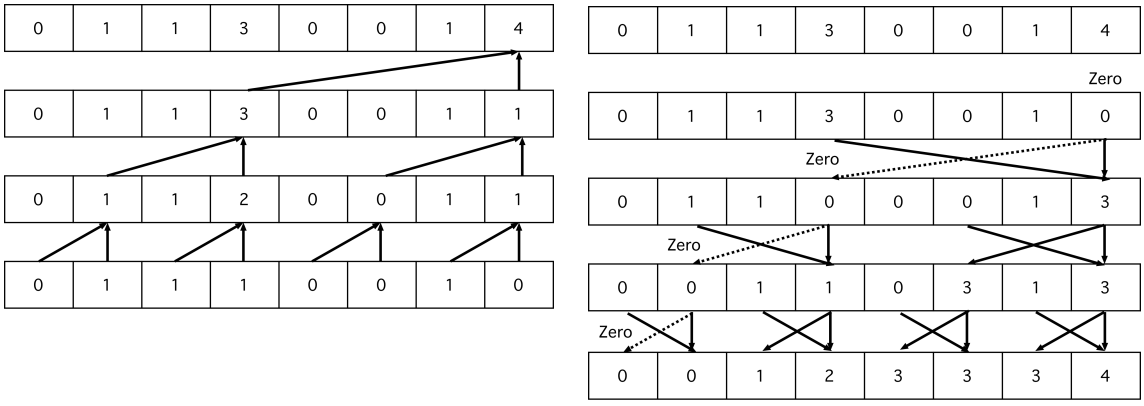


FIGURE 48. Binary parallel prefix sum for calculating partitioning

For every item corresponding to a 1 in the left figure, the prefix sum in the right can be used as local index in the data array.

Like with reduction algorithms, the parallel algorithm above should be replaced with sequential versions when the amount of data in each part is small enough.

17.4.4 Concatenate result

Concatenation is a matter of storing the size of each section in partitioning. With proper bookkeeping of data sizes, each subpart can store its data in proper places.

We conclude that QuickSort is not impossible, but more complex than before.

17.5 Recursion, Concurrent kernels, Dynamic Parallelism

QuickSort is written as a recursive process, but GPUs can't do recursion efficiently, or can they? An option for GPU algorithms that may suit sorting well is recursion though concurrent kernels. These are available since Kepler, that is compute capability 3, which is cur-

rently (2018) the bottom line for GPUs actively supported by NVidia. This allows kernels to spawn new kernels, so kernels are no longer only launched from CPU!

This means that we can perform recursion by spawning new kernels! This can mean less work for the CPU to manage the computation.

18. Image filters

Since GPUs are designed for synthesizing images, other image technologies are obvious applications for GPU computing, and the task of filtering images is perhaps the most fundamental problem of this kind.

I will focus on linear filters, that is filters that can be applied as convolution. I assume that you know the definition of convolution, which is a very common concept in signal processing.

Convolution filters include low-pass (blur) filters (Figure 49), gradient filters and Laplacian. The simplest low-pass filter is the box filter, while what you most likely want is a gaussian filter.

FIGURE 49. Example of low-pass filter on the dandelion image.

In Figure 50, we see a 5x5 box filter (left), a 5x5 approximation of a gaussian (middle), a gradient filter (a.k.a. Sobel filter, top right) and a Laplacian (bottom right).

Note the normalization at the corner! For the low-pass filters, we should normalize in order to create a result with the same average level as the original. How the high-pass filters should be normalized is not as obvious.

FIGURE 50. Linear image filters, convolution filters.

These convolution kernels can be applied as is and produce the expected results. However, the number of computations and memory accesses can get pretty large as kernels get bigger. Let us continue with possible ways to optimize them.

Now, how does this map to the GPU? We can trivially apply the filters just by accessing all pixels in the neighborhood, multiplying with the weight and get a result.

Making a more optimized version can be made using shared memory. The solution is slightly more complicated than the matrix multiplication. Just like for that, we read patches to shared memory and have multiple threads use that data. However, each patch now needs to access an input that is larger than the output patch. The requires some consideration.

Each block should be responsible for the output to a specific output patch. Each thread should be responsible to reading a part of the input. I would balance this work as well as possible, having each thread reading as equal amounts of memory as possible. However, what is likely to be more important is that the reads are coalesced. These two demands are in no conflict with each other. Rather, they fit well as long as you are not tempted to make strided accesses. If you have 2x the output, have all threads read one item each in succession, and then the next sequence.

Also, don’t read image data one byte at a time. One 4-byte pixel at a time is a good chunk.

Remember that a thread does not have to read the same pixel into shared memory that it will output! Giving each thread an equal amount of data (or less) to read to shared memory is both easy and efficient. The amount of work for each thread should be as balanced as possible.

Don’t forget to plan for the border of the patch, the overlap between patches. With an NxN filter, you will get a (N-1)/2 overlap on each side.

18.1 Separable filters

Two convolution kernels applied after each other on the same signal (image) will produce the same result as the convolution of the two kernels applied on the image. A common way to accelerate filters, not least LP filters like this, is to design *separable filters*, two or more filters that in combination will produce the desired filter, and since these filters are smaller, they may provide good optimizations. This is particularly true if we can split a 2D filter into two 1D filters.

We can trivially split an NxN box filter in two, one in each direction, 1xN and Nx1. Applying these in succession will produce the same box filter. See Figure 51.

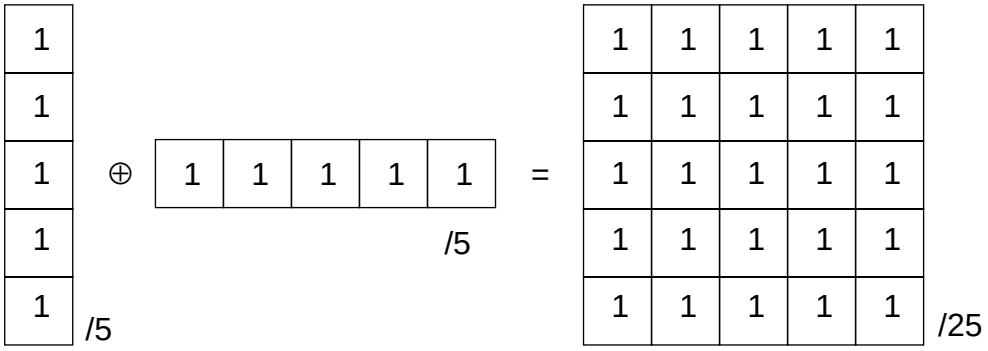


FIGURE 51. Separable box filter.

The same principle holds for some more complicated filters. The gaussian approximation shown in Figure 50 is such a filter.

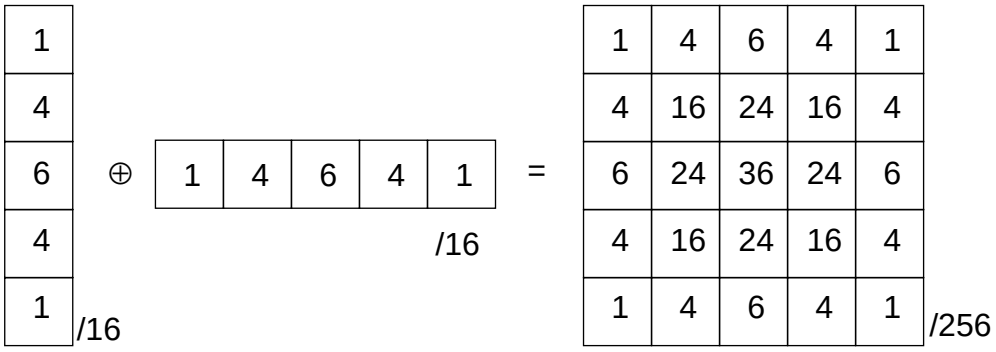


FIGURE 52. Separable gaussian filter.

However, we can take this to extremes! The gaussian in Figure 50 can be separated all the way down to small 1x2 filters. See Figure 53.

This is not only an elegant trick. This can be mathematically proven, using the Central limit theorem.

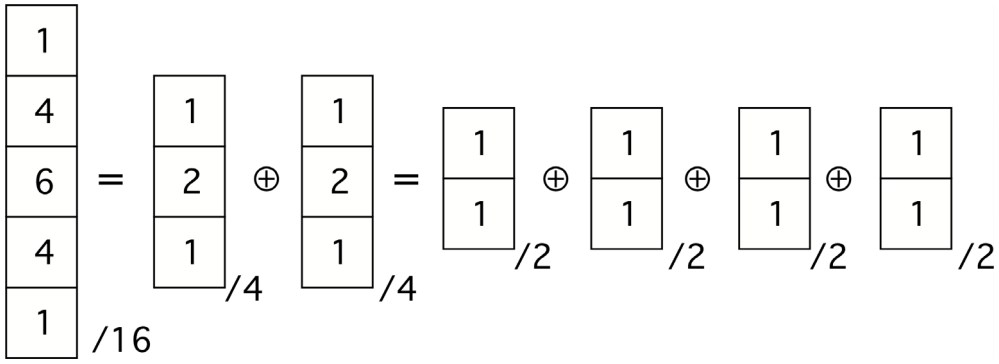


FIGURE 53. Multiple box filters approximating a gaussian.

More intuitively, you can compare this to the statistical outcome when rolling dice. If you roll a since 6-sided die, you have 6 outcomes, all with the same probability. If you roll two, the sum has a probability distribution with the shape of a pyramid. With three dice, you get a distribution that starts to look gaussian, just like the 1x5 filter in Figure 53

18.2 Non-linear filters

Non-linear filters are all filters that can not be expressed as a convolution. One such filter is the *median filter*. The median filter outputs the median of a neighborhood. In the 1D case, that means that the median of the set [1,1,2,7,9] is 2 while the average is 4. In image processing, an application of median filters is noise suppression. The method suppresses noise while preserving edges better than a low-pass filter does.

We need some way to find the median. A naive way that works well for small neighborhoods is sorting. For larger amounts of data, a histogram based solution is more efficient.

It should be noted that a 2D median filter is *not* separable! You can, however, make a separable approximation, making a 1D median in one dimension, and then the median of the result. That is, however, only as approximation.

18.3 Edge checks, clamping

When applying a filter kernel to an image, you will unavoidably reach outside the image. This has to be handled somehow.

If you use texture memory (chapter 9.5), you can take advantage of the hardware edge tests, which will give a good result automatically, and avoid wasting code and cycles on edge tests. For this, you can use clamp or repeat. I would recommend clamp. That will mean that we create an estimation of the first pixels outside the image. Repeating the same pixels as the edge is a pretty decent estimate.

One alternative, which I don’t like, is to skip edges altogether, leaving them without filtering or not producing any output. That will shrink your image or produce visible artifacts. I can’t recommend that.

But you can do clamping in software too! In my code for the current (2018) lab 5 in the TDDD56 course, this is solved for you, like this (un-optimized code):

```
   if (x < imagesizex && y < imagesizey)
   {
// Filter kernel (simple box filter)
   sumx=0;sumy=0;sumz=0;
   for(dy=-kernelsizey;dy<=kernelsizey;dy++)
      for(dx=-kernelsizex;dx<=kernelsizex;dx++)
      {
         // Use max and min to avoid branching!
         int yy = min(max(y+dy, 0), imagesizey-1);
         int xx = min(max(x+dx, 0), imagesizex-1);

         sumx += image[((yy)*imagesizex+(xx))*3+0];
         sumy += image[((yy)*imagesizex+(xx))*3+1];
         sumz += image[((yy)*imagesizex+(xx))*3+2];
      }
   out[(y*imagesizex+x)*3+0] = sumx/divby;
   out[(y*imagesizex+x)*3+1] = sumy/divby;
   out[(y*imagesizex+x)*3+2] = sumz/divby;
```

The clamping is made with the min() and max() functions in the middle. Note that I use min() and max() rather than if statements. Why? It avoids branching! Branching may cause extra processing, but a max or min operation can be made without branching.

18.4 Color images

In the text above, I discuss images as if they were monochromatic. Most images are, of course, color images. They are usually represented in RGB format. There are other representations, like HSV, which can be advantageous for analysis, but I will assume RGB. The three channels are usually processed independently.

The image data is usually stored in interleaved format, “chunky pixels”. A common format is to use one byte per color plus optimally one for alpha (transparency) which requires 24 or 32 bits, that is 3 or 4 bytes.

However, this may not be so practical when processing. GPUs are not made to process bytes, but are best at 32-bit numbers. Perhaps most importantly, if you only load 3 channels, and load one at a time (which I deliberately do in the code example above) you can end up with a strange memory access pattern, with no coalescing. It would be preferable to load an entire pixel at a time, plus include the alpha even if you are not using it.

18.5 Scatter vs. gather

When writing filters, we have one important choice: To make a “scatter” or “gather” filter. The “scatter” filter takes the center pixel, produces the product with each weight and adds the result to a neighbor, while a “gather” filter takes each pixel in the neighborhood, multiplies it with a weight, and stores the result as the center pixel of the output.

In fragment shaders, the shader produces the output for one pixel. Although it is possible for a shader to write into other pixels, the setup is really made for gather solutions.

How about when using CUDA, OpenCL or compute shaders? You should still prefer gather. It requires less synchronization, which comes for a cost. A scatter solution will easily get racing problems.

19. Questions

This chapter is dedicated to exercises, various questions and tasks that you can use for checking that you picked up what I think you should know. It is a collection of past exam questions, and of course I may reuse them in the future. For the 2018 edition, I will not have time to find all duplicates or disturbingly similar questions. I am sorry about that, but I think you would be more sorry about not getting this material.

19.1 Lecture questions

This collection of questions go all the way back to my PhD course in GPU Computing. I still think they are a nice and still use them as “teasers” for my lectures. There is currently an overlap between these and the questions below.

19.1.1 Lecture 1

1. How can a GPU be much faster than a CPU?

2. Why is the G80 so much faster than the previous GPUs (e.g. 7000 series)?

3. A texturing unit provides access to texture memory. What more is it than just another memory?

4. What current trend is driven by the GPU evolution?

19.1.2 Lecture 2

1. What concept in CUDA corresponds to a SM (streaming multiprocessor) in the architecture?

2. How does matrix multiplication benefit from using shared memory?

3. When do you typically need to synchronize threads?

19.1.3 Lecture 3

1. Why can using constant memory improve performance?

2. What is CUDA Events used for?

3. What does coalescing mean and what should we do to get a speedup from coalescing?

4. How can you efficiently calculate the maximum of a dataset in parallel?

19.1.4 Lecture 4

1) What kind of devices will OpenCL run on?

2) What does an OpenCL work group correspond to in CUDA?

3) What geometry is typically used for shader-based GPU computing?

4) Are scatter or gather operations preferable? Why?

19.1.5 Lecture 5

1) How can you efficiently compute the average of a dataset with CUDA?

2) In what way does bitonic sort fit the GPU better than many other sorting algorithms?

3) What is the reason to use pinned memory?

4) What problem does atomics solve?

19.2 GPU Algorithms and Coding (GPU Algorithms)

These questions tend to ask for more algorithmic answers. Therefore, they tend to be the hardest. In older exams, I had a single 5 point question for this section. Later, I tend to split it in two, which makes life easier for me as well as for students.

The following CUDA kernel performs a rank sort (as in lab 6). However, the code is inefficient. Rewrite it for better performance. For full score, the code should be able to accept any length. Make comments to clarify what you do and why.

You may make any assumptions you like about block size, document as necessary.

Minor bugs, syntax errors, as well as mistakes in names of built-in symbols are generally ignored, no point deductions for things that you would easily look up in documentation (like, how many underscores you should use for that particular modifier or exact names on function calls) or that is trivial to fix on the first compilation (within reason). Your effort

should be on describing your algorithm properly, including important concepts. Relevant pseudo-code qualifies for score (partial or even full depending on detail and relevance).

```
__global__ void gpu_Sort ( unsigned int *data,
                           unsigned int *outdata, int length )
  {
    unsigned int pos = 0;
    unsigned int i, ix;
    ix = blockIdx.x * blockDim.x + threadIdx.x;
    //find out how many values are smaller
    for (i = 0; i < length; i++)
       if (data[ix] > data[i])
          pos++;
    outdata[pos] = data[ix];
```

Matrix transposing is an operation with no computations, but its efficiency depends heavily on a certain GPU computing feature. Which feature, why is it important? With code or pseudo code, describe an efficient way to implement matrix transposing. (5p)

(Hint: The operand matrix is given in global device memory of the GPU. The matrix transpose needs not be done in-place.)

A histogram is an array h that records for each possible (integer) value the number of its occurrences in a large integer-valued data structure, e.g., an array a. It can be computed like this:

```
for all elements i in a[] do
    h[a[i]] += 1
```

(a) With what feature can this algorithm be made to run in parallel in CUDA? (1p)

(b) Suggest a different approach that should give good (better) performance and does not rely on this feature (written in ”plain” CUDA, OpenCL or shaders). (2p)

(c) Write this algorithm in CUDA or OpenCL code. (Minor syntax errors are ignored.) (2p)

Describe, in code or sufficient detail, how matrix multiplication of large matrices can be implemented on the GPU. (GPU kernel code only.) Emphasize the most vital considerations for good performance. (5p)

Write code/pseudo-code for computing a 2-dimensional color image filter of size 5 x 5 pixels in a reasonably optimized way. Clearly describe what optimizations you do and why. The filter weights should be specified (i.e. a 5 x 5 matrix), and should be normalized

properly. Full score requires a close-to-real-code solution taking more than one optimization technique into account. You may use CUDA-style syntax or OpenCL- style syntax as you please.

(a) A Mandelbrot algorithm is given as sequential code as follows:

```
for (int x = 0; x < SIZE; x++)
    for (int y = 0; y < SIZE; y++)
        data[x, y] = computeFractal(x, y);
```

that is, the fractal computing code is already available. How can you port this to an efficient GPU implementation? Outline vital parts of the code. (2p)

(b) Describe, in pseudo code and figures, how an optimized matrix multiplication can be performed on the GPU. Your answer should focus on structure and vital features rather than detailed code. (3p)

(a) A large matrix is given, stored in global GPU memory. Describe, using code or pseudo code, an efficient way to transpose it on the GPU. The transposing does not have to be done in-place. Vital features of the algorithm should be clearly stated. (3p)

(b) The following algorithm (given as OpenCL code) performs rank sort on the GPU, a simple but not very efficient sorting algorithm for data with unique keys. However, it has a bug, plus, it can be significantly accelerated.

```
__kernel void sort( __global unsigned int *data,
                    const unsigned int length )
{
  unsigned int pos = 0;
  unsigned int i;
  unsigned int val;
  //find out how many values are smaller
  for (i = 0; i < get_global_size(0); i++)
    if (data[get_global_id(0)] > data[i])
      pos++;
  val = data[get_global_id(0)];
  data[pos]=val;
}
```

What is the bug? (1p)

(c) Describe a way to accelerate the code. (2p)

(a) Describe, using code or pseudo code, how to transpose large matrices efficiently on the GPU. You may be assume square shaped matrices. (2p)

(b) Describe, with code or pseudo code, how reduction can be used to calculate the maximum value of a large array of scalar values on a GPU. (3p)

(a) Write code or detailed pseudo code for calculating the maximum value of a dataset using reduction on a GPU. Both GPU code and the relevant CPU code should be included. The approach should be reasonably optimized. Point out the most important optimization considerations and mark where in the code this occurs. (3p)

(b) A Mandelbrot algorithm is given as sequential code as follows:

```
for (int x = 0; x < SIZE; x++)
    for (int y = 0; y < SIZE; y++)
        data[x, y] = computeFractal(x, y);
```

that is, the fractal computing code is already available (you do not have to write it). How can you port this to an efficient GPU implementation? Outline vital parts of the code, both CPU and GPU sides. (2p)

19.3 GPU Conceptual Questions (AKA GPU Architecture concepts or GPU Computing)

This is the section with questions that are (generally) not supposed to be answered with code or code-like descriptions.

(a) A GPU computation calculates 2048 elements. Each element can be computed in its own thread. The algorithm is not sensitive to any particular block size. It may run on many different GPUs. What number of threads and blocks would you use in such a case? Motivate your answer. (2p)

(b) Describe how computing is mapped onto graphics in shader-based computing (expressed as kernel, input data, output data and iterations over the same data). What limitations are imposed on your kernels compared to CUDA or OpenCL? (3p)

(a) List three different kinds of GPU memory and describe for each their characteristics in terms of performance, usage and accessibility. CUDA terminology is assumed, please note if you use OpenCL terminology. (3p)

(b) If you have a modern GPU with 512 cores, how much speedup can you expect to get? Yes, it depends on the algorithms, but in what way? Make a reasonable assessment and back that with hardware and algorithm based arguments. (2p)

(a) Describe the major architectural differences between a multi-core CPU and a GPU (apart from the GPU being tightly coupled with image output). Focus on the differences that are important for parallel computing. (3p)

(a) Describe three sorting algorithms in terms of their suitability for GPU implementation. Computational complexity should be considered. (3p)

(b) The GPU design in centered around a number of features vital for its primary use, graphics. List three such features, as significant as possible, which are also important for GPU computing and assess their importance. (2p)

(a) In many algorithms, one thread can produce values that affect other threads. Suggest two different ways to make sure that the results are produced without conflicts. The two approaches do not have to be relevant for the same situations. What is the performance impact of each approach? (Only dependencies within the same block are taken into account here.) (3p)

(b) You are given the task of implementing an algorithm that you decide needs to be implemented in a number of blocks, but there are dependencies between the blocks. How can you handle dependencies between different blocks? (2p)

(a) Describe how Bitonic Merge Sort can be implemented on a GPU. A figure to clarify the algorithm is expected. Your solution must be able to handle large data sets (i.e. 100000 items or more). (3p)

(b) Why can coalescing improve performance? How can you take advantage of coalescing for an algorithm with a non-coalesced memory access pattern? (2p)

(a) Motivate why GPUs can give significantly better computing performance than ordinary CPUs. Is there any reason to believe that this advantage will be reduced over time? (2p)

(b) Compare shared memory, global memory, constant memory and register memory in terms of performance, usage and accessibility. CUDA terminology is assumed, please note if you use OpenCL terminology. (3p)

(a) Describe the major architectural differences between a multi-core CPU and a GPU (apart from the GPU being tightly coupled with image output). Focus on the differences that are important for parallel computing. (3p)

(b) Compare shared memory, global memory and register memory in terms of performance, usage and accessibility. CUDA terminology is assumed, please note if you use OpenCL terminology. (2p)

(b) Describe how reduction can be used to calculate the maximum value of a large array of scalar values on a GPU.

Also give at least two examples of other problems that are solved by reduction. (2p)

(b) List three different kinds of GPU memory and describe for each their characteristics in terms of performance, usage and accessibility. CUDA terminology is assumed, please note if you use OpenCL terminology. Constant memory should not be included since that is a separate question below. (2p)

(a) Outline how reduction is implemented in an efficient way, using text and figures. You may assume that the reduction problem in question deals with finding the maximum of a large dataset. Assume that the dataset can be of highly varying size, including very large. (2p)

(b) Three important kinds of GPU memory include shared (local), global and texture memory. Describe these in terms of performance, usage and accessibility. CUDA terminology is assumed, please note if you use OpenCL terminology. (2p)

(b) Why can coalescing improve performance? How can you rewrite an algorithm with non-coalesced memory access patterns to take advantage of coalescing? (2p)

(a) Shared memory is fast temporary storage, but its access times still depends on something. What do you need to do to get the fastest possible shared memory access? (2p)

(b) Explain why is it not possible to synchronize between blocks/work groups. What can you do about it? Give a demonstration of the problem and its solution based on bitonic merge sort. (3p)

(b) Some image filters are separable. A typical case is to split a filter into one horizontal and one vertical filter, often of the same size. This has potential to improve performance. However, the two parts may each run with significantly different performance, one much faster than the other. Suggest a likely reason why this could happen. (2p)

19.4 GPU Quickies

This is my section for “fast points”, both for me and for the students. One point each, and the answer is typically just a single line.

memory

(a) In what way(s) is a texturing unit more than just another memory? (1p)

(c) Texture memory provides interpolation in hardware. Why is this a questionable feature to rely on? (1p)

(c) How can pinned (page-locked) CPU memory improve performance? (1p)

(c) How can a non-coalesced memory access be converted to a coalesced memory access? (1p)

(e) When can constant memory give a performance advantage? (1p)

(b) Why can using constant memory improve performance?

(c) What kind of algorithms benefit from using constant memory? (1p)

(b) In what way(s) is a texturing unit more than just another memory? (1p)

(a) What do you have to do for achieving better performance by coalescing? (1p)

(b) Constant memory is fast under a certain condition. Which condition is that? (1p)

(c) Texture access provides two unique features that we otherwise do not have. Name one, and describe with a brief sentence. (1p)

thread management

(a) Imagine a CUDA programmer who uses the practice to always use as big block size as possible. Why will this not always result in the highest possible performance? (1p)

(e) Can you rely on any threads/work groups in a GPU computation to be literally executed in parallel? If so, which ones? (1p)

(e) When do you typically need to synchronize threads? (1p)

(e) Why is load balancing often not a (big) problem in GPU computing, e.g. when computing fractals? (1p)

history and development

(b) List and briefly explain (short comments of a few words) the importance of two major features of the Fermi architecture that were not available in earlier GPUs. (1p)

(e) Explain why the G80 architecture had significantly higher performance than earlier GPUs. (1p)

(a) Why is the G80 so much faster than the previous GPUs (e.g. 7000 series)

(d) Suggest one important feature in GPUs that was added for performing some specific graphics effect. Name the effect too. (1p)

(a) GPUs have evolved around the needs of graphics applications. Give an example feature, apart from multiple threads, that was added for the needs of graphics which is valuable for GPU computing. (1p)

shaders

(e) For what kind of problems are shader-based GPU computing most suitable? Give one specific example. (1p)

(c) What kind of shaders is most interesting for GPU computing? (What part of the pipeline?)

(a) In graphics, data is always input as geometrical shapes. What geometry is usually used for fragment shader based GPU computing?

(d) In GPU Computing using the graphics pipeline, in what stage are the computations usually carried out? (1p)

(c) If you want to process a large array in fragment shader based computing, how will that data typically be represented (stored in memory)? (1p)

platforms

(b) Translate the following CUDA concepts to corresponding concepts in OpenCL:(1p)

i. shared memory

ii. block

iii. thread

(d) Give one argument each in favor of using

i. CUDA

ii. OpenCL

iii. GLSL

for a general computing task (which can benefit from a parallel implementation).

(e) List three different kinds of hardware that OpenCL runs on. (Similar systems by different vendors count as one.)

(b) What does a Streaming Multiprocessor correspond to in CUDA and OpenCL, respectively? (1p)

(c) With shader-based GPGPU computing, suggest one limitation that prevents it from performing as well as CUDA and OpenCL. (This may apply to certain GPU generations, not necessarily the latest.) (1p)

(b) What concept in CUDA corresponds to a streaming multiprocessor (SM) in the GPU architecture? (1p)

(d) List three different kinds of hardware that OpenCL runs on. (Similar systems by different vendors count as one.) (1p)

(e) Compare OpenCL and Compute Shaders in terms of portability. You should know at least one strong point of each. (1p)

(e) State one advantage with CUDA/OpenCL over fragment shader based GPU computing. (1p)

misc

(a) Describe how multiple CUDA streams can be used to accelerate a computation. (1p)

(a) Where does GPU computing fit in Flynn's taxonomy? What name(s) does the architecture type have according to Flynn's taxonomy? (1p)

(d) Some operations can be implemented either as scatter or gather operations. Which is most suitable for parallel implementation (on GPUs in particular)? Why? (1p)

(b) What particular algorithm feature makes bitonic merge sort particularly suitable for parallel implementation? (1p)

(d) Some operations, like image filters, can be implemented using scatter or gather algorithms. If you use scatter, what specific operation must be used to make it work correctly? (1p)

(a) In CUDA, you can use the modifiers __global__ and __device__. What is the difference between them? (1p)

(a) In CUDA, you can use the modifier __host__. What does this signify? (1p)

(a) In CUDA, you can use the modifier __global__. What does this signify? (1p)

(a) In CUDA, you can use the modifier __device__. What does this signify? (1p)

20. Final words

For a long time, I thought I would never write this book. Other books appeared, the university disencourages book authoring and considers it not qualifying and implies that it is done to rip off money off the students.

That is not the purpose with any of my books. My books are written with the only purpose to improve the courses with fitting course material, at low or even no cost. I provide my books as low-cost paperbacks as well as on-line digital versions, the latter for free.

One important help in making this book, and keeping the cost low, is the new on-line book production facilities, like CreateSpace and Publit. I can now produce a good-looking book much easier than before, for a lower cost, and perfect control over its appearance. With those tools, authoring felt easy and fun, and I got new inspiration to write this. I hope you enjoy it!

I went up one morning,

sat on the porch in the dawn sun

and pondered about some idea.

I took up my notebook and scribbled down my thoughts.

Then I read what I had written

and I saw, that there was a thought in it

and only then I knew for sure

that I exist.

21. References

[1] I. Ragnemalm, “Polygons feel no pain”, 2008/2017.

[2] Erik Pettersson, “Signal- och bildbehandling på moderna grafikprocessorer”, LiTH-ISY-EX--05/3761--SE, 2005.

[3] NVidia GPU Programming Guide, http://developer.download.nvidia.com/GPU_Programming_Guide/GPU_Programming_Guide.pdf

[4] A. Blackert, “Evaluation of Multi-Threading in Vulkan”, thesis performed 2016.

[5] colfaresearch.com/xeon-2017 (retrieved 2018-02-26)

[6] www.nvidia.com/sv-se/titan-v/#specs (retrieved 2018-02-26)

[7] Kenneth E. Hoff III, Tim Culver, John Keyser, Ming Lin and Dinesh Manocha, “Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware”, in Computer Graphics, SIGGRAPH Annual Conference Proceedings, ACM, 1999.

[8] E. Scott Larsen, David McAllister, “Fast Matrix Multiplies using Graphics Hardware”, in Proceeedings, Supercomputing 2001.

[9] David M. W. Powers, “Parallelized QuickSort with Optimal Speedup”, Proceedings of International Conference on Parallel Computing Technologies, Novosibirsk, 1991.

[10] Naga K. Govindaraju, Nikunj Raghuvanshi, Michael Henson, David Tuft, Dinesh Manocha, “A Cache-Efficient Sorting Algorithm for Database and Data Mining Computations using Graphics Processors”, technical report, University of North Carolina, Chapel Hill, 2005.

[11] Daniel Cederman, Philippas Tsigas, “GPU-Quicksort: A Practical Quicksort Algorithm for Graphics Processors”,

[12] Erik Sintorn, Ulf Assarsson, “Fast parallel GPU-sorting using a hybrid algorithm”, Journal, of Parallel Distributed Computing 68, pp 1381-1388, October 2008

[13] Jörgen Ahlberg, “Model-based Coding”, PhD thesis, Linköping University, 2002.

[14] Introduction to Data-Oriented Design, DICE, www.dice.se/wp-content/uploads/2014/12/Introduction_to_Data-Oriented_Design.pdf (retrieved 2018-03-02)

[15] Mark Harris, “How To Access Global Memory Efficiently in CUDA C/C++ Kernels”, NVIDIA Developer Blog, devblogs.nvidia.com/how-access-global-memory-efficiently-cuda-c-kernels (retrieved 2018-03-05)

[16] CUDA C Best Practices Guide 2018, docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html (retrieved 2018-03-05)

[17] Mark Harris, “Optimizing parallel reduction in CUDA”, developer.download.nvidia.com/assets/cuda/files/reduction.pdf (retrieved 2018-03-26)

[18] Marco Fratarcangeli, “Cloth simulation using GLSL, OpenCL and CUDA”, Game engine gems 2, 2011.

[19] Torbjörn Sörman, “Comparison of Technologies for General Purpose Computing on Graphics Processing Units ”, LiTH-ISY-EX--16/4923--SE, 2016.

[20] Bitonic sort implementation by Nikos Pitsianis, www2.cs.duke.edu/courses/fall08/cps196.1/Pthreads/bitonic.c (retrieved 2018-05-27)

[21] Ken Batcher, “Sorting networks and their applications”, Spring Joint Computer Conference, 1968, pp 307-314.

[22] Maxim Shevtsov, “OpenCL and OpenGL Interoperability Tutorial”, software.intel.com/en-us/articles/opencl-and-opengl-interoperability-tutorial (ret 2018-05-03)

[23] Mark Harris, “Parallel Prefix Sum (Scan) with CUDA”, NVidia, 2007 (retrieved spring 2018)

[24] Guy E. Blelloch, “Prefix Sums and Their Applications”, in J.H.Reif (ed), *Synthesis of Parallel Algorithms*, Morgan Kaufmann, 1990.

[25] Adam Lake, “Getting the most from OpenCL 1.2: How to increase performance by minimizing buffer copies on Intel Processor Graphics”, Intel, 2014 (retrieved 2018)

[26] Mark Harris, “How To Overlap Data Transfers in CUDA C/C++”, NVIDIA Developer Blog, devblogs.nvidia.com/how-overlap-data-transfers-cuda-cc (retrieved 2018)

[27] Sanders, J., Kandrot, E, “CUDA by example”, Addison-Wesley, 2010

22. Index

D

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X