J. Parallel Distrib. Comput. 73 (2013) 4–13

Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Graphics processing unit (GPU) programming strategies and trends in GPU computing

André R. Brodtkorb[a,*], Trond R. Hagen[a,b], Martin L. Sætra[b]

[a] SINTEF ICT, Department of Applied Mathematics, P.O. Box 124, Blindern, NO-0314 Oslo, Norway

[b] Centre of Mathematics for Applications, University of Oslo, P.O. Box 1053 Blindern, NO-0316 Oslo, Norway

ARTICLE INFO

Article history:
Received 19 December 2011
Received in revised form
28 March 2012
Accepted 16 April 2012
Available online 4 May 2012

Keywords:
GPU computing
Heterogeneous computing
Profiling
Optimization
Debugging
Hardware
Future trends

ABSTRACT

Over the last decade, there has been a growing interest in the use of graphics processing units (GPUs) for non-graphics applications. From early academic proof-of-concept papers around the year 2000, the use of GPUs has now matured to a point where there are countless industrial applications. Together with the expanding use of GPUs, we have also seen a tremendous development in the programming languages and tools, and getting started programming GPUs has never been easier. However, whilst getting started with GPU programming can be simple, being able to fully utilize GPU hardware is an art that can take months or years to master. The aim of this article is to simplify this process, by giving an overview of current GPU programming strategies, profile-driven development, and an outlook to future trends.

1. Introduction

Graphics processing units (GPUs) have for well over a decade been used for general-purpose computation, called GPU computing or GPGPU [8]. When the first GPU programs were written, the GPU was used much like a calculator: it had a set of fixed operations that were exploited to achieve some desired result. As the GPU is designed to output a two-dimensional (2D) image from a three-dimensional (3D) virtual world, the operations it could perform were fundamentally linked with graphics, and the first GPU programs were expressed as operations on graphical primitives such as triangles. These programs were difficult to develop, debug, and optimize, and compiler bugs were frequently encountered. However, the proof-of-concept programs demonstrated that the use of GPUs could give dramatic speedups over central processing units (CPUs) for certain algorithms [20,4], and research on GPUs soon led to the development of higher-level third-party languages that abstracted away the graphics. These languages, however, were rapidly abandoned when the hardware vendors released dedicated non-graphics languages that enabled the use of GPUs for general-purpose computing (see Fig. 1).

The key to the success of GPU computing has partly been its massive performance when compared to the CPU: today, there is a performance gap of roughly seven times between the two when comparing theoretical peak bandwidth and gigaflops performance (see Fig. 2). This performance gap has its roots in physical per-core restraints and architectural differences between the two processors. The CPU is in essence a serial *von Neumann* processor, and is highly optimized to execute a series of operations in order. One of the major performance factors of CPUs has traditionally been its steadily increasing frequency. If you double the frequency, you double the performance, and there has been a long-standing trend of exponential frequency growth. In the 2000s, however, this increase came to an abrupt stop as we hit the *power wall* [3]: because the power consumption of a CPU is proportional to the frequency cubed [4], the power density was approaching that of a nuclear reactor core [22]. Unable to cool such chips sufficiently, the trend of exponential frequency growth stopped at just below 4.0 GHz. Coupled with the *memory wall* and the *ILP*[1] *wall*, serial computing had reached its zenith in performance [3], and CPUs started increasing performance through multicore and vector instructions instead.

At the same time as CPUs hit the serial performance ceiling, GPUs were growing exponentially in performance due to *massive*

[*] Corresponding author.
E-mail addresses: Andre.Brodtkorb@sintef.no (A.R. Brodtkorb), TrondRunar.Hagen@sintef.no (T.R. Hagen), Martin.L.Satra@cma.uio.no (M.L. Sætra).

[1] ILP stands for instruction level parallelism.

0743-7315/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2012.04.003

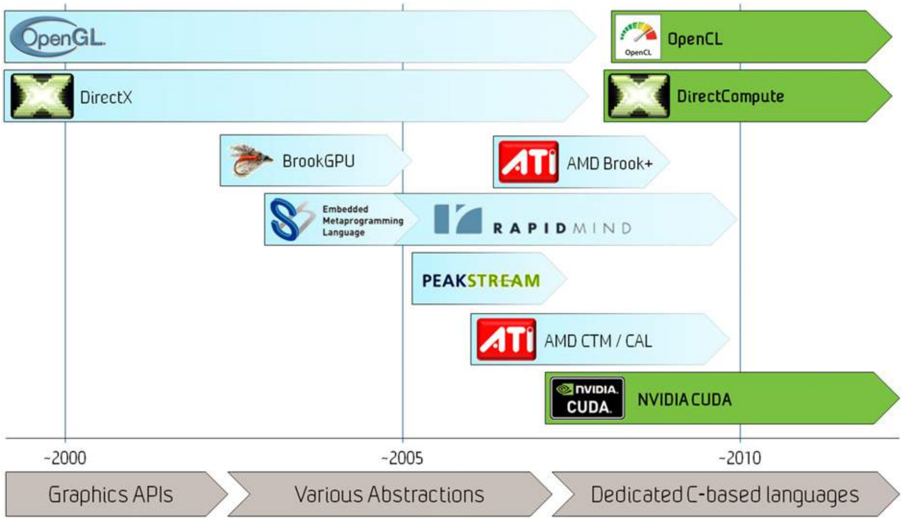


Fig. 1. History of programming languages for GPU computing. When GPUs were first being used for non-graphics applications, one had to rewrite the application in terms of operations on graphical primitives using languages such as OpenGL or DirectX. As this was cumbersome and error prone, several academic and third-party languages that abstracted away the graphics appeared. Since 2007, however, vendors started releasing general-purpose languages for programming the GPU, such as AMD Close-to-Metal (CTM) and NVIDIA CUDA. Today, the predominant general-purpose languages are NVIDIA CUDA, DirectCompute, and OpenCL.

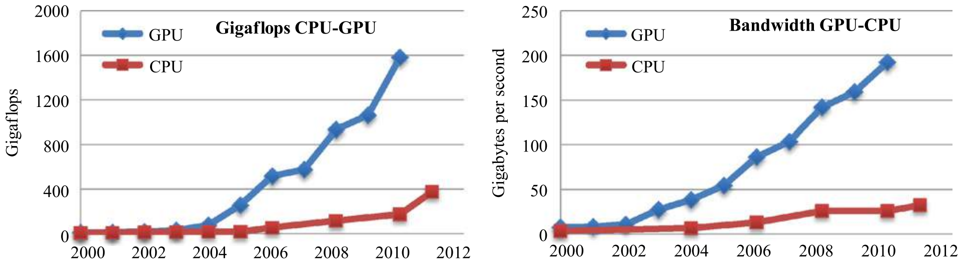


Fig. 2. Historical comparison of theoretical peak performance in terms of gigaflops and bandwidth for the fastest available NVIDIA GPUs and Intel CPUs. Today, the performance gap is currently roughly seven times for both metrics.

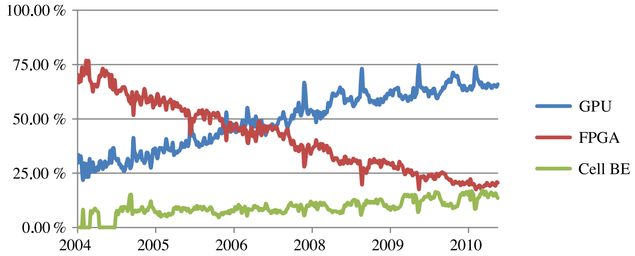


Fig. 3. Google search trends for GPU, FPGA, and Cell BE. These terms were chosen as the highest-ranking keywords for each of the architectures. Whilst both FPGAs and the Cell BE today receive a modest number of searches, the GPU has a large growing interest.

parallelism: as computing the color of a pixel on screen can be performed independently of all other pixels, parallelism is a natural way of increasing performance in GPUs. Parallelism appears to be a sustainable way of increasing performance, and there are many applications that display embarrassingly parallel workloads that are perfectly suited for GPUs. However, increased parallelism will only increase the performance of parallel code sections, meaning that the serial part of the code soon becomes the bottleneck. This is often referred to as Amdahl's law [2]. Thus, most applications benefit from the powerful combination of a massively parallel GPU and a fast multicore CPU. This combination is known as a heterogeneous computer: the combination of traditional CPU cores and specialized accelerator cores [4].

GPUs are not the only type of accelerator core that has gained interest over the last decade. Other examples include field-programmable gate arrays (FPGAs) and the Cell Broadband Engine (Cell BE), which have both been highly successful in many application areas [4]. Today, however, these receive only a fraction of the attention of that given to GPUs (see Fig. 3). One of the major reasons for this is that a very large percentage of desktop and laptop computers have a dedicated GPU already, whilst FPGAs and the Cell BE are only found in specially ordered setups. Furthermore, the future of the Cell BE is currently uncertain, as the road map for the second version has not been followed through. FPGAs, on the other hand, have a thriving community in the embedded markets, but are unfortunately too hard to program for general-purpose computing. The cumbersome and difficult programming process that requires detailed knowledge of low-level hardware has recently improved dramatically with the advent of C-like languages, but the development cycle is still slow due to the time-consuming place and route stage.

There are three major GPU vendors for the PC market today, Intel being the largest. However, Intel is only dominant in the integrated and low-performance market. For high-performance and discrete graphics, AMD and NVIDIA are the sole two suppliers. In academic and industrial environments, NVIDIA appears to be the clear predominant supplier, and we thus focus on GPUs from NVIDIA in this article, even though most of the concepts and techniques are also directly transferable to GPUs from AMD. For NVIDIA GPUs, there are three languages suitable for general-purpose computing, as shown in Fig. 1. Of these, we focus on NVIDIA CUDA. Even though the three languages are conceptually equivalent and offer more or less the same functionality, CUDA is the most mature technology with the most advanced development tools.

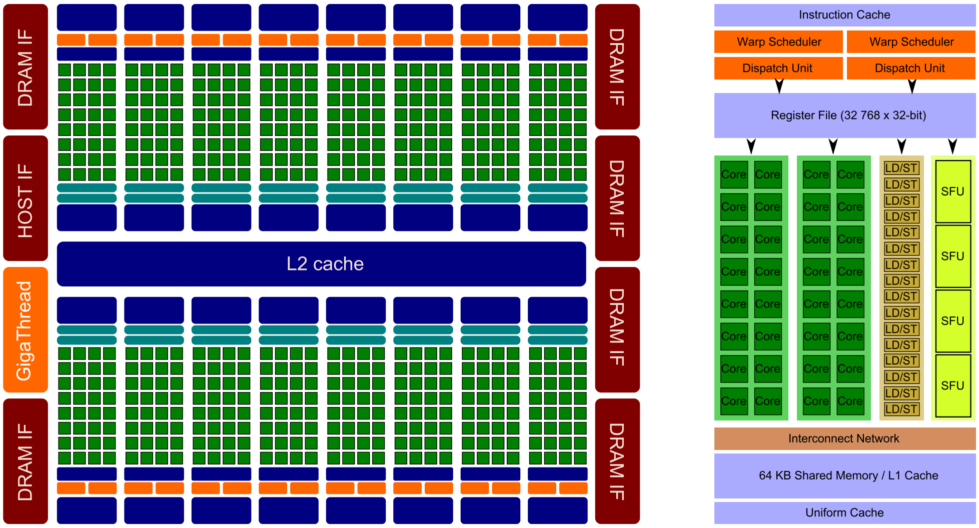


Fig. 4. Current Fermi-class GPU hardware. The GPU consisting of up to 16 streaming multiprocessors (also known as SMs) is shown in (left), and (right) shows a single multiprocessor.

Much of the information in this article can be found in a variety of different sources, including books, documentation, manuals, conference presentations, and on Internet fora. Getting an overview of all this information is an arduous exercise that requires a substantial effort. The aim of this article is therefore to give an overview of state-of-the-art programming techniques and profile-driven development, and to serve as a step-by-step guide for optimizing GPU codes. The rest of the article is sectioned as follows. First, we give a short overview of current GPU hardware in Section 2, followed by general programming strategies in Section 3. Then, we give a thorough overview of profile-driven development for GPUs in Section 4, and a short overview of available debugging tools in Section 5. Finally, we offer our view on the current and future trends in Section 6, and conclude with a short summary in Section 7.

2. Fermi GPU architecture

The multicore CPU is composed of a handful of complex cores with large caches. The cores are optimized for single-threaded performance and can handle up to two hardware threads per core using hyper-threading. This means that a lot of transistor space is dedicated to complex instruction-level parallelism such as instruction pipelining, branch prediction, speculative execution, and out-of-order execution, leaving only a tiny fraction of the die area for integer and floating-point execution units. In contrast, a GPU is composed of hundreds of simpler cores that can handle thousands of concurrent hardware threads. GPUs are designed to maximize floating-point throughput, whereby most transistors within each core are dedicated to computation rather than complex instruction-level parallelism and large caches. The following part of this section gives a short overview of a modern GPU architecture.

Today's Fermi-based architecture [17] feature up to 512 accelerator cores called CUDA cores (see Fig. 4a). Each CUDA core has a fully pipelined integer arithmetic logic unit (ALU) and a floating-point unit (FPU) that executes one integer or floating-point instruction per clock cycle. The CUDA cores are organized in 16 streaming multiprocessors, each with 32 CUDA cores (see Fig. 4b). Fermi also includes a coherent L2 cache of 768 KB that is shared across all 16 multiprocessors in the GPU, and the GPU has a 384-bit GDDR5 DRAM memory interface supporting up to a total of 6 GB of on-board memory. A host interface connects the GPU to the CPU via the PCI Express bus. The GigaThread global scheduler distributes thread blocks to multiprocessor thread schedulers (see Fig. 4a). This scheduler handles concurrent *kernel*[2] execution and out-of-order thread block execution.

Each multiprocessor has 16 load/store units, allowing source and destination addresses to be calculated for 16 threads per clock cycle. Special function units (SFUs) execute intrinsic instructions such as sine, cosine, square root, and interpolation. Each SFU executes one instruction per thread, per clock. The multiprocessor schedules threads in groups of 32 parallel threads called *warps*. Each multiprocessor features two warp schedulers and two instruction dispatch units, allowing two warps to be issued and executed concurrently. The Fermi dual warp scheduler selects two warps, and issues one instruction from each warp to a group of 16 CUDA cores, 16 load/store units, or 4 SFUs. The multiprocessor has 64 KB of on-chip memory that can be configured as 48 KB of *shared memory* with 16 KB of L1 cache or as 16 KB of shared memory with 48 KB of L1 cache.

A traditional critique of GPUs has been their lack of IEEE-compliant floating-point operations and error-correcting code (ECC) memory. However, these shortcomings have been addressed by NVIDIA, and all of their recent GPUs offer fully IEEE-754-compliant single-precision and double-precision floating-point operations, in addition to ECC memory.

3. GPU programming strategies

Programming GPUs is unlike traditional CPU programming, because the hardware is dramatically different. It can often be a relatively simple task to get started with GPU programming and get speedups over existing CPU codes, but these first attempts at GPU computing are often suboptimal, and do not utilize the hardware to a satisfactory degree. Achieving a scalable high-performance code that uses hardware resources efficiently is still a difficult task that can take months or years to master.

3.1. Guidelines for latency hiding and thread performance

The GPU execution model is based around the concept of launching a kernel on a grid consisting of blocks (see Fig. 5). Each block again consists of a set of threads, and threads within

[2] A kernel is a GPU program that typically executes in a data-parallel fashion.

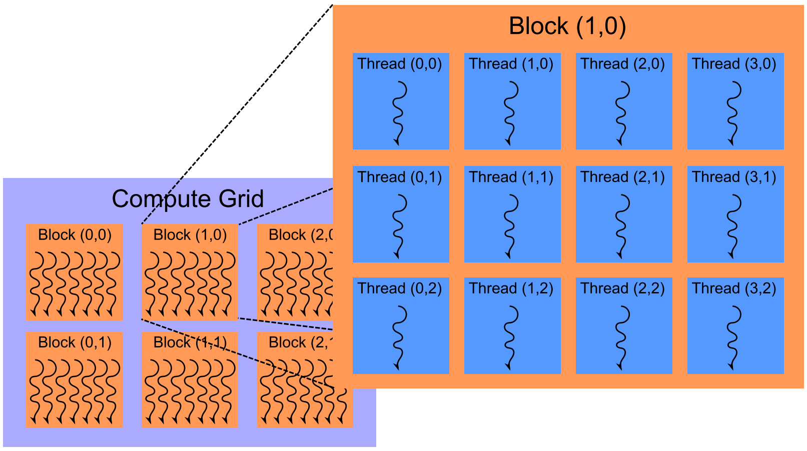
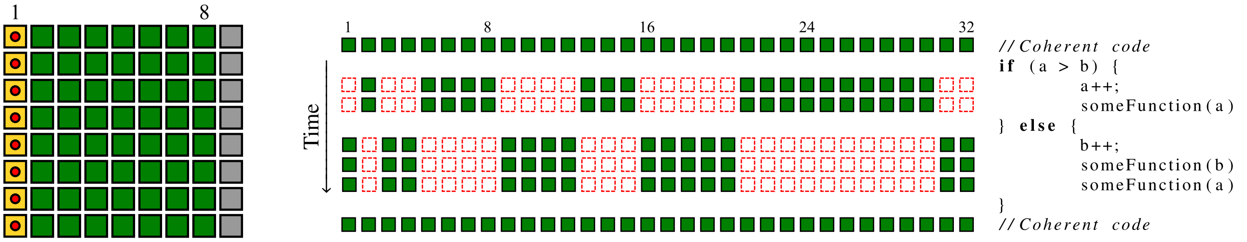


Fig. 5. The CUDA concept of a grid of blocks. Each block consists of a set of threads that can communicate and cooperate. Each thread uses its block index in combination with its thread index to identify its position in the global grid.



```
//Coherent code
if (a > b) {
        a++;
        someFunction(a)
} else {
        b++;
        someFunction(b)
        someFunction(a)
}
//Coherent code
```

Fig. 6. Bank conflicts and thread divergence. (left) Conflict-free column-wise access of shared memory. The example shows how padding shared memory to be the number of banks plus one gives conflict-free access by columns (marked with circles for the first column). Without padding, all elements in the first column would belong to the same bank, and thus give an eight-way bank conflict. By padding the width by one, we ensure that the elements belong to different banks. Please note that the constructed example shows eight banks, whilst current hardware has 32 banks. (right) Branching on 32-wide SIMD GPU architectures. All threads perform the same computations, but the result is masked out for the dashed boxes.

the same block can synchronize and cooperate using fast shared memory. This maps to the hardware so that a block runs on a single multiprocessor, and one multiprocessor can execute multiple blocks in a time-sliced fashion. The grid and block dimensions can be one, two, or three dimensional, and they determine the number of threads that will be used. Each thread has a unique identifier within its block, and each block has a unique global identifier. These are combined to create a unique global identifier per thread.

The massively threaded architecture of the GPU is used to hide memory latencies. Even though the GPU has a vastly superior memory bandwidth compared to CPUs, it still takes on the order of hundreds of clock cycles to start the fetch of a single element from main GPU memory. This latency is automatically hidden by the GPU through rapid switching between threads. Once a thread stalls on a memory fetch, the GPU instantly switches to the next available thread in a fashion similar to hyper-threading [14] on Intel CPUs. This strategy, however, is most efficient when there are enough available threads to completely hide the memory latency, meaning that we need a lot of threads. As there is a maximum number of concurrent threads a GPU can support, we can calculate the percentage of this figure we are using. This number is referred to as the *occupancy*, and as a rule of thumb it is good to keep a relatively high occupancy. However, a higher occupancy does not necessarily equate to higher performance: once all memory latencies are hidden, a higher occupancy may actually degrade the performance as it also affects other performance metrics.

Hardware threads are available on Intel CPUs as hyper-threading, but a GPU thread operates quite differently from these CPU threads. One of the things that differs from traditional CPU programming is that the GPU executes instructions in a 32-way SIMD[3] fashion, in which the same instruction is simultaneously executed for 32 different data elements, called a warp. This is illustrated in Fig. 6b, in which a branch is taken by only some of the threads within a warp. This means that all threads within a warp must execute both parts of the branch, which in the utmost consequence slows down the program by a factor 32. Conversely, this does not affect performance when all threads in a warp take the same branch.

One technique used to avoid expensive branching within a kernel is to sort the elements according to the branch, and thus make sure the threads within each warp all execute their code without branching. Another way of preventing branching is to perform the branch once on the CPU instead of for each warp on the GPU. This can be done for example using templates: by replacing the branch variable with a template variable, we can generate two kernels, one for condition true, and one for condition false, and let the CPU perform the branch and from this select the correct kernel. The use of templates, however, is not particularly powerful in this example, as the overhead of running a simple coherent if statement in the kernel would be small. But when there are a lot of parameters, there can be a large performance gain from using template kernels [7,5]. Another prime example of the benefit of template kernels is the ability to specify different shared memory sizes at compile time, thus allowing the compiler to issue warnings for out-of-bounds access. The use of templates can also be used to perform compile-time loop unrolling, which has a great performance impact. By using a switch-case statement, with a separate kernel being launched for different for-loop sizes, the performance can be greatly improved.

[3] SIMD stands for single instruction multiple data.

3.2. Memory guidelines

CPUs have struggled with the memory wall for a long time. The memory wall, in which transferring data to the processor is far more expensive than computing on that data, can also be a problem on GPUs. This means that many algorithms will often be memory bound, making memory optimizations important. The first lesson in memory optimization is to reuse data and keep it in the fastest available memory. For GPUs, there are three memory areas, listed in decreasing order by speed: registers, shared memory, and global memory.

Registers are the fastest memory units on a GPU, and each multiprocessor on the GPU has a large, but limited, register file which is divided amongst threads residing on that multiprocessor. Registers are private for each thread, and if the threads use more registers than are physically available, registers will also spill to the L1 cache and global memory. This means that, when you have a high number of threads, the number of registers available to each thread is very restricted, which is one of the reasons why a high occupancy may actually hurt performance. Thus, thread-level parallelism is not the only way of increasing performance. It is also possible to increase performance by decreasing the occupancy to increase the number of registers available per thread.

The second fastest memory type is *shared memory*, and this memory can be just as fast as registers if accessed properly. Shared memory is a very powerful tool in GPU computing, and the main difference between registers and shared memory is the ability for several threads to share data. Shared memory is accessible to all threads within one block, thus enabling cooperation. It can be thought of as a kind of programmable cache, or scratchpad, in which the programmer is responsible for placing often-used data there explicitly. However, as with caches, its size is limited (up to 48 KB) and this can often be a limitation on the number of threads per block. Shared memory is physically organized into 32 banks that serve one warp with data simultaneously. However, for full speed, each thread must access a distinct bank. Failure to do so leads to more memory requests, one for each *bank conflict*. A classical way to avoid bank conflicts is to use padding. In Fig. 6a, for example, we can avoid bank conflicts for column-wise access by padding the shared memory with an extra element, so that neighboring elements in the same column belong to different banks.

The third, and slowest type of memory on the GPU is the *global memory*, which is the main memory of the GPU. Even though it has an impressive bandwidth, it has a high latency, as discussed earlier. These latencies are preferably hidden by a large number of threads, but there are still large pitfalls. First of all, just as with CPUs, the GPU transfers full cache lines across the bus (called coalesced reads). As a rule of thumb, transferring a single element consumes the same bandwidth as transferring a full cache line. Thus, to achieve full memory bandwidth, we should program the kernel such that warps access continuous regions of memory. Furthermore, we want to using full cache lines, which is done by starting at a quad word boundary (the start address of a cache line), and using full quadwords (128 bytes) as the smallest unit. This address alignment is typically achieved by padding arrays. Alternatively, for non-cached loads, it is sufficient to align to word boundaries and transfer words (32 bytes). To fully occupy the memory bus, the GPU also uses memory parallelism, in which a large number of outstanding memory requests are used to occupy the bandwidth. This is both a reason for a high memory latency, and a reason for high bandwidth utilization.

Fermi also has hardware L1 and L2 caches that work in a similar fashion as traditional CPU caches. The L2 cache size is fixed and shared between all multiprocessors on the GPU, whilst the L1 cache is per multiprocessor. The L1 cache can be configured to be either 16 or 48 KB, at the expense of shared memory. The L2 cache, on the other hand, can be turned on or off on at compile time, or by using inline PTX assembly instructions in the kernel. The benefit of turning off the L2 cache is that the GPU is now allowed to transfer smaller amounts of data than a full cache line, which will often improve performance for sparse and other random access algorithms.

In addition to the L1 and L2 caches, the GPU also has dedicated caches that are related to traditional graphics functions. The constant-memory cache is one example, which in CUDA is typically used for arguments sent to a GPU kernel. It has its own dedicated cache tailored for broadcast, in which all threads in a block access the same data. The GPU also has a texture cache that can be used to accelerate reading global memory. However, the L1 cache has a higher bandwidth, so the texture cache is most useful if combined with texture functions such as linear interpolation between elements.

3.3. Further guidelines

The CPU and the GPU are different processors that operate asynchronously. This means that we can let the CPU and the GPU perform different tasks simultaneously, which is a key ingredient of heterogeneous computing: the efficient use of multiple different computational resources, in which each resource performs the tasks for which it is best suited. In the CUDA API, this is exposed as *streams*. Each stream is an in-order queue of operations that will be performed by the GPU, including memory transfers and kernel launches. A typical use case is that the CPU schedules a memory copy from the CPU to the GPU, a kernel launch, and a copy of results from the GPU to the CPU. The CPU then continues to perform CPU-side calculations simultaneously as the GPU processes its operations, and only synchronizes with the GPU when its results are needed. There is also support for independent streams, which can execute their operations simultaneously as long as they obey their own streams order. Current GPUs support up to 16 concurrent kernel launches [18], which means that we can have both data parallelism, in terms of a computational grid of blocks, and task parallelism, in terms of different concurrent kernels. GPUs furthermore support overlapping memory copies between the CPU and the GPU and kernel execution. This means that we can simultaneously copy data from the CPU to the GPU, execute 16 different kernels, and copy data from the GPU back to the CPU if all these operations are scheduled properly to different streams.

When transferring data between the CPU and the GPU over the PCI Express bus, it is beneficial to use so-called page-locked memory. This essentially disables the operating system from paging memory, meaning that the memory area is guaranteed to be continuous and in physical RAM (not swapped out to disk, for example). However, page-locked memory is scarce, and is rapidly exhausted if used carelessly. A further optimization for page-locked memory is to use write-combining allocation. This disables CPU caching of a memory area that the CPU will only write to, and can increases the bandwidth utilization by up to 40% [18]. It should also be noted that enabling ECC memory will negatively affect both the bandwidth utilization and available memory, as ECC requires extra bits for error control.

CUDA now also supports a *unified address space*, in which the physical location of a pointer is automatically determined. That is, data can be copied from the GPU to the CPU (or the other way round) without specifying the direction of the copy. While this might not seem like a great benefit at first, it greatly simplifies code needed to copy data between CPU and GPU memories, and enables advanced memory accesses. The unified memory space is particularly powerful when combined with mapped memory.

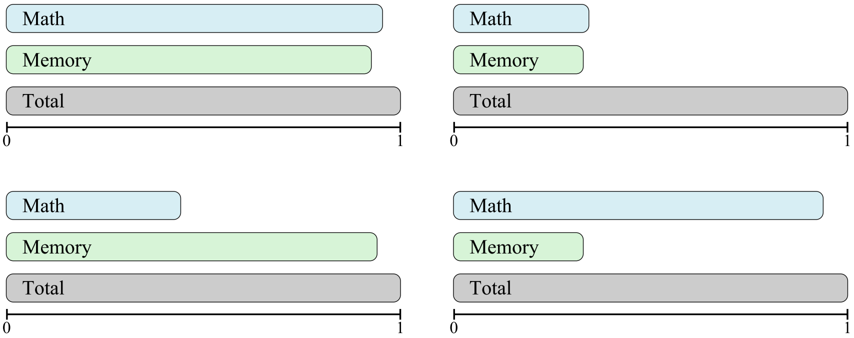


Fig. 7. Normalized run time of modified kernels which are used to identify bottlenecks: (top left) a well-balanced kernel, (top right) a latency-bound kernel, (bottom left) a memory-bound kernel, and (bottom right) an arithmetic-bound kernel. "Total" refers to the total kernel time, whilst "Memory" refers to a kernel stripped of arithmetic operations, and "Math" refers to a kernel stripped of memory operations. It is important to note that latencies are part of the measured run times for all kernel versions.

A mapped memory area is a continuous block of memory that is available directly from both the CPU and the GPU simultaneously. When using mapped memory, data transfers between the CPU and the GPU are executed asynchronously with kernel execution automatically.

The most recent CUDA version has also become threadsafe [18], so that one CPU thread can control multiple CUDA contexts (e.g., one for each physical GPU), and conversely multiple CPU threads can share control of one CUDA context. The unified memory model together with the new threadsafe context handling enables much faster transfers between multiple GPUs. The CPU thread can simply issue a direct GPU–GPU copy, bypassing a superfluous copy to CPU memory.

4. Profile-driven development[4]

A famous quote attributed to Donald Knuth is that "premature optimization is the root of all evil" [11], or, put another way, make sure that the code produces the correct results before trying to optimize it, and optimize only where it will make an impact. The first step in optimization is always to identify the major application bottlenecks, as performance will increase the most when removing these. However, locating the bottleneck is hard enough on a CPU, and can be even more difficult on a GPU. Optimization should also be considered a cyclic process, meaning that, after having found and removed one bottleneck, we need to repeat the profiling process to find the next bottleneck in the application. This cyclic optimization can be repeated until the kernel operates close to the theoretical hardware limits or all optimization techniques have been exhausted.

To identify the performance bottleneck in a GPU application, it is important to chose the appropriate performance metrics, and then compare the measured performance to the theoretical peak performance. There are several bottlenecks one can encounter when programming GPUs. For a GPU kernel, there are three main bottlenecks: the kernel may be limited by instruction throughput, memory throughput, or latencies. However, it might also be that CPU–GPU communication is the bottleneck, or that application overheads dominate the run time.

4.1. Locating kernel bottlenecks

There are two main approaches to locating the performance bottleneck of a CUDA kernel, the first and most obvious being to use the CUDA profiler. The profiler is a program that samples different hardware counters, and the correct interpretation of these numbers is required to identify bottlenecks. The second option is to modify the source code, and compare the execution time of the differently modified kernels.

The profiler can be used to identify whether a kernel is limited by bandwidth or arithmetic operations. This is done by simply looking at the instruction-to-byte ratio, or in other words finding out how many arithmetic operations your kernel performs per byte it reads. The ratio can be found by comparing the instructions issued counter (multiplied with the warp size, 32) to the sum of global store transactions and L1 global load miss counters (both multiplied with the cache line size, 128 bytes), or directly through the `instruction/byte` counter. Then we compare this ratio to the theoretical ratio for the specific hardware the kernel is running on, which is available in the profiler as the `ideal instruction/byte ratio` counter.

The profiler does not always report accurate figures, because the number of load and store instructions may be lower than the actual number of memory transactions, depending on address patterns and individual transfer sizes. To get the most accurate figures, we can follow another strategy which is to compare the run time of three modified versions of the kernel: the original kernel, one *math* version in which all memory loads and stores are removed, and one *memory* version in which all arithmetic operations are removed (see Fig. 7). If the *math* version is significantly faster than the original and *memory* kernels, we know that the kernel is memory bound, and conversely for arithmetic. This method has the added benefit of showing how well memory operations and arithmetic operations overlap.

To create the *math* kernel, we simply comment out all load operations, and move every store operation inside conditionals that will always evaluate to false. We do this to fool the compiler so that it does not optimize away the parts we want to profile, since the compiler will strip away all code not contributing to the final output to global memory. However, to make sure that the compiler does not move the computations inside the conditional as well, the result of the computations must also be used in the condition, as shown in Listing 1. Creating the *memory* kernel, on the other hand, is much simpler. Here, we can simply comment out all arithmetic operations, and instead add all data used by the kernel, and write out the sum as the result.

Note that, if control flow or addressing is dependent on data in memory, the method becomes less straightforward and requires special care. A further issue with modifying the source code is that the register count can change, which again can increase the occupancy and thereby invalidate the measured run time. This, however, can be is solved by increasing the shared memory parameter in the launch configuration of the kernel,

```
someKernel<<<grid_size, block_size,
  shared_mem_size, ...>>>(...),
```

[4] Many of the optimization techniques presented in this section are from the excellent presentations by Paulius Micikevicius [16,15].

```
__global__ void main(..., int flag) {
          float result = ...;
          if(1.0f == result * flag)
                 output[i] = value;
}
```

Listing 1: Compiler trick for arithmetic only kernel. By adding the kernel argument *flag* (which we always set to 0), we disable the compiler from optimizing away the if-statement, and simultaneously disable the global store operation.

until the occupancy of the unmodified version is matched. The occupancy can easily be examined using the profiler or the CUDA Occupancy Calculator.

If a kernel appears to be well balanced (i.e., neither memory nor arithmetics appears to be the bottleneck), we must still check whether or not our kernel operates close to the theoretical performance numbers since it can suffer from latencies. These latencies are typically caused by problematic data dependencies or the inherent latencies of arithmetic operations. Thus, if your kernel is well balanced, but operates at only a fraction of the theoretical peak, it is probably bound by latencies. In this case, a reorganization of memory requests and arithmetic operations is often required. The goal should be to have many outstanding memory requests that can overlap with arithmetic operations.

4.2. Profiling and optimizing memory

Let us assume that we have identified the major bottleneck of the kernel to be memory transactions. The first and least time-consuming thing to try for memory-bound kernels is to experiment with the settings for caching and non-caching loads and the size of the L1 cache to find the best settings. This can have a large impact in cases of register spilling and for strided or scattered memory access patterns, and it requires no code changes. Outside these short experiments, however, there are two major factors that we want to examine, namely the access pattern and the number of concurrent memory requests.

To determine if the access pattern is the problem, we compare the number of memory instructions with the number of transferred bytes. For example, for global load we should compare the number of bytes requested (`gld_request` multiplied by bytes per request, 128 bytes) to the number of transferred bytes (the sum of `l1_global_load_miss` and `l1_global_load_hit` multiplied by the cache line size, 128 bytes). If the number of instructions per byte is far larger than one, we have a clear indication that global memory loads have a problematic access pattern. In that case, we should try reorganizing the memory access pattern to better fit with the rules in Section 3. For global store, the counters we should compare are `gst_request` and `global_store_transactions`.

If we are memory bound, but the access patterns are good, we might be suffering from having too few outstanding memory requests: according to Little's Law [13], we need (memory latency × bandwidth) bytes in flight to saturate the bus. To determine if the number of concurrent memory accesses is too low, we can compare the achieved memory throughput (`glob_mem_read_throughput` and `glob_mem_write_throughput` in the profiler) against the theoretical peak. If the hardware throughput is dramatically lower, the memory bus is not saturated, and we should try increasing the number of concurrent memory transactions by increasing the occupancy. This can be done through adjustment of block dimensions or reduction of register count, or we can modify the kernel to process more data elements per thread. A further optimization path is to move indexing calculations and memory transactions in an attempt to achieve better overlap of memory transactions and arithmetic operations.

An important note when it comes to optimization of memory on the GPU is that traditional CPU cache-blocking techniques typically do not work. This is because the GPU's L1 and L2 caches are not aimed at *temporal* reuse like CPU caches usually are, which means that attempts at cache blocking can even be counterproductive. The rule of thumb is therefore that, when optimizing for memory throughput on GPUs, do not think of caches at all. However, fetching data from textures can alleviate pressure on the memory system since these fetches go through a different cache in smaller transactions. Nevertheless, the L1 cache is superior in performance, and only in rare cases will the texture cache increase performance [18].

4.3. Profiling and optimizing latencies and instruction throughput

If a kernel is bound by instruction throughput, there may be several underlying causes. Warp serialization (see Section 3.1) may be a major bottleneck, or we may similarly have that bank conflicts cause shared memory serialization. The third option is that we have data dependencies that inhibit performance.

Instruction serialization means that some threads in a warp "replay" the same instruction as opposed to all threads issuing the same instruction only once (see Fig. 6b). The profiler can be used to determine the level of instruction serialization by comparing the `instructions_executed` counter to the `instructions_issued` counter, in which the difference is due to serialization. Note that even if there is a difference between instructions executed and instructions issued, this is only a problem if it constitutes a significant percentage.

One of the causes for instruction replays is divergent branches, identified by comparing the `divergent_branch` counter to the `branch` counter. We can also profile it by modifying the source code so that all threads take the same branch, and compare the run-times. The remedy is to remove as many branches as possible, for example by sorting the input data or splitting the kernel into two separate kernels (see Section 3.1).

Another cause for replays is bank conflicts, which can be the case if the `l1_shared_bank_conflict` counter is a significant percentage of the sum of the `shared_loads` and `shared_stores` counters. Another way of profiling bank conflicts is to modify the kernel source code by removing the bank conflicts. This is done by changing the indexing so that all accesses are either broadcasts (all threads access the same element) or conflict free (each thread uses the index `threadIdx.y*blockDim.x+threadIdx.x`). The shared memory variables also need to be declared as volatile to prevent the compiler from storing them in registers in the modified kernel. Padding is one way of removing these bank conflicts (see Section 3.2), and one can also try rearranging the shared memory layout (e.g., by storing by columns instead of by rows).

If we have ruled out the above causes, our kernel might be suffering from arithmetic operation latencies and data dependencies. We can find out if this is the case by comparing the kernel performance to hardware limits. This is done by examining the `IPC - instructions/cycle` counter in the profiler, which gives the ratio of executed instructions per clock cycle. For compute capability 2.0, this figure should be close to 2, whilst for compute capability 2.1 it should approach 4 instructions per cycle. If the achieved instructions per cycle count is very low, this is a clear indication that there are data dependencies and arithmetic latencies that affect the performance. In this case, we can try storing intermediate calculations in separate registers to minimize arithmetic latencies due to register dependencies.

4.4. Further optimization parameters

Grid size and block size are important optimization parameters, and they are usually not easy to set. Both the grid size and the block size must be chosen according to the size and structure of the input data, but they must also be tuned to fit the GPU's architecture in order to yield a high performance. Most importantly, we need enough total threads to keep the GPU fully occupied in order to hide memory and other latencies. Since each multiprocessor can execute up to eight blocks simultaneously, choosing too small blocks prevents a high occupancy. Simultaneously, we do not want too large blocks since this may cause register spilling if the kernel uses a lot of registers. The number of threads per block should also, if possible, be a multiple of 32, since each multiprocessor executes full warps. When writing a kernel for the GPU, one also often encounters a situation in which the number of data elements is not a multiple of the block size. In this case, it is recommended to launch a grid larger than the number of elements and use an out-of-bounds test to discard the unnecessary computations.

In many cases, it is acceptable to trade accuracy for performance, either because we simply need a rough estimate, or because modeling or data errors shadow the inevitable floating-point rounding errors. The double-precision to single-precision ratio for GPUs is 2:1,[5] which means that a double-precision operation takes twice as long as a single-precision operation (just as for CPUs). This makes it well worth investigating whether or not single precision is sufficiently accurate for the application. For many applications, double precision is required for calculating results, but the results themselves can be stored in single precision without loss of accuracy. In these cases, all data transfers will execute twice as fast, simultaneously as we will only occupy half the space in memory. For other cases, single precision is sufficiently accurate for arithmetics as well, meaning that we can also perform the floating-point operations twice as quickly. Remember also that all floating-point literals without the `f` suffix are represented using 64 bits of precision according to the C standard, and that, when one operand in an expression uses 64 bits, all operations must be performed using 64 bit precision. Some math functions can be compiled directly into faster, albeit less accurate, versions. This is enabled using the double-underscore version of the function, for example `__sin()` instead of `sin()`. By using the `--use_fast_math` compiler flag, all fast hardware math functions that are available will be used. It should also be noted that this compiler flag also treats denormalized numbers as zero, and faster (but less accurate) approximations are used for divisions, reciprocals, and square roots.

Even if an application does none of its computing on the CPU, there still must be some CPU code for setting up and controlling CUDA contexts, launching kernels, and transferring data between the CPU and the GPU. In most cases it is also desirable to have some computations performed on the CPU. There is almost always some serial code in an algorithm that cannot be parallelized and therefore will execute faster on the CPU. Another reason is that there is no point in letting the CPU idle while the GPU does computations: use both processors when possible.

There are considerable overheads connected with data transfers between the CPU and the GPU. To hide a memory transfer from the CPU to the GPU before a kernel is launched one can use streams, issue the memory transfer asynchronously, and do work on the CPU while the data is being transferred to the GPU (see Section 3.3). Since data transfers between the CPU and the GPU goes through the PCI Express bus both ways, these transfers will often be a bottleneck. By using streams and asynchronous memory transfers, and by trying to keep the data on the GPU as much as possible, this bottleneck can be reduced to a minimum.

4.5. Auto-tuning of GPU kernels

The above-mentioned performance guidelines are often conflicting, one example being that you want to optimize for occupancy to hide memory latencies simultaneously as wanting to increase the number of per-thread registers for more per-thread storage. The first of these criteria requires more threads per block, the second requires fewer threads per block, and it is not given which configuration will give the best performance. With the sheer number of conflicting optimization parameters, it rapidly becomes difficult to find out what to optimize. Experienced developers are somewhat guided by educated guesses together with a trial and error approach, but finding the global optimum is often too difficult to be performed manually.

Auto-tuning strategies have been known for a long time on CPUs, and are used to optimize the performance using cache blocking and many other techniques. On GPUs, we can similarly create auto-tuning codes that execute a kernel for each set of optimization parameters, and select the best-performing configuration. However, the search space is large, and brute force techniques are thus not a viable solution. Pruning of this search space is still an open research question, but several papers on the subject have been published (see, for example, [12,6]).

Even if auto-tuning is outside the scope of a project, preparing for it can still be an important part of profile-guided development. The first part of auto-tuning is often to use template arguments for different kernel parameters such as shared memory and block size. This gives you many different varieties of the same kernel, so you can easily switch between different implementations. A benefit of having many different implementations of the same kernel is that you can perform run-time auto-tuning of your code. Consider the following example. For a dam-break simulation on the GPU, you might have one kernel that is optimized for scenarios in which most of the domain is dry. However, as the dam breaks, water spreads throughout the domain, making this kernel inefficient. You then create a second kernel that is efficient when most of the domain contains water, and switch between these two kernels at run time. One strategy here is to perform a simple check of which kernel is the fastest after given number of iterations, and use this kernel. Performing this check every hundredth time step, for example, gives a very small overhead to the total computational time, and ensures that you are using the most efficient kernel throughout the simulation.

4.6. Reporting performance

One of the key points made in early GPU papers was that one could obtain high speedups over the CPU for a variety of different algorithms. The tendency has since been to report ever-increasing speedups, and today papers report that their codes run anything from tens to hundreds and thousands times faster than CPU "equivalents". However, when examining the theoretical performance of the architectures, the performance gap is roughly seven times between state-of-the-art CPUs and GPUs (see Fig. 2). Thus, reporting a speedup of hundreds of times or more holds no scientific value without further explanations supported by detailed benchmarks and profiling results.

The sad truth about many papers reporting dramatic speedup figures is that the speedup is misleading, at best. Often, a GPU code can be compared to an inefficient CPU code, or a state-of-the-art desktop GPU can be compared to a laptop CPU several years old. Some claims of SSE and other optimizations of the CPU code are often made, giving the impression that the CPU code is efficient. However, for many implementations, this still might not be the case: if you optimize using SSE instructions, but the bottleneck is memory latency, your optimizations are worthless in a performance perspective.

Reporting performance is a difficult subject, and the correct way of reporting performance will vary from case to case. There is also a balance between the simplicity of the benchmark and its

[5] For the GeForce products this ratio is 8:1.

ease of interpretation. For example, the Top500 list [23], which rates the fastest supercomputers in the world, is often criticized for being too simplistic as it gives a single rating (gigaflops) from a single benchmark (solving a linear system of equations). For GPUs, we want to see an end to the escalating and misleading speedup race, and rather see detailed profiling results. This will give a much better view of how well the algorithm exploits the hardware, both on the CPU and on the GPU. Furthermore, reporting how well your implementation utilizes the hardware, and what the bottlenecks are, will give insight into how well it will perform on similar and even future hardware. Another benefit here is that it becomes transparent what the bottleneck of the algorithm is, meaning that it will become clear what hardware vendors and researchers should focus on for improving performance.

5. Debugging

As an ever-increasing portion of the C++ standard is supported by CUDA and more advanced debugging tools emerge, debugging GPU codes becomes more and more like debugging CPU codes. Many CUDA programmers have encountered the ''unspecified launch failure'', which could be notoriously hard to debug. Such errors were typically only found by either modification and experimenting, or by careful examination of the source code. Today, however, there are powerful CUDA debugging tools for all the most commonly used operating systems.

CUDA-GDB, available for Linux and Mac, can step through a kernel line by line at the granularity of a warp, e.g., identifying where an out-of-bounds memory access occurs, in a similar fashion to debugging a CPU program with GDB. In addition to stepping, CUDA-GDB also supports breakpoints, variable watches, and switching between blocks and threads. Other useful features include reports on the currently active CUDA threads on the GPU, reports on current hardware and memory utilization, and in-place substitution of changed code in a running CUDA application. The tool enables debugging on hardware in real time, and the only requirement for using CUDA-GDB is that the kernel is compiled with the `-g -G` flags. These flags make the compiler add debugging information into the executable, and the executable to spill all variables to memory.

On Microsoft Windows, Parallel NSight is a plug-in for Microsoft Visual Studio which offers conditional breakpoints, assembly-level debugging, and memory checking directly in the Visual Studio IDE. It furthermore offers an excellent profiling tool, and is freely available to developers. However, debugging requires two distinct GPUs, one for display, and one for running the actual code to be debugged.

6. Trends in GPU computing

GPUs have truly been a disruptive technology. Starting out as academic examples, they were shunned in scientific and high-performance communities: they were inflexible, inaccurate, and required a complete redesign of existing software. However, with time, there has been a low-end disruption, in which GPUs have slowly conquered a large portion of the high-performance computing segment, and three of the five fastest supercomputers are powered mainly by GPUs [23]. GPUs were never designed with high-performance computing in mind, but have nevertheless evolved into powerful processors that fit this market perfectly in less than ten years. The software and the hardware have developed in harmony, both due to the hardware vendors seeing new possibilities and due to researchers and industry identifying points of improvement. It is impossible to predict the future of GPUs, but by examining their history and current state, we might be able to identify some trends that are worth noting.

The success of GPUs has partly been due to their price: GPUs are ridiculously inexpensive in terms of performance per dollar. This again comes from the mass production and target market. Essentially, GPUs are inexpensive because NVIDIA and AMD sell a lot of GPUs to the entertainment market. After researchers started to exploit GPUs, these hardware vendors eventually developed functionality that is tailored for general-purpose computing and started selling GPUs intended for computing alone. Backed by the mass market, this means that the cost for the vendors to target this emerging market is very low, and the profits high.

There are great similarities between the vector machines of the 1990s and today's use of GPUs. The interest in vector machines eventually died out, as the x86 market took over. Will the same happen to GPUs once we conquer the power wall? In the short term, the answer is no, for several reasons. First of all, conquering the power wall is not even on the horizon, meaning that, for the foreseeable future, parallelism will be the key ingredient that will increase performance. Also, while vector machines were reserved for supercomputers alone, today's GPUs are available in everything from cell phones to supercomputers. A large number of software companies now use GPUs for computing in their products due to this mass market adaption, and this is one of the key reasons why we will have GPUs or GPU-like accelerator cores in the future. Just as x86 is difficult to replace, it is now becoming difficult to replace GPUs, as the software and hardware investments in this technology are large and increasing.

NVIDIA have just released their most recent architecture, called Kepler [19], which has several updates compared to the Fermi architecture described in this paper. First of all, it has changed the organization of the multiprocessors: whilst the Fermi architecture has up to 16 streaming multiprocessors, each with 32 CUDA cores, the new Kepler architecture has only 4 multiprocessors, but each with 192 CUDA cores. This totals 1536 CUDA cores for one chip, compared to 512 for the Fermi architecture. A second change is that the clock frequency has been decreased from roughly 1.5 GHz to just over 1 GHz. These two changes are essentially a continuation of the existing trend of increasing performance through more cores running at a decreased clock frequency. Combined with a new production process of 28 nm (compared to 40 nm for Fermi), the effect is that the new architecture has a lower power consumption, yet roughly doubles the gigaflops performance. The bandwidth to the L2 cache has been increased giving a performance boost for applications that can utilize the L2 cache well, yet the main memory bandwidth is the same. NVIDIA has also announced that it plans to build high-performance processors with integrated CPU cores and GPU cores based on the low-power ARM architecture and the future Maxwell GPU architecture.

Competition between CPUs and GPUs is likely to be intense for conquering the high-performance and scientific community, and heterogeneous CPU–GPU systems are already on the market. The AMD Fusion architecture [1] incorporates multiple CPU and GPU cores into a single die, and Intel have released their Sandy Bridge architecture based on the same concept. Intel have also developed other highly parallel architectures, including the Larrabee [21] based on simple and power-efficient x86 cores, the Single-chip Cloud Computer (SCC) [10], and the 80-core tera-scale research chip Polaris [24]. These architectures have never been released as commercial products, but have culminated into the Knights Corner co-processor with up to 1 teraflops performance [9]. This co-processor resides on an adapter connected through the PCI Express bus, just as today's graphics adapters, but exposes a traditional C, C++ and Fortran programming environment, in which existing legacy code can simply be recompiled for the new architecture.

Today, we see that the processors converge towards incorporating traditional CPU cores in addition to highly parallel accelerator cores on the same die, such as the aforementioned AMD Fusion and Intel Sandy Bridge. These architectures do not target the high-performance segment. However, NVIDIA have plans for the high-performance segment with their work on combining ARM

CPU cores and Maxwell GPU cores on the same chip. We thus see it as likely that we will see a combination of CPU and GPU cores on the same chip, sharing the same memory space, in the near future. This will have a dramatic effect on the memory bottleneck that exists between these architectures today, and open for tighter cooperation between fast serial execution and massive parallel execution to tackle Amdahl's law.

7. Summary

In this article, we have given an overview of hardware and traditional optimization techniques for the GPU. We have furthermore given a step-by-step guide to profile-driven development, in which bottlenecks and possible solutions are outlined. The focus is on state-of-the-art hardware with accompanying tools, and we have addressed the most prominent bottlenecks: memory, arithmetics, and latencies.

References

[1] Advanced micro devices, AMD Fusion family of APUs: enabling a superior, immersive PC experience, Technical report, 2010.

[2] G.M. Amdahl, Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000, pp. 79–81 (Chapter 2).

[3] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, K. Yelick, The landscape of parallel computing research: a view from Berkeley, Technical report, EECS Department, University of California, Berkeley, December 2006.

[4] A.R. Brodtkorb, C. Dyken, T.R. Hagen, J.M. Hjelmervik, O. Storaasli, State-of-the-art in heterogeneous computing, Scientific Programming 18 (1) (2010) 1–33.

[5] A.R. Brodtkorb, M.L. Sætra, M. Altinakar, Efficient shallow water simulations on GPUs: implementation, visualization, verification, and validation, Computers & Fluids 55 (0) (2012) 1–12.

[6] A. Davidson, J.D. Owens, Toward techniques for auto-tuning GPU algorithms, in: Proceedings of Para 2010: State of the Art in Scientific and Parallel Computing, 2010.

[7] M. Harris, NVIDIA GPU computing SDK 4.1: optimizing parallel reduction in CUDA, 2011.

[8] M. Harris, D. Göddeke, General-purpose computation on graphics hardware, http://gpgpu.org.

[9] Intel, Intel many integrated core (Intel MIC) architecture: ISC'11 demos and performance description, Technical report, 2011.

[10] Intel Labs, The SCC platform overview, Technical report, Intel Corporation, 2010.

[11] D.E. Knuth, Structured programming with go to statements, Computing Surveys 6 (1974) 261–301.

[12] Y. Li, J. Dongarra, S. Tomov, A note on auto-tuning gemm for GPUs, in: Proceedings of the 9th International Conference on Computational Science: Part I, 2009.

[13] J.D.C. Little, S.C. Graves, Building Intuition: Insights from Basic Operations Management Models and Principles, Springer, 2008, pp. 81–100 (Chapter 5).

[14] D.T. Marr, F. Binns, D.L. Hill, G. Hinton, D.A. Koufaty, J.A. Miller, M. Upton, Hyper-threading technology architecture and microarchitecture, Intel Technology Journal 6 (1) (2002) 1–12.

[15] P. Micikevicius, Analysis-driven performance optimization, [Conference presentation], 2010 GPU Technology Conference, Session 2012, 2010.

[16] P. Micikevicius, Fundamental performance optimizations for GPUs, [Conference presentation], 2010 GPU Technology Conference, session 2011, 2010.

[17] NVIDIA, NVIDIA's next generation CUDA compute architecture: Fermi, 2010.

[18] NVIDIA, NVIDIA CUDA programming guide 4.1, 2011.

[19] NVIDIA, NVIDIA GeForce GTX680, Technical report, NVIDIA Corporation, 2012.

[20] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, J. Phillips, GPU computing, Proceedings of the IEEE 96 (5) (2008) 879–899.

[21] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, P. Hanrahan, Larrabee: a many-core x86 architecture for visual computing, ACM Transactions on Graphics 27 (3) (2008) 18:1–18:15.

[22] G. Taylor, Energy efficient circuit design and the future of power delivery, [Conference presentation], Electrical Performance of Electronic Packaging and Systems, October 2009.

[23] Top 500 supercomputer sites, http://www.top500.org/, November 2011.

[24] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, S. Borkar, An 80-tile sub-100-w teraflops processor in 65-nm CMOS, Solid-State Circuits 43 (1) (2008) 29–41.

Dr. André R. Brodtkorb is a research scientist at SINTEF, a non-profit research organization in Norway with roughly 2000 researchers, where he works on GPU acceleration and algorithm design. He is also associate professor at the Norwegian School of Information Technology, where he teaches several courses on graphics programming. His research interests include GPU and heterogeneous computing, simulation of partial differential equations (PDEs), and real-time graphics and visualization.

Trond R. Hagen has been head of the Heterogeneous Computing Group at SINTEF ICT since January 2009 and has an adjunct position at Narvik University Collage, where he teaches computer graphics, virtual reality, and animation. His research focus is on massive parallel computing, visual computing, and cloud computing on heterogeneous architectures, e.g., combination of CPUs and GPUs. His research areas are computer graphics, visualization, geometric modeling, and simulation (conservation laws, smoothed particle hydrodynamics).

Martin L. Sætra is a Ph.D. fellow working with heterogeneous computing, and the GPU in particular. In his master's thesis, he was one of the first to utilize GPU clusters to numerically solve partial differential equations. He also holds a part-time position at the Norwegian Meteorological Institute as a developer, where he works with visualization tools, among other things.