Prepared (also subject responsible if other)		No.		
SMW/DD/GX Jimmy Pettersson, Ian Wainwright		5/0363-FCP1041180 en		
Approved	Checked	Date	Rev	Reference
SMW/DD/GCX Lars Henricson	DD/LX	2010-01-27	A	

Radar Signal Processing with Graphics Processors (GPUs)

Abstract

Radar Signal Processing algorithms place strong real-time performance demands on computer architectures. These algorithms do however have an inherent data-parallelism that allows for great performance on massively parallel architectures, such as the Graphics Processing Unit (GPUs). Recently, using GPUs for other than graphics processing has become a possibility through the CUDA and OpenCL (Open Computing Language) architectures. This master thesis aims at evaluating the Nvidia GT200 series GPU-architecture for radar signal processing applications.

The investigation is conducted through comparing a GPU (GTX260) against a modern desktop CPU for several HPEC (High Performance Embedded Computing) and other radar signal processing algorithms; 12 in total. Several other aspects are also investigated, such as programming environment and efficiency, future GPU-architectures, and applicability in radar systems.

Our CUDA GPU-implementations perform substantially better than the CPU and associated CPU-code used for all but one of the 12 algorithms tested, sometimes by a factor of 100 or more. The OpenCL implementations also perform substantially better than the CPU.

The substantial performance achieved when using CUDA for almost all benchmarks can be attributed to both the high theoretical performance of the GPU, but also to the inherent data-parallelism, and hence GPU-suitability, of almost all of the investigated algorithms. Programming CUDA is reasonably straight forward, largely due to the mature development environment and abundance of documentation and white-papers. OpenCL is a lot more tedious to program. Furthermore, the coming CUDA GPU-architecture called Fermi is expected to further increase performance and programmability.

When considering system integration of GPU-architectures into harsh radar application environments, one should be aware of potential heat and also possible obsolescence issues.

Prepared (also subject responsible if other)		No.		
SMW/DD/GX Jimmy Pettersson, Ian Wainwright		5/0363-FCP1041180 en		
Approved	Checked	Date	Rev	Reference
SMW/DD/GCX Lars Henricson	DD/LX	2010-01-27	A	

Contents

Prepared (also subject responsible if other)		No.		
SMW/DD/GX Jimmy Pettersson, Ian Wainwright		5/0363-FCP1041180 en		
Approved	Checked	Date	Rev	Reference
SMW/DD/GCX Lars Henricson	DD/LX	2010-01-27	A	

1 Terminology

Active thread: A thread that is in the process of executing on an SM, unlike non-active threads that have yet to be assigned an SM.

AI: Arithmetic Intensity. A measure the number of instructions per memory element, detailed in 9.4.

Block: A unit of threads that can communicate with each other. Blocks build up the grid, described below.

CPU: Central Processing Unit.

CUDA: (**koo** - *duh*): Formally "Compute Unified Device Architecture", now simply CUDA, the architectural framework that is used to access and code the GPU.

Device: The graphics board, i.e. the GPU and GPU–RAM.

ECC: Error-Correcting Code.

Embarrassingly parallel: Problems that can easily be parallelized at no cost, for example vector addition where all elements are independent of each other and can hence the global problem can with ease be divided into many smaller parallel problems.

FMA: Fused Multiply Addition. Multiplication and addition are performed in a single instruction.

FPU: Floating point unit.

GPGPU: General-Purpose computing on Graphics Processing Units.

GPU: Graphics Processing Unit.

Grid: A set of blocks. The grid is the highest part of the hierarchy defining the computational workspace.

Host: The CPU and its related hardware, such as CPU–RAM.

HPC: High Performance Computing.

Kernel: A set of code that runs on the GPU but is initiated from the CPU. Once the kernel has been started, the CPU is free and does not have to wait for the kernel to finish.

OpenCL: Open Computing Language. An open standard designed for cross-platform program execution on heterogeneous hardware.

SFU: Special Function Unit in the micro-architecture of the GPU, used for transcendental and functions such as sin and exp.

SIMD: Single Instruction Multiple Data.

SIMT: Single Instruction Multiple Thread.

SM: Streaming Multiprocessor consisting of 8 SPs, one 64-bit FMA FPU and 2 SFUs.

SP: Streaming Processor core, a 32-bit FMA FPU. 8 of them reside on each SM.

Thread: Similar to a light CPU thread. Threads build up the blocks. Sets of 32 threads build up a warp. A thread has no stack.

Warp: A set of 32 threads that follows the same instruction path.

2 Introduction

As the laws of physics are forcing chip makers to turn away from increasing the clock frequency and instead focus on increasing core-count, cheap and simple performance increases following Moore's law alone are no longer possible. The following rapid and broad hardware development has created a large set of diverse parallel hardware architectures, such as many-core CPUs, the Cell Broadband Engine, Tillera 64, Larrabee, Sun UltraSparc, Niagara [1] and recently also GPUs.

Despite the rapid development in parallel computing architectures, leveraging this computing power is largely missing in software due to many of the great and costly difficulties with writing parallel code and applications. This is however not due to lack of attempts to write programming languages and APIs that exploit the recent boom in hardware parallelism. Some examples are CUDA, Ct (C/C++ for throughput), MPI, the recent OpenCL (Open Computing Language), and also OpenMP.

This report focuses on evaluating the Nvidia CUDA GPU architecture for radar signal processing. Specifically, we investigate some of the HPEC (High Performance Embedded Computing) challenge algorithms and also other typical radar signal processing algorithms. CUDA's suitability for the various algorithms is also investigated for radar relevant data sets and sizes, and comparisons to benchmarks run on a regular desktop CPU and other similar CUDA performance investigations are also made. OpenCL is also implemented on a few algorithms to serve as a brief OpenCL evaluation.

The evaluation is conducted by first thoroughly investigating the Nvidia CUDA hardware and software, then relating CUDA's strong points and weaknesses in relation to the performance of each respective algorithm based on benchmark tests. Alternative views to current CUDA programming recommendations and also an investigation into sparsely or not at all documented features are also included. A brief history of GPGPU programming, the CUDA software environment, and some potential future developments are also discussed.

This investigation is a master thesis supervised by and conducted at Saab Microwave Systems in Gothenburg, now Electronic Defence Systems within Saab. It was done as a part of the EPC (Embedded Parallel Computing) project run by Halmstad University's CERES (Centre for Research on Embedded Systems) department in which Saab participates. The master thesis is done via Uppsala University and accounts for two students working full-time for 20 weeks each.

3 The potential of the GPU as a heavy duty co-processor

Before reading the following chapters of this document, the reason why one should consider the GPU for heavy duty computing in the first place must be established, and hence we make a quantitative comparison of parameters of significance for high performance computing. In the past years, GPUs have increased their potential compute and bandwidth capacity remarkably faster than similarly priced CPUs, which can be seen in Figure 1 below.

Prepared (also subject responsible if other)		No.		
SMW/DD/GX Jimmy Pettersson, Ian Wainwright		5/0363-FCP1041180 en		
Approved	Checked	Date	Rev	Reference
SMW/DD/GCX Lars Henricson	DD/LX	2010-01-27	A	

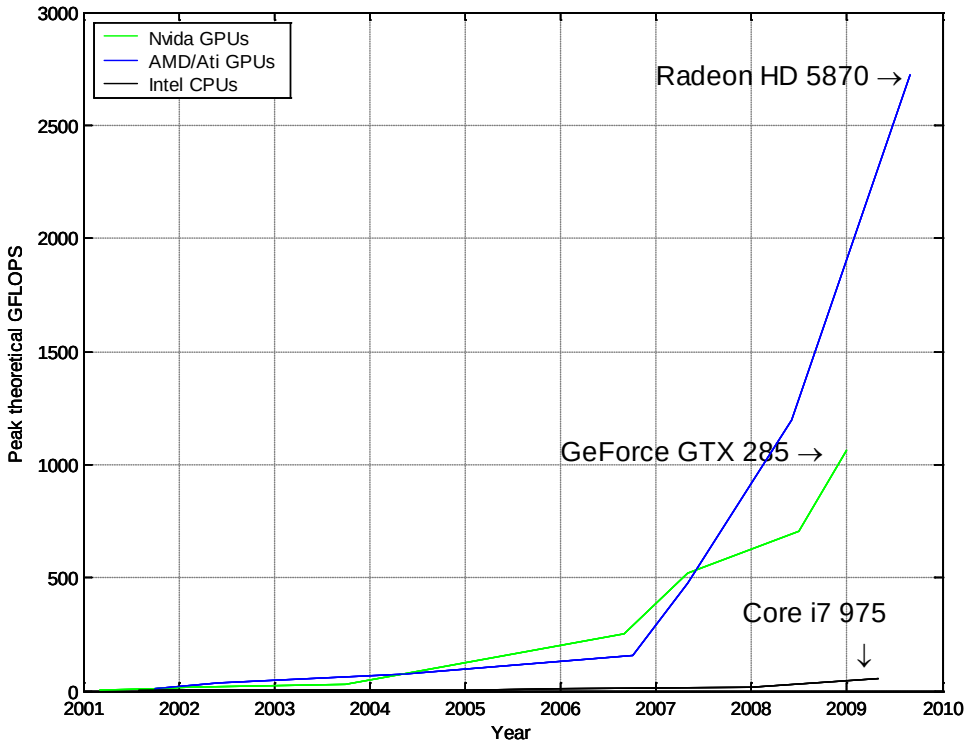

Figure 1 The peak performance of single GPUs versus single Intel multicore CPUs over time.

Modern GPUs are highly parallel computing devices that have a potential of performing certain computations many times faster than similarly priced CPU and at the same power dissipation. High-performance GPUs are as of November of 2009, close to 3 TFLOPS of single precision computations [2]. 1 Table 2 below shows specified peak performance, bandwidth and watt for various present computing devices. Note that we are comparing graphics boards, i.e. GPU-chip and GPU RAM vs. only the CPU-chip.

Graphics board	GFLOPS	Bandwidth (GB/s)	Watt	GFLOPS/ Watt	GFLOPS/ Bandwidth
GeForce GT 240M[1]	174	25.6	23	7.6	6.7
GeForce GTS 250M[1]	360	51.2	28	12.9	7.0
GeForce GTX 260 core 216[1]	875	112	171	5.1	7.8
AMD/ATI 5870 [2]	2720	153.6	188	14.5	17.7
AMD 5750[3]	1008	73.6	86	11.8	13.7
AMD 5770[3]	520	51.2	30	17.3	10.2
AMD 5830[3]	800	25.6	24	33.3	31.3
AMD 5870[3]	1120	64	50	22.4	17.5

Table 1 Specifications for some graphics boards, i.e. GPU-chip and GPU RAM.

CPU	GFLOPS	Bandwidth (GB/s)	Watt	GFLOPS/ Watt	GFLOPS/ Bandwidth
Intel Core2Duo E8600[4]	27	10.4	65	0.4	2.6
Intel Core2Duo SP9600, [4], [5]	20	8.33	25	0.8	2.4
Intel Core i7-870, [4], [6]	47	22.89	95	0.5	2.1
Intel Core i7-820QM, [4], [6]	28	21	45	0.6	1.3

Table 2 Specifications for some CPUs.

As can be seen, the GPUs have a clear advantage both in raw performance, bandwidth, and also performance-per-watt.

[1] http://en.wikipedia.org/wiki/Comparison_of_Nvidia_graphics_processing_units
[2] http://www.amd.com/US/PRODUCTS/DESKTOP/GRAPHICS/ATI-RADEON-HD-5000/Pages/ati-radeon-hd-5000.aspx
[3] http://www.amd.com/US/PRODUCTS/NOTEBOOK/GRAPHICS/Pages/notebook-graphics.aspx
[4] http://www.intel.com/support/processors/sb/cs-023143.htm#4
[5] http://en.wikipedia.org/wiki/Wolfdale_(microprocessor)
[6] http://en.wikipedia.org/wiki/List_of_Intel_Core_i7_microprocessors

The reason for the GPUs advantage compared to the CPU is that the GPU design-idea has been to sacrifice the complex, low serial latency architecture of the desktop CPU and instead focusing on a simple, highly parallel, high bandwidth architecture instead, as this is more important in both gaming and graphic intense applications, the main areas of use for GPUs today. Not only is there more potential performance and performance/Watt in a GPU, but also, according to Michael Feldman, HPCwire Editor, the “GPUs have an additional advantage. Compared to a graphics memory, CPU memory tends to be much more bandwidth constrained, thus it is comparatively more difficult to extract all the theoretical FLOPS from the processor. This is one of the principal reasons that performance on data-intensive apps almost never scales linearly on multicore CPUs. GPU architectures, on the other hand, have always been designed as data throughput processors, so the FLOPS to bandwidth ratio is much more favourable.”[7]

Access to the Nvidia GPUs and their potential is available through their proprietary framework called CUDA. CUDA uses the common C-language with some extension to allow easy writing of parallel code to run on the GPU.

In the following chapters we will in detail describe the hardware that supports CUDA, its consequences on performance and programmability, and how to best exploit its benefits and avoid the downsides. But first, a brief history of GPGPU and its recent developments.

4 GPGPU: A concise history

GPGPU is the process of using the GPU to perform calculations other than the traditional graphics they once were solely designed for. Initially, GPUs where limited, inflexible, fixed function units and access to their computational power lay in the use of various vertex shader languages and graphics APIs, such as DirectX, Cg, OpenGL, and HLSL. This meant writing computations as if they were graphics through transforming the problem into pixels, textures and shades [3]. This approach meant programmers had to learn how to use graphics APIs and that they had little access to the hardware in any direct manner.

This, however, has recently begun to change.

One of the larger changes[3] in the GPGPU community was Microsoft's media API DirectX10 released in November 2006. As Microsoft has a dominating role in most things related to the GPU, Microsoft was able to make demands on GPU-designers to increase the flexibility of the hardware in terms of memory access and adding better support for single precision floats; in essence making the GPU more flexible.

[7] http://www.hpcwire.com/specialfeatures/sc09/features/Nvidia-Unleashes-Fermi-GPU-for-HPC-70166447.html

Also, the two main GPU developers, AMD which purchased GPU developer ATI in 2006, and Nvidia have also developed their own proprietary GPGPU systems; AMD/ATI with *Close to Metal* in November 2006 [4], cancelled and replaced by its successor *Stream Computing* in released in November 2007; and Nvidia's *CUDA*, released in February 2007 [5]. Also, AMD/ATI are, in similar fashion with Intel, presently preparing to integrate as of yet small GPUs onto the CPU chip, creating what AMD call an APU, Accelerated Processing Unit.

In the summer of 2008, Apple Inc., possibly in fear of being dependent on other corporations' proprietary systems, also joined the GPGPU bandwagon by proposing specifications [6] for a new open standard called OpenCL (Open Computing Language), run by the Khronos group, a group for royalty-free open standards in computing and graphics. The standard “is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, and other processors”[8], and now has support from major hardware manufacturers such as AMD, Nvidia, Intel, ARM, Texas Instruments and IBM among many others.

Additionally, with the release of Windows 7 in October 2009, Microsoft released its present version of the DirectX API, version 11, which now includes a Windows specific API specifically aimed at GPGPU programming, called DirectCompute.

As can been seen above, the history of GPGPU is both short and intense. With the release of DirectX11 in 2009, AMD/ATI not only supporting DirectCompute but also OpenCL beside their own Stream Computing, Nvidia's CUDA as of writing having already been through one major with a second on its way [7] and also supporting both DirectCompute and OpenCL [8], Apple integrating and supporting only OpenCL [9] into their latest OS released in August 2009, an OpenCL Development kit for the Cell Broadband Engine, Intel's hybrid CPU/GPU Larrabee possibly entering the stage in the first half of 2010 [10] and AMD planning to release their combined CPU/GPU called Fusion in 2011 [11], the road in which GPGPU is heading is far from clear, both in regular desktop applications, science and engineering. Also worth pointing out is that both AMD but mainly Nvidia have begun to push for use of GPUs in HPC and general supercomputing.

[8] http://en.wikipedia.org/wiki/OpenCL

5 CUDA: History and future

CUDA 1.0 was first released to the public in February 2007 and to date; over 100 million CUDA enabled GPUs have been sold [12]. CUDA is currently in version 2.3 and is supported by Nvidia on multiple distributions of Linux, as well as Windows XP, Vista and 7, and also on the Mac. Since its release, Nvidia have been adding new features accessible to the programmer and also improvements of the hardware itself. Support for double precision has been added in version 2.0 to serve the needs of computational tasks that demand greater accuracy than the usual single precision of GPUs. A visual profiler has also been developed to assist the programmer in optimization.

Major developments in programmability since its release have mostly been to add new functions, such as exp10, the sincos function among many others. Also, there has been increased accuracy in many functions, and vote and atomic functions have also been added in the 2.0 release.

Furthermore, there has been plenty of development with various libraries adding CUDA-support, such as CUFFT, CUDA's Fast Fourier Transforms library, CUBLAS, CUDA extensions to speed up the BLAS[9] library, CULA, CUDA Linear Algebra Package, a version of LAPACK written to use CUDA and also a CUDA-enhanced version of VSIPL called GPUVSIPL.

As can be seen from CUDA's recent history, the flexibility of the Nvidia GPUs is increasing rapidly both in hardware and software, and considering Nvidia's clear aim at the HPC market [13] and coming Fermi architecture [14], it should be clear that CUDA is no short-time investment.

6 An introduction to CUDA enabled hardware

The CUDA framework consists of two parts; hardware and software. Here, we will give a brief introduction to both parts before delving deeper in the following chapters. This introductory chapter will be simplified and try to draw as many parallels to regular desktop CPUs as possible, bluntly assuming that the reader will be more used to dealing with them.

The hardware can be grouped into two parts; computing units and the memory hierarchy.

[9] Basic Linear Algebra Subprogram

The main element of the computing units is the streaming multiprocessor, or SM. It consists of 8 processor cores, or streaming processor cores, SPs, a simple 32-bit FMA FPU. This is not like the CELL Broadband Engine where there's one processor that controls the other eight. Instead, the SM could in many aspects be thought of as a single threaded CPU core with an 8-floats wide SIMD vector unit, as all the 8 SPs must execute the same instruction. Also, the SM has its own on-chip memory shared by the SPs, and also thread local registers. This configuration of computing units has been the standard for all CUDA capable hardware to date, the difference between various cards in regards to the computing units has been the number of SMs and the clock frequency of the SPs. A simplified image of the computing units of the SM is shown in Figure 2 below.

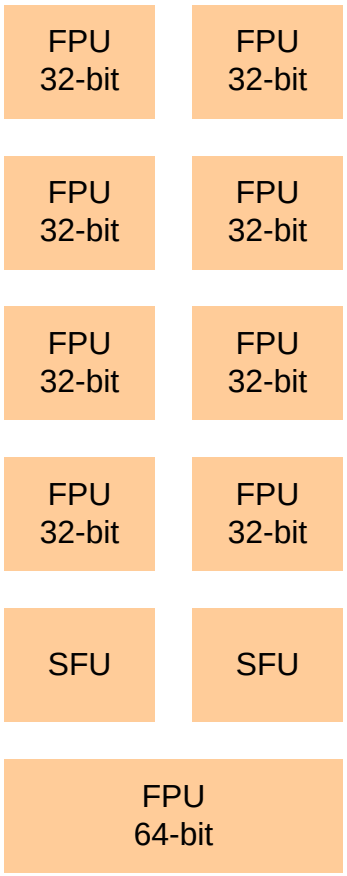

Figure 2 A schematic view of the computational units and the instruction scheduler that reside on a Streaming Multiprocessor, or SM.

The SMs are in many ways much simpler than regular desktop CPUs. For instance, there is no hardware branch prediction logic, no dynamic memory (memory sizes must be known at compile time), no recursion (except by use of templates), and the SM executes all instructions in-order.

The memory hierarchy of the GPU is in many aspects similar to most regular desktop CPUs; there is a large chunk of off-chip RAM roughly the size of your average desktop's regular RAM called *global memory*, a small "scratch pad" of fast on-chip memory per SM known as *shared memory*, in some regards similar to an L2 on-chip memory of a multi-core CPU, and lastly there are thread local *registers*, also per SM. An important difference here is that on the GPU, none of these types of memory are cached.

Hence, from high up, the GPU is in many regards similar to a simplified, many-core CPU, each CPU with its own an 8-floats wide SIMD vector unit, and also, the GPU has a similar relation to its RAM and similar on-chip memory hierarchy.

The programming language and development environments are, though severely limited in number, reasonably mature also similar to what one often would use on the CPU side. The only language supported so far is a subset of C with some extensions, such as templates and CUDA-specific functions, with support for FORTRAN in the works. For windows, there is well developed integration with Visual Studio, and similar integration for both Linux and Mac.

The main difference from the programmer's perspective is the very explicit memory control, and trying to fit computing problems to the constraints of the simple yet powerful hardware. Many of the common difficulties with parallel programming, such as passing messages and dead locks, do not exist in CUDA simply because message passing is not possible in the same way that it is on the CPU. Also, as memory is not cached, a program built using the idea of caching in mind will not execute properly. These are of course drawbacks of the simplistic hardware, but there are reasonably simple workarounds for many issues. The lack of global message passing, for instance, can often be overcome with global or local syncing, as explained in 9.7. The benefit beyond more computing power is that problems associated with the above mentioned missing features never occur; the architecture simply doesn't allow the programmer to fall into such bottlenecks as they are not possible to create in the first place.

The larger programming picture consists of the programmer writing kernels, i.e. sets of code that run on the device, for the parallel parts of a computation and letting the host handle any serial parts. The kernel is initiated from the host. Memory copies from host RAM to device RAM have to be manually initiated and completed before a kernel is executed. Likewise, memory transfers back to the host after the kernel has executed must also be manually initiated by the host. Memory transfers and kernel launches are initiated using the CUDA API.

This concludes the brief introduction of CUDA, and we will now continue with a deeper look at the hardware, followed by the software.

7 Nvidia GT200 family GPU hardware

Below is a detailed description of the device discussed throughout this document, namely the GTX 260 of Nvidia's GT200 series. First, the computing units are described in detail. This is then followed by a description of the memory hierarchy. A simplified illustration of an SM is given in Figure 3 below.

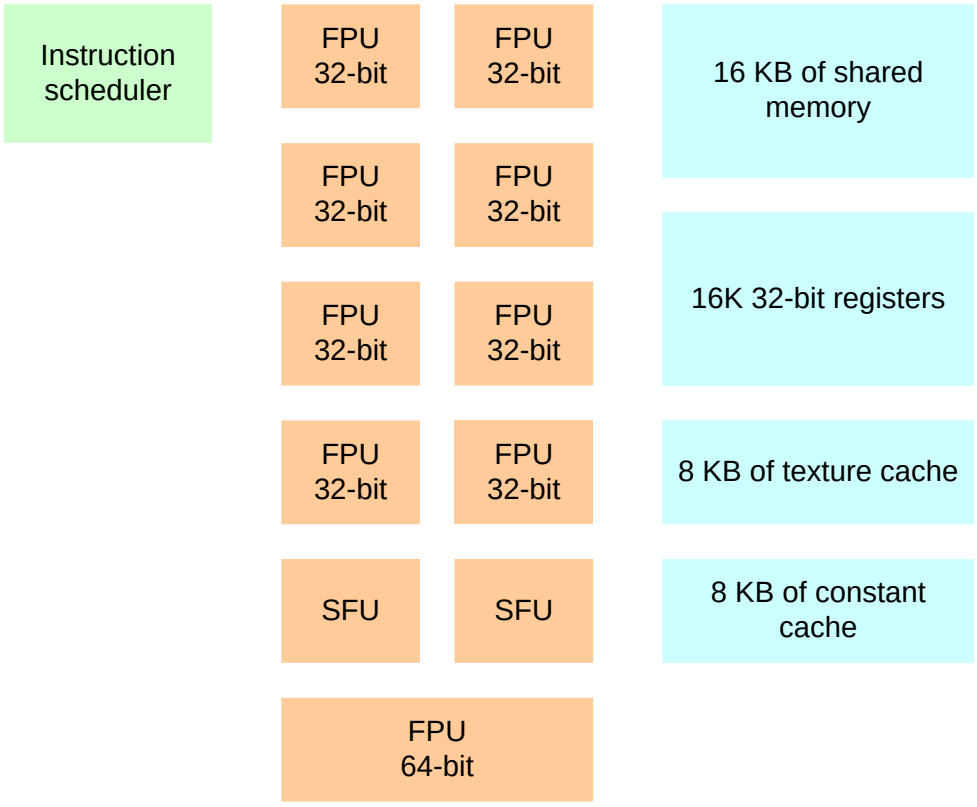

Figure 3 A schematic view of the parts of an SM. The SM includes an instruction scheduler (**green**), computational units (**pink**) and memory (**blue**).

One important point to note before we delve deeper is that finding detailed information on the underlying hardware is difficult. This seems to be a consequence solely due to Nvidia’s secrecy regarding their architecture. Hence, there is a section with theories why certain aspects of CUDA work the way they do in 7.4. Also, before continuing, we must have a brief introduction of a unit known as the warp.

7.1 A brief introduction to the Warp

CUDA exploits parallelism through using many threads in a SIMD fashion, or SIMT, 'Single Instruction Multiple Thread' as Nvidia calls it. SIMT should, when using CUDA, be read as 'Single instruction 32 threads', where 32 threads constitute a so called warp. The threads of a warp are 32 consecutive threads aligned in multiples of 32, beginning at thread 0. For instance, both threads 0-31 and threads 64-93 each constitute warps, but threads 8-39 do not as they do not form a consecutive multiple of 32 beginning at thread 0. No fewer threads than a whole warp can execute an instruction path, akin to SIMD where the same instruction is performed for several different data elements in parallel. This is all we need to know for now to continue with the hardware at a more detailed level. A programming perspective of warps is given in 9.6.1.

7.2 The SM's computational units

The GTX 260 consists of 24 Streaming Multiprocessors, SMs, and global memory. Each SM in turn consists of

- 32-bit FPU (Floating Point Unit) called Streaming Processor cores, SPs, all capable of performing FMA,
- 1 64-bit FPU, cable of FMA,
- 2 special function units, SFUs,

in total 192 SPs, 24 64-bit FPUs and 48 SFUs. The 8 SPs of an SM all perform the same instruction in a SIMD fashion. The two SFUs (Special Function Units) deal with transcendental functions such as sin, exp and similar functions. The 8 SPs, single 64-bit FPU and the 2 SFUs all operate at approximately 1200 MHz for the GT200 series. The computational units and the instruction scheduler are illustrated in Figure 4 below.

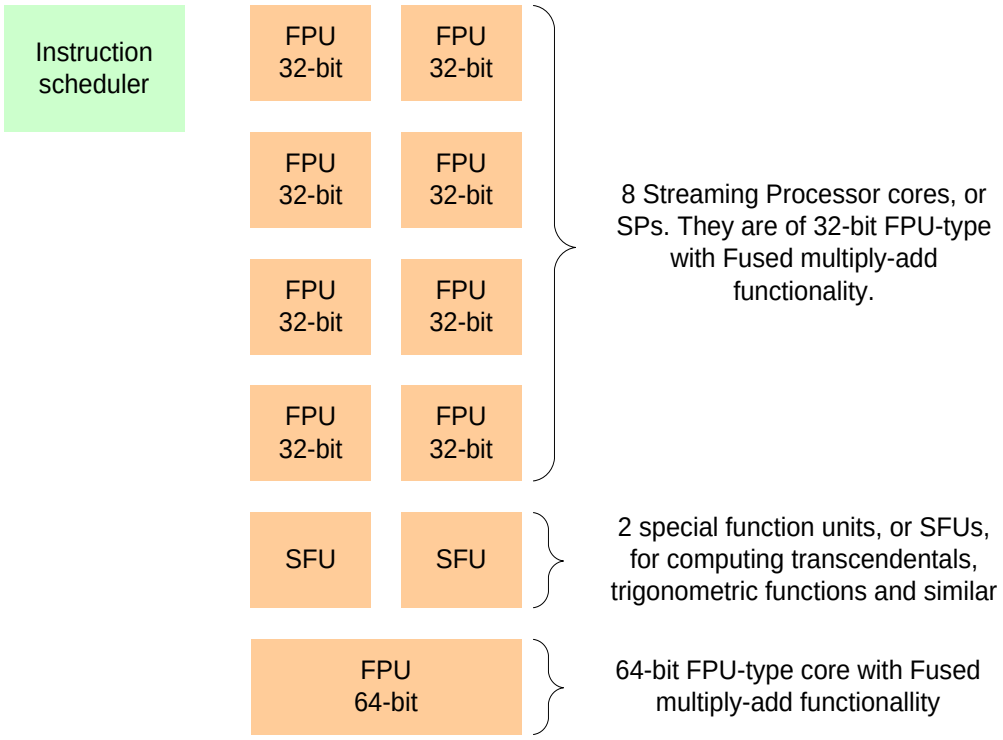

Figure 4 A schematic view of the instruction scheduler and the computational units that reside on an SM.

Note that we only have one 64-bit FPU per SM compared to 8 32-bit ones. This leads to significant slowdown when performing double precision operations; generally 8 times slower than when working with single precision [15].

Each of the 8 SPs executes the same instruction 4 times, once on their respective thread of each quarter of the warp, as shown in Figure 5 below. Hence, a warp's 32 threads are fully processed in 4 clock cycles, all with the same instruction but on different data.

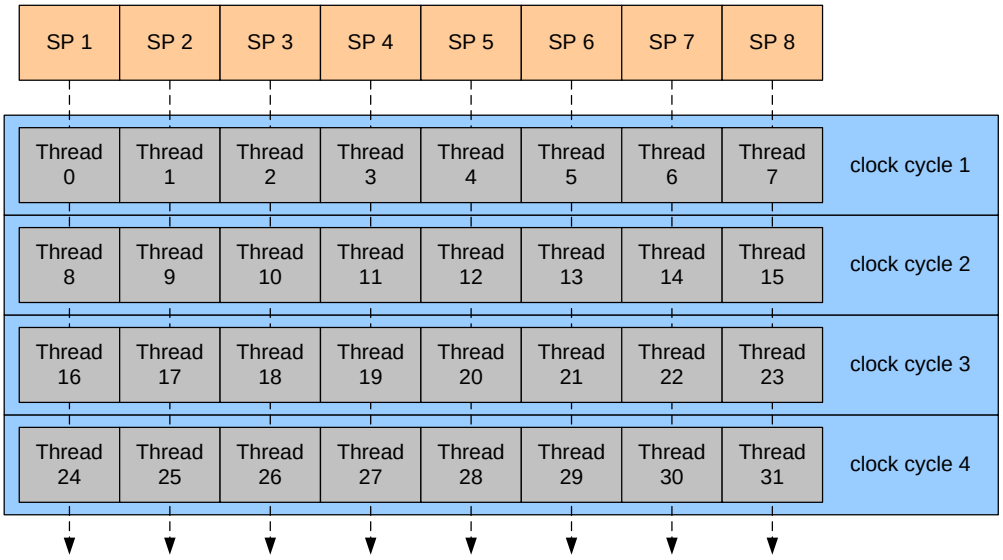

Figure 5 Warp execution on an SM.

7.3 CUDA memory architecture

The memory of all CUDA enabled devices consists of five different types; *global memory* (device RAM) accessible by all SMs, *shared memory* (similar to an L2 on-chip memory of CPUs) and *registers*, both of which are local per SM, and also *texture cache* and *constant cache*, which are a bit more complicated. There is also a type of memory used for register spilling, called *local*, which resides in global memory. Their layout on the device and their respective sizes and latencies are shown in Figure 6 and Table 3 below respectively.

Prepared (also subject responsible if other)		No.		
SMW/DD/GX Jimmy Pettersson, Ian Wainwright		5/0363-FCP1041180 en		
Approved	Checked	Date	Rev	Reference
SMW/DD/GCX Lars Henricson	DD/LX	2010-01-27	A	

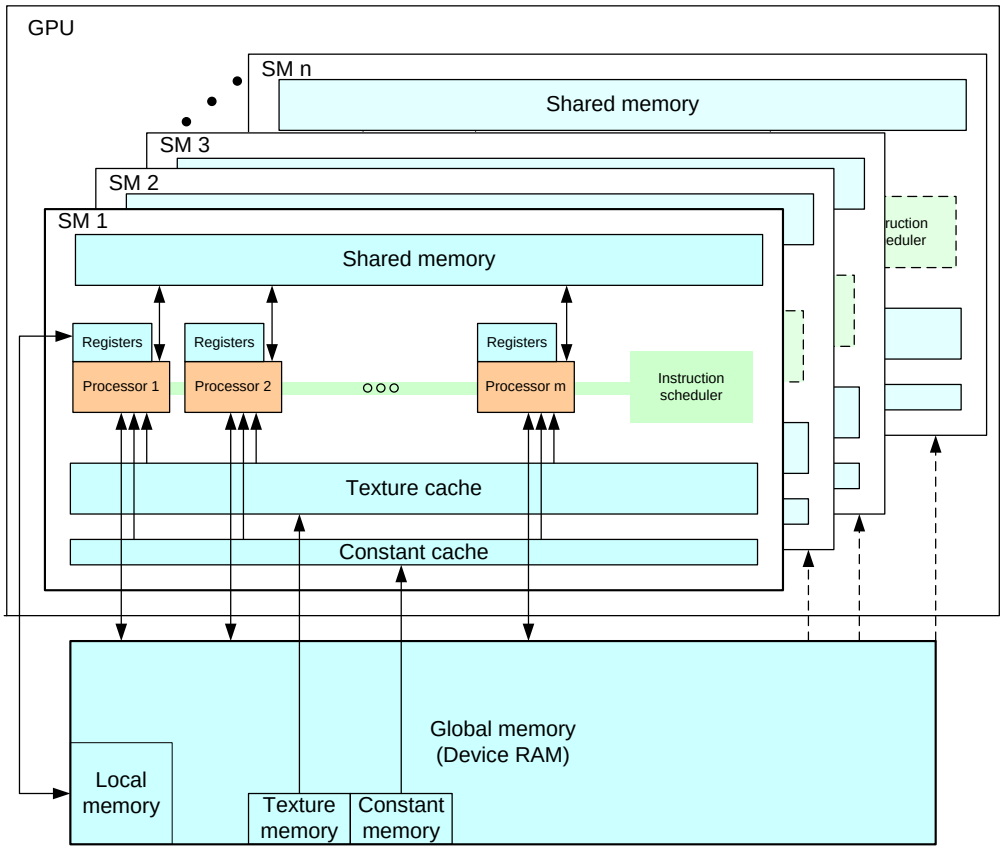

Figure 6 A schematic view of the device.

The actual memory structure is more complicated than this illustration. Also, the registers are actually one large chunk of registers per SM, not separate as in the figure above, though as they are thread local, they have been split into separate parts in the figure to illustrate this fact.

Memory	Location	Size	Latency (approximately)	Read only	Description
Global(device)	off-chip	1536 MB / Device	400-600 cycles	no	The DRAM of the device is accessed directly from the CUDA kernels and the host.
Shared	on-chip	16KB / SM	Register speed (practically instant)	no	Shared among all the threads in a block.
Constant cache	on-chip	8 KB / SM	Register speed (practically instant), if cached	yes	The constant cache is read only and managed by the host as described in 7.3.4
Constant memory	off-chip	64 KB / Device	400-600 cycles	yes	
Texture cache	on-chip	8 KB / SM.	> 100 cycles	yes	The texture cache is spread across the Thread Processing Cluster (TPC) as described in 7.3.5
Local(device)	off-chip	up to global	400-600 cycles	no	Space on global memory that takes care of register spills. Avoid at all cost!
Register	on-chip	64KB / SM (16K 32-bit)	practically instant	no	Stores thread local variables.

Table 3 Different properties of the various types of memory of the FX4800 card.

The different types of memory have various sizes and latency among other differences. Having a good understanding of the global, shared memory and the registers is essential for writing efficient CUDA-code; the constant and texture memories less so. The local memory only suffices as a last resort if too many registers are used and should not be a part of an optimized application.

Data between host and device, and also internally on the graphics card can be transferred according to Figure 7 below.

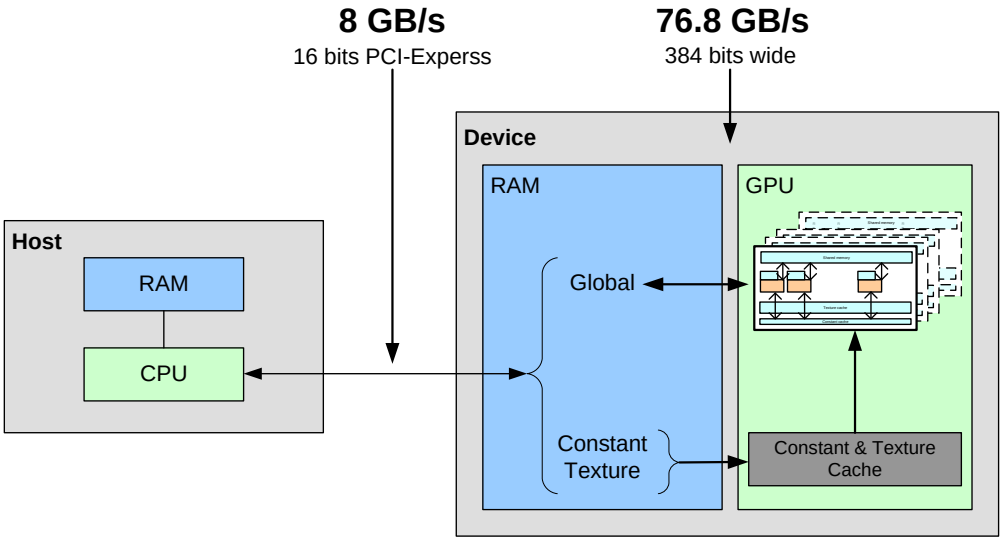

Figure 7 Memory access patterns.

As shown above, data can be transferred between the host to the device's RAM, where it is placed in either regular global, constant or texture memory. From there, the threads can read and write data. Notice that different SM cannot write to each other, nor can they write to constant or texture memory as they both are read only.

7.3.1 Global memory

Global memory, often called device memory, is a set of large off-chip RAM, measuring 1568 MB in the FX 4800. Being off-chip, it has a latency of roughly 400-600 clock cycles, with a bandwidth of typically around 80 GB/s. Global memory can be accessed by all threads at any time and also by the host, as shown in Figure 7 above.

7.3.1.1 Coalesced memory access

In order to utilize global memory efficiently, one needs to perform so called coalesced memory reads. This is the most critical concept when working with global memory and it comes from the fact that any access to any memory segment in global memory will lead to the whole memory bank being transferred, irrespective of how many bytes from that segment are actually used. It should also be mentioned that achieving coalesced memory reads is generally easier on devices with compute capability 1.2 and higher.

Coalesced memory transfers entail reading or writing memory in segments of 32, 64, or 128 bytes per warp, *and* aligned to this size [16]. Each thread in a warp can be set to read a 1, 2, or 4 byte word, yielding the mentioned segments as a warp consists of 32 threads. However, the actual memory access and transfer is done a half-warp at a time; either the lower or upper half. An example of a coalesced memory access is illustrated in Figure 8 below:

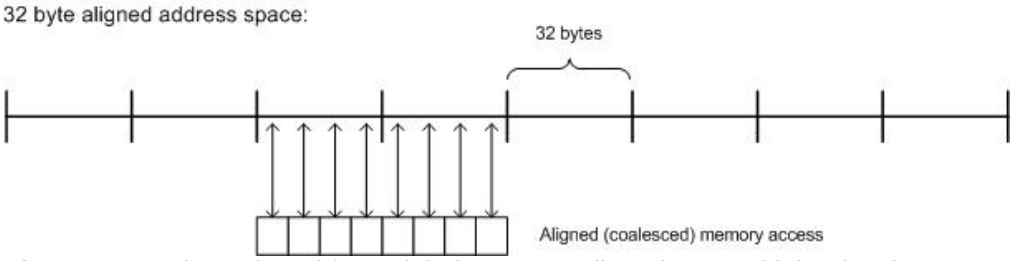

Figure 8 A coalesced read from global memory, aligned to a multiple of 32 bytes.

Above, all threads are accessing the same contiguous memory segment aligned to 32 bytes. All threads access one element each, and hence this is a coalesced read and the whole memory segment will transfer its data in one operation.

As global memory can be considered to be divided into separate 32, 64 or 128-byte segments depending on how it is accessed, it is easy to see why only reading the last 8 bytes of a memory segment would give rise to wasted bandwidth, as this will still leads to a transfer of a whole 32-byte memory segment. In other words, we are reading 32 bytes of data but only using 8 bytes of effective data. The other 24 bytes are simply wasted. Thus we have an efficiency degradation by a factor of 32 / 8 = 4 for this half-warp. A non-coalesced access is illustrated in Figure 9 below.

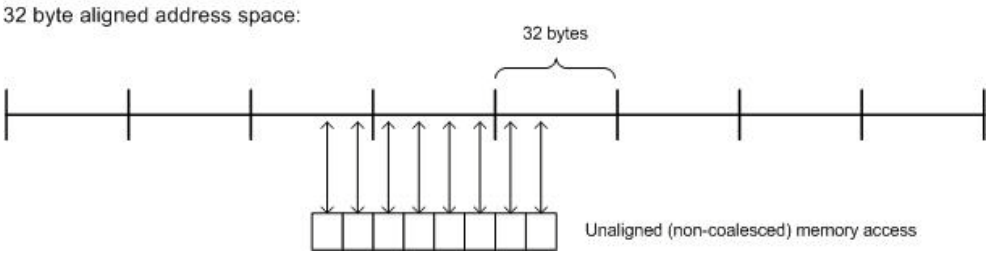

Figure 9 A non-coalesced read from global memory.

Even though only a small part of the left and right memory segment are accessed in Figure 9, all three segments are fully transferred which wastes bandwidth as all elements are not used.

Because of the memory issues explained above, a common technique to achieve even 32, 64 or 128-byte multiples is to perform padding of the data. This can be performed automatically through the CUDA API when writing to global memory. As global memory bandwidth is often the bottleneck, it is essential to access global memory in coalesced manner if at all possible. For this reason, computations with very scattered access to global memory do not suite the GPU.

7.3.2 Shared memory

Shared memory is a set of fast 16KB memory that resides on-chip, one set per SM. In general, reading from shared memory can be done at register speed, i.e. it can be viewed as instant. Shared memory is not cached, nor can it be accessed by any threads that reside on other SMs. There is no inter-SM communication. To allow for fast parallel reading by a half-warp at a time, the shared memory is divided into 16 separate 4-byte banks. If the 16 threads of a half-warp each accessed different bank, no issues occur. However, if two or more threads access the same memory bank, a *bank conflict* occurs as described below.

7.3.2.1 Bank conflicts

A *bank conflict* occurs when two or more threads read from the same memory bank at the same time. These conflicts cause a linear increase in latency, i.e. if there are *n* bank conflicts (n threads accessing the same bank), the read or write time to shared memory will increase by a factor *n* [17].

To avoid these bank conflicts, one needs to understand how memory is stored in shared memory. If an array of 32 floats is stored in shared memory, they will be stored as shown in Figure 10 below.

```
__shared__ float sharedArray[32];
```

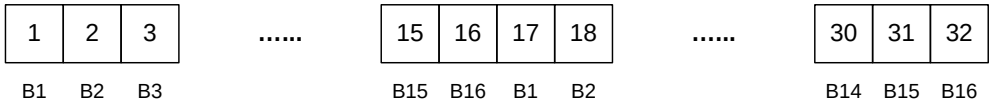

Figure 10 Shared memory and bank conflicts.

In Figure 10, ‘B1’ and ‘B2’ etc refer to shared memory bank number one and two respectively.

Looking at Figure 10, we can see that both element 1 and element 17 in our float array are stored in the same shared memory bank, here denoted as 'B1'. This means that if the threads in a warp try to access both element number 1 and element number 17 simultaneously, a so called bank conflict occurs and the accesses will be serialized one after another.

One way of storing the array would be to place element 1 in bank number 1, element 2 in bank 2, and so on. Notice that both element *i* and element *i+16* are in the same bank. This means that whenever two threads of the same half-warp accesses elements in shared memory with an index *i* and index *i+16* respectively, there arises a bank conflict.

Bank conflicts are often unavoidable when accessing more than 8 doubles in shared memory at the same time. As a double consist of 8 bytes, it consumes two 4-byte memory banks. Due to the SIMD-like architecture of CUDA, it is likely that both thread number 1 and 9 will be accessing elements 1 and 9 at the same time and hence give rise to a bank conflict, as shown in Figure 11 below.

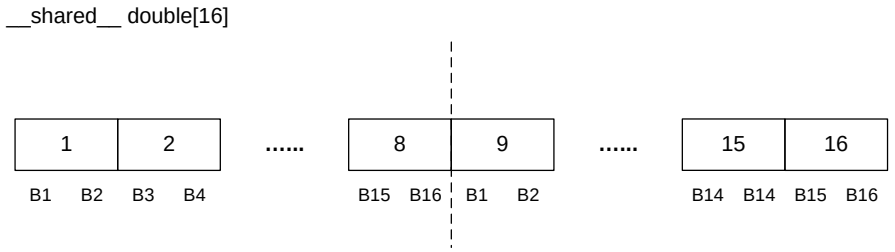

Figure 11 Bank conflicts with doubles.

However, shared memory also has a feature known as “broadcast”, which in some cases acts as a counter to the problem of bank conflicts. The broadcast feature lets one and only one shared memory bank be read by any number of the 16 threads of a half-warp. Thus, if for instance threads 0-7 read banks 0-7 respectively, and threads 8-15 all read bank 8, then this will be a 2-way bank conflict as it will first entail a simultaneous read instruction by threads 0-7, followed by a broadcast read instruction by threads 8-15. The broadcast feature is operated automatically by the hardware.

7.3.3 Registers and local memory

In the GT200 series, each SM has 16K 4-byte registers. These have an access time that can be viewed as instant. The most common issue with the registers is that there aren't enough of them. If one exceeds the limit of 16K 4-byte registers by for example trying to store too many variables in the registers, some register will spill over into local memory as determined by the compiler at runtime. This is highly undesirable since local memory is stored off-chip on global memory. This means that the latency is around 400-600 clock cycles, just like the rest of global memory. Thus it is highly recommended that one either decreases the number of threads sharing the registers so as to allow more registers per thread, or tries to make more use of shared memory so as to put less strain on the registers.

Another issue with registers are that they may also be victims of banking conflicts similar to shared memory, and also to read-after-write dependencies. The issue of read-after-write dependencies is generally taken care of by the compiler and instruction scheduler if there are enough threads. They can accomplish this as approximate waiting time for read-after-write dependencies is 24 clock cycles, i.e. 6 warps of instructions. If at least 6 warps are active, i.e. at least 192 threads, the compiler and instruction scheduler execute the threads in a round robin-type fashion, effectively eliminating and read-after-write dependencies. As the maximum total number of threads per SM is 1024, this leads to an optimal thread occupancy that is at least 192 / 1024 = 18.75%.

7.3.4 Constant cache

The constant cache is an 8 KB of read only (hence constant) cache residing on each SM, as seen in Table 3. Reading from the constant cache is as fast as reading from registers if all threads read from the same address if no cache misses occur. If threads of the same half-warp access *n* different addresses in the constant cache, the *n* reads are serialized. If a cache miss does occur, the cost will be that of doing a read from global memory. This is in some sense the reverse of the shared memory bank conflicts where it is optimal to have all threads reads from different banks.

7.3.5 The texture cache

The Texture cache is an 8 KB read only cache residing on each SM and is optimized for 2D spatial locality. This means that threads in the same warp should read from memory that is spaced close together for optimal performance. The GT200 series groups its SMs into TPCs (Texture Processing Clusters) consisting of 3 SMs each. Thus, we have 24 KB shared texture cache per TPC.

The *texture cache* is slower than the constant cache but generally faster than reading directly from global memory. As indicated before, the texture fetches have good locality which means that the threads in a warp should read from close-by addresses. This essentially means that the texture cache can sometimes act similar to a more traditional CPU cache, meaning that fetching from global memory via the texture cache can sometimes be beneficiary if data is reused. This is particularly so when random accesses to global memory is required. Remember that for good performance, the constant cache requires that we fetch memory from the same address while reading from global should be performed in a coalesced manner. In situations when this is not suitable, the texture cache can be faster.

The texture cache is also capable of linear interpolation between one, two or three dimensional arrays stored in the texture cache while reading data from them.

7.3.6 Host-Device communication

The bandwidth between host and device is approximately 8 GB/s when using PCI-express. These types of transfers should be kept to a minimum due to the low bandwidth. In general, one should run as many computations as possible for the transferred data in order to minimize multiple transfers of the same data.

One way of hiding the low bandwidth is to use asynchronous memory transfers that enable data to be transferred between the host and the device while at the same time running a kernel.

7.4 Implementation details

One way to get an idea of how the micro architecture maps to the CUDA API is to look at the threads and how they seem to relate to the SMs. We have an inner high clock frequency layer of 8 SPs and 2 SFUs each capable of reading their operands, performing the operation, and returning the result in one inner clock cycle. The 8 SPs only receive a new instruction from the instruction scheduler every 4 fast clock cycles [18]. This would mean that in order to keep all of the SPs fully occupied they should each be running 4 threads each, thus yielding a total of 32 threads, also known as a warp. The programming aspects of warps are discussed in depth in the section 9.5.

Why exactly the SPs are given 4 clock cycles to complete a set of instructions, and say not 2 making a warp consist of 16 threads instead, is difficult to say. Nvidia has given no detailed information on the inner workings of the architecture. It could be either directly related to present hardware constraints or a way of future-proofing. Some theories as to why a warp is 32 threads are discussed in the rest of this section.

One theory as to why a warp consists of 32 threads is that the instruction scheduler works in a dual pipeline manner, first issuing instructions to the SPs and next to the special function units. This is done at approximately 600 MHz, half the speed of the SPs or SFUs. This would mean that if we had 8 threads running on the SPs, they would be forced to wait 3 clock cycles before receiving new instructions. Thus Nvidia choose to have 32 threads (8 threads/cc * 4 cc) running, keeping the SPs busy between instructions. This is illustrated in Figure 12 below.

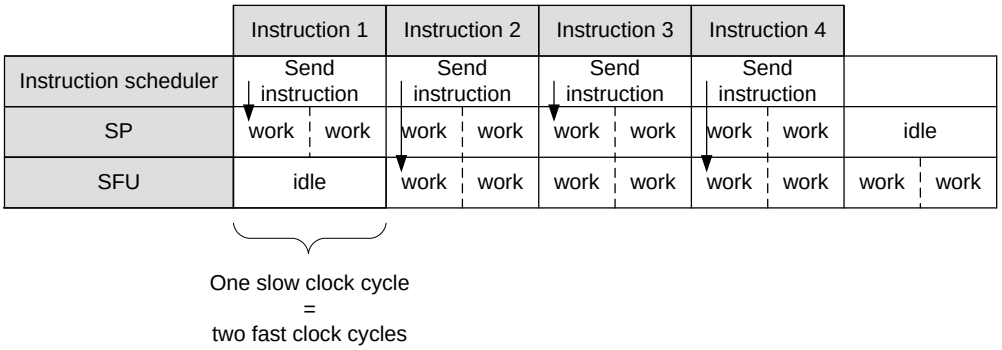

Figure 12 Scheduling of instructions.

This can also help to explain why the coalesced memory reads is performed in half-warps. To fetch memory, there is no need for the SFUs to be involved. Thus the instruction scheduler may issue new fetch instructions every 2 clock cycles, thus yielding 16 threads (a half-warp).

Another theory is that it is simply a question of future-proofing. Since it is fully possible for the SM to issue new instructions every 2 clock cycles a warp might as well have been defined as 16 threads. The idea is that Nvidia is keeping the door open to in the future have 16 SPs per SM in the micro architecture and then have the scheduler issue instructions twice as fast, still having 32 threads in a warp. This would keep CUDA code compatible with newer graphics cards while halving the execution time of the warp as a warp would be finished in 2 clock cycles instead of 4.

7.5 Precision and lack of IEEE-compliance

Certain CUDA functions have a reasonably large error, both when using the fast math library and when using regular single precision calculations. The limited precision in the present hardware reflects its still strong relation to graphics as single precision is acceptable when performing graphics for e.g. computer games. Some errors are seemingly more reasonable than others, such as the log(x), log2(x) and log10(x) all of which have 1 ulp[10] as their maximum error. The lgamma(x) function on the other hand has the slightly peculiar error of 4 ulp outside the interval -11.0001... -2.2637; larger inside. For a full list of supported functions and their error bounds, see the CUDA Programming Guide, section C.1.

Worth noting beside purely numerical errors is that CUDA does not comply with IEEE-754 binary floating-point standard treatment of NaNs and denormalized numbers. Also, the square root is implemented as the reciprocal of the reciprocal square root in a non-standard way. These and many other exceptions are detailed in the CUDA Programming Guide, section A.2.

[10] Unit of Least Precision

However, the coming Fermi architecture will solve many of the issues of precision as it is fully IEEE 754 – 2008 compliant as noted in 13.

Also worth noting when performing real time analysis of for instance radar data is that single precision is more than likely not the largest source of error, as this is rather the input signal, noise or similar. Hence, the lack of good double precision support is likely not an issue for radar signal processing and similar. However, for certain numerical calculations, such as QRDs of ill conditioned matrices, and simulations, the lack of good double precision support can be an issue. Again, the coming Fermi architecture will largely solve this precision issue as it will have greatly increased double precision support, having only twice as many single precision units as double precision instead of the present 8.

8 Nvidia Quadro FX 4800 graphics board

A picture of a Nvidia Quadro FX 4800 graphics board is shown in Figure 13 below.

Figure 13 A Nvidia Quadro FX 4800 graphics board.

Some specifications are shown in Table 4 below.

Memory	1.5 GB
Memory interface	384-bit
Memory Bandwidth	76.8 GB/s
Number of SPs	192
Maximum Power Consumption	150W

Table 4 Some specifications for the Nvidia Quadro FX 4800 graphics board.

9 CUDA Programming

9.1 A programmers perspective

CUDA uses a subset of ANSI C with extensions for coding of so called kernels; code that runs on the GPU. The kernel is launched from the host. As the CPU performs better than the GPU for purely serial code, whereas the GPU is often superior in parallel tasks, a typical application with both serial and parallel code might work as shown in Figure 14 below.

Prepared (also subject responsible if other)		No.		
SMW/DD/GX Jimmy Pettersson, Ian Wainwright		5/0363-FCP1041180 en		
Approved	Checked	Date	Rev	Reference
SMW/DD/GCX Lars Henricson	DD/LX	2010-01-27	A	

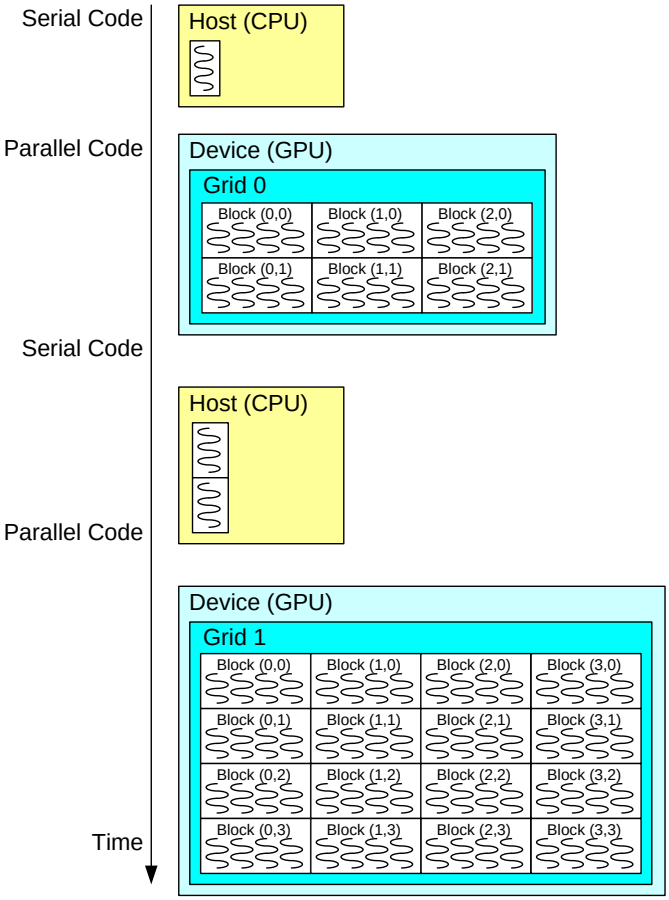

Figure 14 Typical allocation of serial and parallel workloads.

An example of how the host and device work to solve a part serial, part parallel problem. Serial code is run on the host, and parallel code is run on the device through so called kernels.

Writing CUDA implementations that achieve a few times application speedup is generally not too strenuous, but writing well-written CUDA-code in order to truly utilize the hardware is often not an easy task, as is the case with most parallel code writing. Even though regular IDEs such as Visual Studio can be used when programming and even though CUDA-code is as non-cryptic as C gets, one has to have a reasonably good understanding of the most important hardware units and features. Not understanding the hardware sufficiently can give performance differences on the order of several magnitudes.

The most important hardware parts to have a good understanding of are the three main kinds of memory, namely global memory (GPU-RAM), shared memory (on-chip scratch pad), the registers, and also the SIMD-like fashion the GPU executes instructions.

The types of memory that are directly accessible by the programmer, described in detail in section 7.3, are summarized in this table along with simple syntax examples.

Name	Syntax example	Size	Description
register	`float a;`	64 KB/SM	Per thread memory.
shared	`__shared__ float a;`	16 KB/SM	Shared between threads in a block.
constant	`__constant__ float a;`	8 KB/SM	Fast read-only memory.
texture	N/A	8 KB/SM	Global texture memory. Accessed through CUDA API.
global	`float* a`	>=1 GB	Accessed by passing pointers through the CUDA API.

Table 5 The types of memories and their syntax that are explicitly available to the programmer.

9.2 Programming environment

Debugging can at the time of writing be quite a tedious task. This is mainly due to the fact that one cannot as of yet “see inside” the GPU. One can put breakpoints inside the GPU-code, but these can only be observed in so called “Emulation mode” in which the CPU emulates the device. Sometimes, the emulated mode will run code in a desired manner, whereas the GPU does not. Finding errors in this case is not always an easy task. However, a tool known as Nexus has recently been released that allows for real debugging on the hardware and advanced memory analysis features.

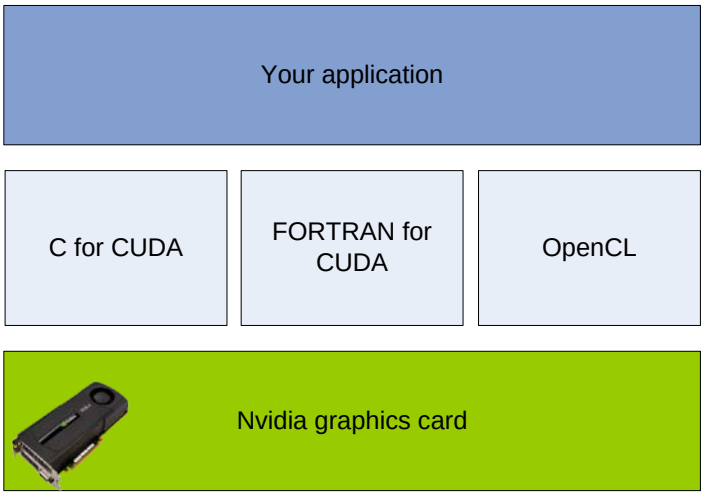

Figure 15 The CUDA platform.

The programming platform consists of a variety of API:s through which one may access the potential of the graphics card. The APIs that are available are C for CUDA which is basically C with some C++ extensions, FORTRAN for CUDA, and OpenCL (Open Computing Language). Most commonly used is C for CUDA which is what we've been using in this master thesis project.

The compilation process entails separating the device and host code. The host code is executed by the CPU and may be compiled using any regular CPU compiler. Meanwhile the code that should be executed on the device is compiled into a type of assembler code called PTX (Parallel Thread eXecution). This is as far as the programmer can really do any programming. In the next step the PTX assembler code is compiled into binary code called cubin (CUDA binaries). Figure 16 below shows this compilation process.

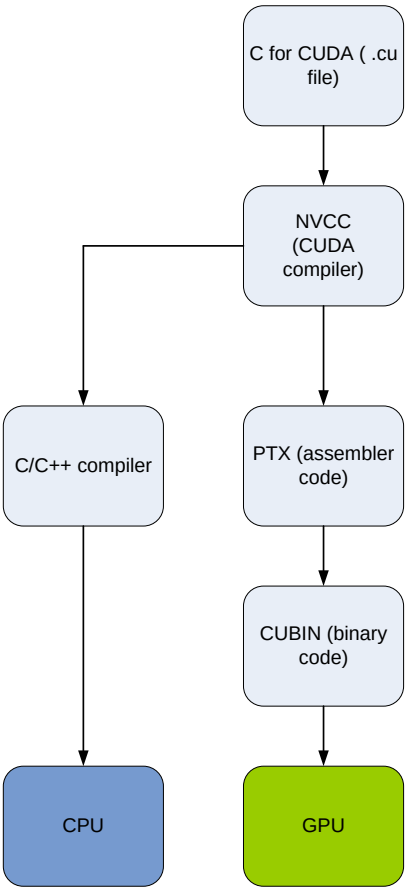

Figure 16 The NVCC compilation process.

9.3 CUDA-suitable computations

All computations and algorithms cannot reap the benefits of CUDA. To be CUDA-suitable, a computation must be very parallel, have high enough arithmetic intensity or AI, not be recursive, lack or have easily predicted branching, and also preferably have localized or known memory access patterns. The host will practically always win over the device in serial computations. The main reasons for this is first the relatively large latency between the host and device, and second the low clock frequency of the GPU's cores compared to the CPU's, all of which are described in 7.3.6. The latency in data transfers also gives rise to a demand in the magnitude of the problem. Computations that are finished on the host before the device has even begun to compute due to slow host-device data transfer will obviously be faster on the host. For instance, if one wants to add two large matrices together, the CPU will likely be faster at computing the addition. This is due to the fact that the short instruction execution time of the addition negates the benefit of having hundreds of cores and a very high off-chip bandwidth due to the low host-device bandwidth as the cores will mostly be idle waiting for data. If calculations are large and have a high enough arithmetic intensity, the device has a good chance of being much faster.

If an algorithm is dependent on recursion, it is not CUDA-suitable as recursion is not possible in CUDA. Also, if there is a lot of branching in a calculation, this can be an issue as described in 9.6.1. The problems of non-localized random memory access are described in 7.3.1, and a short detailing on AI is given in 9.4.

9.4 On Arithmetic Intensity: AI

One important matter to consider when understanding how an algorithm will map to a GPU architecture such as CUDA is the importance of arithmetic intensity, or AI. From Table 1 and Table 2, it is clear that the GPU has less bandwidth per FLOPS than a CPU has. If the AI of an algorithm is too low, the SMs will not receive data faster than they can compute, and they will hence be idle a large part of the time. However, if the AI is high enough, the low bandwidth / FLOPS ratio can be overcome. Part of the puzzle is shown in Figure 17 below.

Prepared (also subject responsible if other)		No.		
SMW/DD/GX Jimmy Pettersson, Ian Wainwright		5/0363-FCP1041180 en		
Approved	Checked	Date	Rev	Reference
SMW/DD/GCX Lars Henricson	DD/LX	2010-01-27	A	

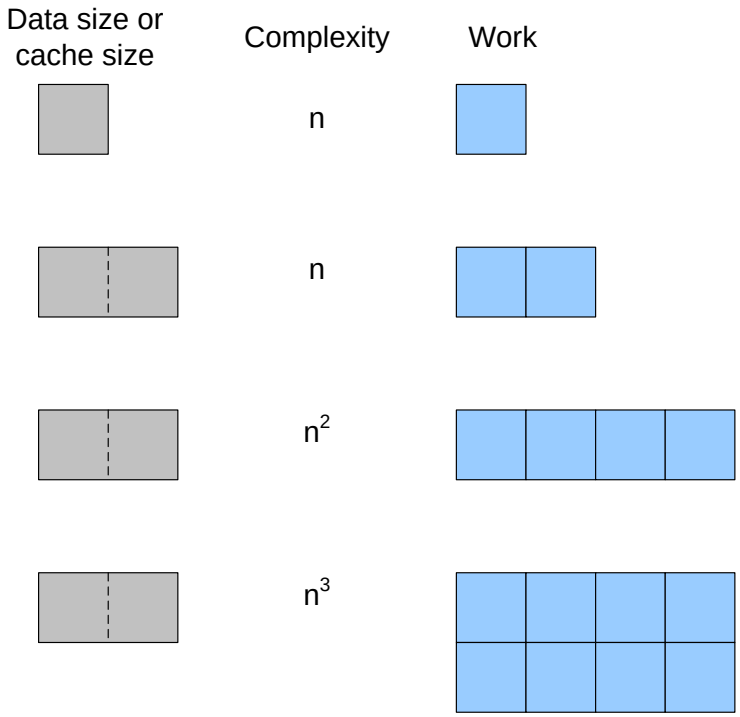

Figure 17 Arithmetic Intensity: AI.

In Figure 17, it is clear that if the complexity of a problem is say n^2, i.e. squared, and the data size is doubled, the amount of work is quadrupled. Notice that the data in the figure is represents on chip-data, not total global data. In other words, we are in fact considering **local AI**, not the global AI for the whole algorithm.

If both the performance and bandwidth are used at 100% for an *n* complexity algorithm, doubling the problem size only doubles the time needed for the computation, while still utilizing both potential performance and bandwidth at 100%. This is not so for a problem of n^2 complexity. Doubling the problem size will in this case lead to a quadrupling of the amount of work for the same bandwidth.

Worthy of note, the bandwidth of GPUs has not increased linearly with its computational potential. However, if the on-chip memory sizes increase instead, a linear performance-to-bandwidth increase is no longer necessary, at least not for high AI computations. The reason for this is that if the cache size is double for a quadratic algorithm, the amount of work quadruples as the amount of data on-chip doubles. Hence, for algorithms of quadratic complexity, a doubling of the potential compute performance must either be done with either a doubling of the cache sizes or a quadrupling of the bandwidth. Hence, though bandwidth does not keep up with potential compute potential in a linear fashion, this is only a serious issue for low AI algorithms, and hence high AI algorithms benefit more from the GPU than low AI ones.

9.5 Grids, blocks and threads; the coarse and fine grained data parallel structural elements of CUDA

Before we continue with how a workload is placed on the device, we need to point out a major difference between regular CPUs and GPUs regarding how threads work so as not to mystify the extreme number of threads used in CUDA. Threads on the device are very lightweight, lack any kind of stack, and are created and destroyed on the fly. Also, the management of threads is done solely by the thread scheduler, i.e. management of the threads is not done by the computing cores. Furthermore, all the SMs of the device can switch between threads instantly, without wasting any clock cycles in so called context switches. This differs significantly from regular CPUs where the creation, management and destruction of threads and also context switches all take time from the computing cores and cost clock cycles. The fact that the device can instantly jump between several threads makes it possible to let one warp say read data from global memory into the warp's registers, and instantly after having issued the read instruction, let the next warp execute instructions on data in say shared memory. This makes it possible to hide access time to device RAM as it is possible for some threads to do calculations while others wait for more data. On the GTX 260 there are 24 SMs each capable of having 1024 threads active at any time. In other words, it is possible to have 24576 threads active. The device will at runtime decide in which order it will execute the threads but will try to do so in a round robin fashion so that it always has threads that are ready to work on on-chip data, while other threads read or write data from global memory. Often it is good to issue one thread per work item, for instance for each addition in a vector addition. Hence, if for example two N elements long vectors will be added together, it might well be practical to create N threads, where each thread will work on a specific element of both vectors. After a thread has read the input from both vectors and placed the result back in global memory, the thread dies and another one takes its place, the new thread working on a different data element.

Hopefully, after the brief explanation of thread usage on the device, the following part will seem more natural.

As stated above, the device often prefers having several thousands of threads active at the same time to be able to reach maximum performance. Beside the several thousand active threads, several million threads should be waiting to be computed to reach an even higher degree of performance. This can be seen for instance in the results of the TDFIR benchmark in 15.1, where more threads give rise to a high performance. Basically, each SM can handle 1024 threads at any time. As we have 24 SMs, we can have a total of 24576 threads running simultaneously. However, if our blocks are of size 512, having only 48 blocks in total, i.e. 2 blocks per SM, does not make it possible to hide latency to global memory. This gives rise to the preference of having several blocks per SM. If we have say 80 blocks per SM, each consisting of 512 threads, we have a total of roughly one million threads. Having this many threads also makes it easier to hide the start-up time and the end time when the SMs are not fully loaded as the start-up and end a relatively smaller compared to the whole computation.

Though the exact organization of the threads is of less importance, it is necessary to understand how a grid, the computational workspace, is organized and its relation to inter-thread communication among other issues.

The threads are sorted into a hierarchical system that defines in what groups the threads are computed, consisting of a grid at the top, followed by blocks, warps and finally threads that make out the lowest level of the hierarchy. This produces a coarse grained structure of blocks and a fine grained structure of threads. We begin by looking into grids, blocks and threads and how they are related, before moving on to the last structural element, the warp, which we also have briefly mentioned earlier.

When a kernel, a set of code to run on the device, is initiated from the host, the host first defines a computational unit called a 'grid' that defines the computational workspace for the device. The grid may have up to two dimensions; x and y. Elements in a grid are made up of units called 'blocks'. These blocks may in turn have up to three dimensions; x, y and z. Elements in a block are made up of threads. In essence, we have a two dimensional grid built up by three dimensional blocks, in turn consisting of threads.

An illustration of a computational 2-dimensional grid consisting of 2-dimensional blocks, each in turn consisting of 96 threads is shown in Figure 18 below.

Prepared (also subject responsible if other)		No.		
SMW/DD/GX Jimmy Pettersson, Ian Wainwright		5/0363-FCP1041180 en		
Approved	Checked	Date	Rev	Reference
SMW/DD/GCX Lars Henricson	DD/LX	2010-01-27	A	

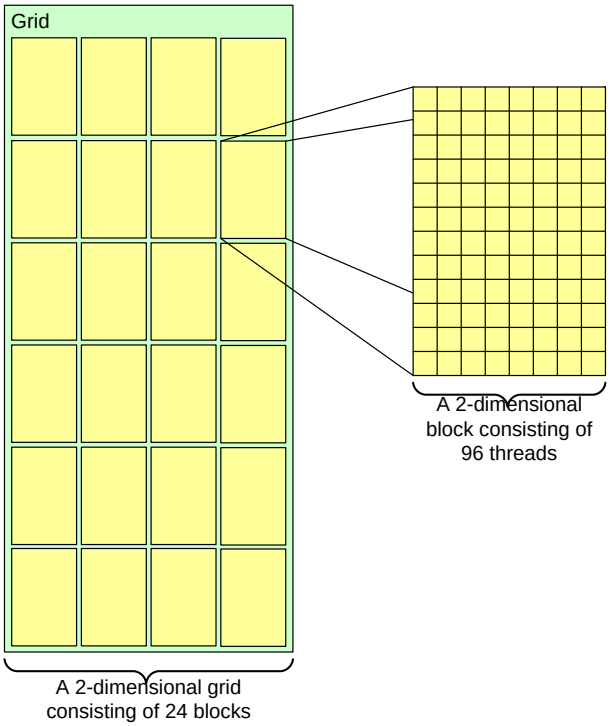

Figure 18 A 2-dimensional grid.

To better explain how a grid works, an example is given below.

For sake of argument, we start with an 8 elements long vector ‘A’, similar to a row in the block of Figure 18 above, written linearly in memory where we want to increment each element by 1. Typical CPU and CUDA implementations are given below.

CPU

```
for( i = 0; i < 8; i++)
     A[i]++;
```

CUDA

```
A[threadIdx.x]++;
```

Example Code 1 A comparison of incrementing each element in an 8 element long vector in both CPU and CUDA-fashion.

threadIdx.x above is pronounced ‘Thread Index x’, i.e. the index of the thread in the x-dimension. Where as the CPU will have one or possibly several threads that together go through all additions, CUDA will in this case have 8 threads that all will run the code in a SIMD fashion. threadIdx.x gives the thread its index offset in the vector ‘A’. The number of threads to be run has in the CUDA case been specified earlier by the host.

An extension of the addition example into a two dimensional matrix B[8][12], the block in Figure 18 above, the typical implementation might be

CPU

```
for( i = 0; i < 8; i++)
     for( j = 0; j < 12; j++)
          B[i * 12 + j]++;
```

CUDA

```
B[threadIdx.x +                         // Code line 1
  threadIdx.y * blockDim.x]++;          // Code line 2
```

Example Code 2 A comparison of incrementing an 8 by 12 matrix in both CPU and CUDA-fashion.

The CUDA-code has been split over two lines for sake of clarity. "blockDim.x" is pronounced "Block Dimension x", i.e. the size of the block in the x dimension which has, again, been specified earlier by the host and in this case is equal to 8, the number of threads in the x dimension for the block. "Code line 1" gives the thread its index offset in the x-dimension and "Code line 2" gives the thread it's index offset in the y-dimension, the length of a row (i.e. blockDim.x) multiplied with the number of rows down (i.e. threadIdx.y). The sum of the offsets gives the thread its individual index. Notice also the lack of for-loops in the CUDA-code. Again, one thread is in this case created per addition. It is important to understand that the layout within a block is in this example purely for the programmer's abstraction and has no influence on performance at all. The example could well have been ordered such that a block consisted of 2 rows of 16 elements, for instance. Some of the finer details of a block are described further down.

To finish off our explanation of how grids, blocks and threads are related, we will attempt to increment each element in the matrix in Figure 18 above.

CPU

```
for( i = 0; i < 8 * 4; i++)
     for( j = 0; j < 12 * 6; j++)
          C[i * 8 * 4 + j ]++;
```

CUDA

```
C[threadIdx.x +                                    // Code line 1
threadIdx.y * blockDim.x +                         // Code line 2
blockIdx.x *  blockDim.x * blockDim.y +            // Code line 3
blockIdx.y *  blockDim.x * blockDim.y * gridDim.x]++;       4
```

Example Code 3 A comparison of incrementing a 32 by 72 matrix in both CPU and CUDA-fashion**.**

"Code line 1" and "Code line 2" serve the same function as in the block-sized matrix, i.e. gives the thread its offset within the block. "Code line 3" serves the same function as "Code line 1" but in the grid-perspective, i.e. it gives the value of the offset due to the threads to the left in the grid on the same block row. "Code line 4" serves the same purpose as "Code line 2", i.e. it gives the value of the offset due to the threads above a thread in the grid. The sum of the offsets gives the thread its individual index and also its address in the array's memory. This is illustrated in Figure 19 below, where the sum of the 4 code lines above contribute to the offset of a certain thread.

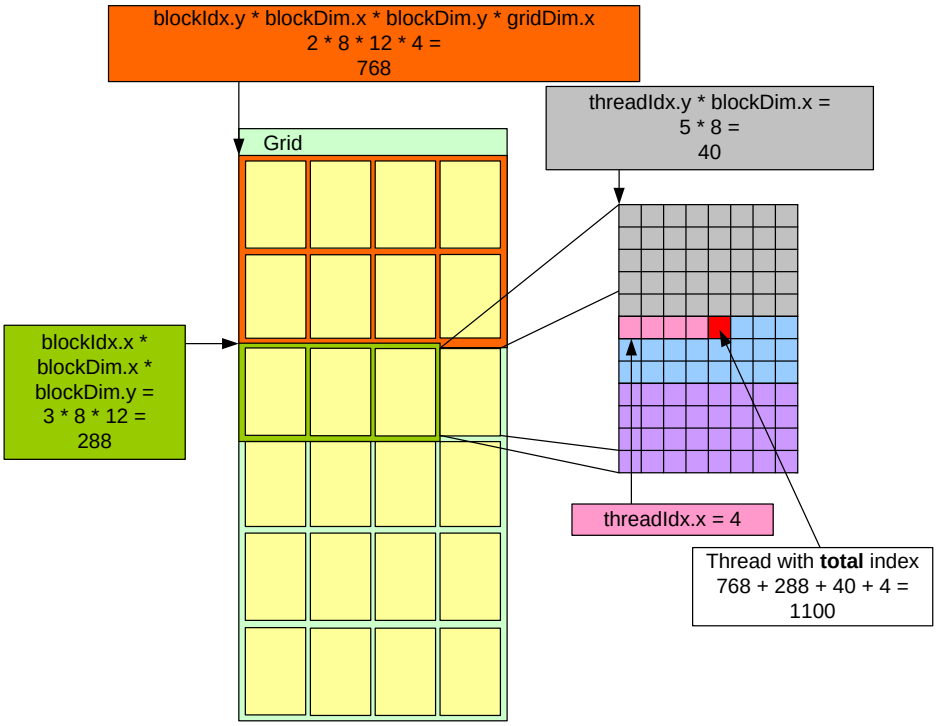

Figure 19 Computing the offset.

Now that the basics of grids, blocks and threads have been explained, we move on to their finer details.

The configuration of the grid structure, mainly the size and layout of blocks, was largely irrelevant in the above example, but is sometimes important. The importance is mainly related to thread communication and memory limitations. Finding the optimal or even a good grid structure can sometimes be difficult and one often has to try various configurations to see which one is fastest. The hardware limits regarding the size of blocks and grids and also communication between threads and their relation to the blocks are detailed in 9.6 below.

9.6 Hardware limitations

The maximum sizes of the grid and blocks that can be declared are not arbitrary. Their maximum sizes are

- (65535, 65535) blocks per grid
- (512, 512, 64) threads per block

for the latest compute capability 1.3. Also, a block may not include more than 512 threads, in other words no more than 8 warps per block as a warp always consists of 32 threads. Warps are handled in detail in 9.6.1 and only the necessary for our present perspective is noted here.

A warp is chosen to consist of threads in sequential order, such that if a block has 64 threads, the first warp will consist of threads 0-31 and the second warp of threads 32-63. If a block has 70 threads, the first warp will consist of threads 0-31, the second warp of threads 32-63, and a third warp will consist of threads 64-95. Even though threads 70-95 have no part in the computation as only the first 70 threads are used, they will still count as active threads and be counted to the total number of threads in any circumstance, including the maximum number of threads for a block. Thus, one should always try to use blocks based on multiples of 32 threads.

Besides the constraints of how big a computational grid may be configured, there are other hardware limits that also apply.

An SM can have a total of 1024 active threads at any given time, which is the same as 32 active warps, and for optimization reasons explained in 9.5, the more threads the merrier. We could, for example, have a grid where blocks consist of 16 warps each, i.e. 16*32 = 512 threads. This would mean that an SM would have two blocks active as 512*2 = 1024, the maximum number of threads allowed. If we on the other hand have blocks consisting of 480 threads, i.e. 15 warps a block, we run into some issues. We can still only fit 2 blocks onto an SM; 3 blocks would mean 480 * 3 = 1360 threads, clearly exceeding the maximum allowed of 1024 per SM. This would mean we only have 480*2 = 960 threads active. Leaving other considerations aside, more threads are usually better and hence blocks consisting of 16 warps are probably better than blocks consisting of 15 warps.

The curios reader might ask why we have not chosen blocks of a different size, say 64 threads a block and then simply have 16 blocks per SM, again giving us 1024 active threads. The answer is that we could well have, but due to further limitations described below, the optimal size of the block is far from obvious. Also, only 8 blocks are allowed per SM at any one time.

A block may not terminate until all its warps have finished all their calculations for all threads. Hence, situations where only a few warps are unfinished may give rise to efficiency issues as the rest of the warps are idle. The most extreme example of this is if we, as in the first of the two alternatives above, choose 512 threads in one block, i.e. 16 warps. If only one of the 16 warps is unfinished, the other 15 warps will be idle, unable to do anything until the last warp has finished. If this were to occur in both blocks at the same time, the SM would have only 2 * 32 = 64 threads active, far from the maximum of 1024. The active reader might now have come to the conclusion that blocks should only consist of one warp, as this would minimize the consequences if some warps for some reason need a longer time to compute than the rest. The faster blocks will finish, giving room to new active warps, while the ones taking a long time to compute will simply take the time they need without blocking other warps unnecessarily, a completely correct analysis so far.

However, there are still more limitations and issues to consider.

Threads are only able to communicate with other threads if they are within the same block, and do so via shared memory, so for sake of thread-communication, we want to increase the size of the blocks as much as possible. All threads have the ability to read and write to global memory, but the execution order of the blocks is arbitrary, and hence one cannot expect threads of different block to be able to communicate with threads of other blocks.

As mentioned earlier, more threads in a block are not always better due to the limited amount of memory, and it is often not be preferable to have 1024 active threads because of this, as described below.

An SM has 16K 32-bit thread local registers and 16 KB of on-chip shared memory as shown in Table 6 below and both of which are described in detail in section 7.3.3 and 7.3.2 respectively. Their limited size adds to the difficulty in designing the computational grid.

If each thread in a block only uses 16 32-bit registers each, then there are no issues with the registers as 1024 active threads * 16 32-bit register per thread equal 16K 32-bit registers. However, if each thread uses say 20 registers, some issues will occur. In this case, we cannot run 1024 active threads on the SM as this would mean a total of 20 * 1024 = 20480 32-bit registers, clearly exceeding the allowed 16K number of registers. Instead, the maximum is 16K / 20 = 819 active threads. This must influence the choice in choosing the size of the blocks.

Resource	**Limits**
Maximum active threads per SM	1024 threads
Maximum threads per block	512 threads
Threads per warp	32 threads
Maximum Blocks per SM	8 blocks
32-bit registers per SM	16K
Shared memory	16 KB

Table 6 Some of the limits of the SM.

Furthermore, the limited amount of shared memory must be taken into account. The whole SM must share the 16 KB of shared memory. If the blocks use too much, fewer blocks will be able to run simultaneously similarly to the situation of excessive register use described above.

Also to keep in mind in all this is that we preferably want several blocks running on each SM so that the SM is not idle in between block creating and destroying blocks. If this is achieved, when one block is destroyed to give room for a new block, the other currently active blocks will be occupied with computations while the new block will likely read data from slow global memory. If there is enough work to be done with the other blocks, the latency of reading from global memory for the new can in essence be hidden as the SM is busy calculating the former blocks.

Beside the communication and memory size related issues above, there are other parts necessary for a programmer to understand regarding CUDA memory architecture. They are described in detail under 7.3.

Beside grids, blocks and threads, warps also act as a building block of structuring computations in CUDA and have limitations and issues of their own which are detailed below.

9.6.1 Thread execution

One issue to consider is that all threads of a warp must execute the same instruction. This is what is usually known as SIMD and Nvidia calls SIMT. Referring to our earlier increment example where each thread received its own index and hence address, the SM will execute the same instruction on every thread in a warp, each thread having different operand addresses according to its position in the grid. This inflexibility is a serious problem if branching occurs, i.e. when different threads need to go different instruction paths, as the example below illustrates.

```
if (X is true)
    execute Branch A
else
    execute Branch B
```

Example Code 4 An example of branching. Branching can severely hamper performance if it occurs within a warp.

This might seem to contradict the fact that all threads execute the same instruction, but it does in fact not. If all threads choose the same branch, no issues occur. However, if X is true only for some threads, these threads will execute instructions along branch A and the rest will execute instructions along branch B; the warp is branched. A branched warp is more commonly called a divergent warp. If a warp branches, the branches will be executed in series. This is due to the SIMT architecture which only permits one instruction to be executed per warp at a time. If a warp is branched N times and all branches have the same workload, then the performance of the computations is lowered by a factor of N within the warp. In the case above, all threads of branch A will be idle while branch B executes. Even though there might only be a handful of threads executing along branch B, the warp will still take 4 clock cycles to execute each instruction. When branch B has finished executing, branch A will execute and the threads of branch B will be idle. Having threads sit idle is very wasteful and should be avoided where possible.

To make the situation worse, a warp cannot be terminated until all of its 32 threads have finished, similar to how a block cannot terminate until all its warps have finished. Only after all 32 threads have finished may the warp terminate and give room to new warps. So in theory, a single branching thread can occupy a whole warp, which in turn can occupy a whole block.

9.7 Synchronization and execution order

When working with any parallel programming, syncing correctly is vital. CUDA has two ways of syncing; inter-block-wise and kernel-wise.

Inter-block-wise syncing is simply controlled by calling the function *__syncthreads()*. At this point in the code, all threads within a block will wait until all threads have reached the sync before any threads are allowed to continue.

Kernel-wise syncing is implemented simply by having separate kernels for separate parts of a larger computation different parts must be run in series. As only one kernel can run at a time, processes can easily be made to compute in series as separate kernels and calling them from the CPU in series.

However, there is no way in which to order the execution of threads, warps or blocks, even with using syncing. No code must be dependent on the order of execution. Block 0 can, for instance, be the first block to be computed just as likely as is any other block in the grid, and the same goes for warps in a block. An example of how an SM with room for four blocks each consisting of four warps might be scheduled is illustrated in Figure 20 below.

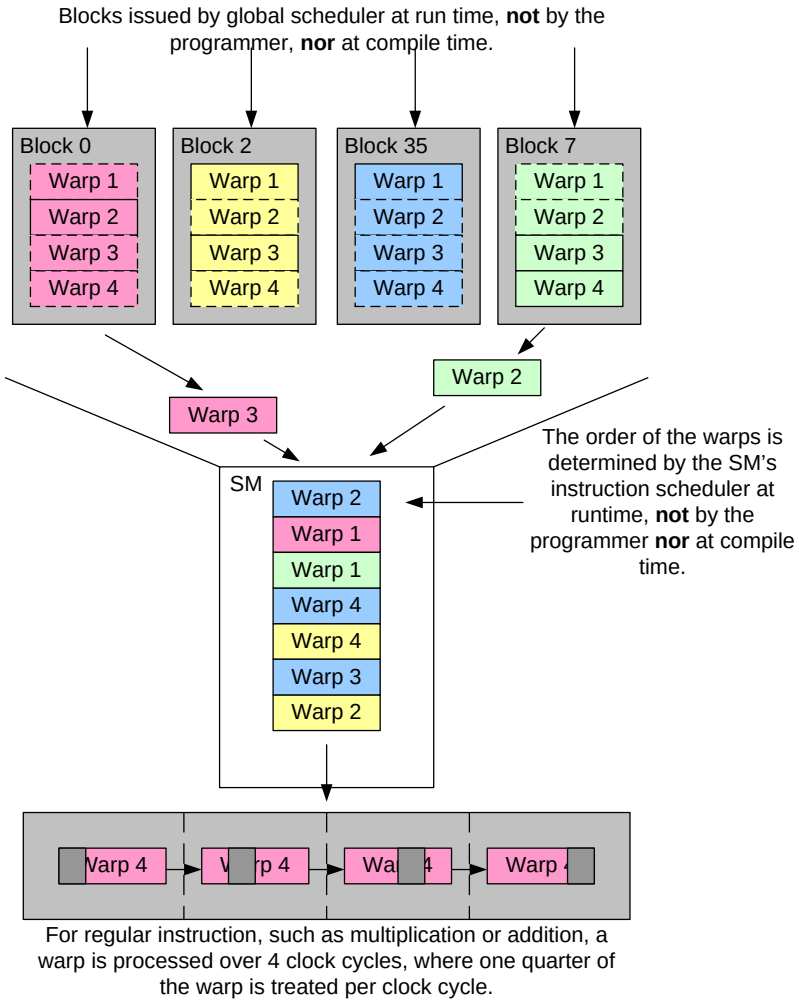

Figure 20 The indeterminist behaviour of the global and SM schedulers.

The global scheduler determines in Figure 14 assigns blocks to each SM. The SM then in turn determines in which order the warps of the blocks are computed. The ordering of blocks, the SM they are computed on, and the ordering of which the block on each SM are computed, are, for all practical purposes, arbitrary.

9.8 CUDA portability between the GT200 cards

All Nvidia graphic cards of the GT200 series have the same hardware components, namely global RAM memory, a certain number of SMs and a bus between the RAM and the SMs. The only difference between various cards is the amount of RAM, the number and clock frequency of the SMs, and the bus bandwidth. As CUDA code is written with the SMs limitations in mind rather than a specific card, programs will scale very well between different cards as they all have identical SMs, save for the clock frequency. For the current architecture all SMs have the same number of SPs, SFUs and double precision units, the same amount of shared and register memory and constant and texture cache, and also the same maximum number of blocks and threads. So if one for instance wants more performance or use less power, no changes have to be made to the software. It's simply a matter of changing the graphics card.

9.8.1 Current code running on future hardware

After detailing the current hardware, it should be noted that code written for any of the three as of yet known architectures scales without problem to the newer architectures as the hardware limitations only increase in size, as is shown in Table 7 below.

Architecture	Register per SM	Shared memory per SM	Native integer size	Maximum number of threads per block
GT80	8192	16	24-bit	768
GT200	16384	16	24-bit	1024
GF100 Fermi	32768	16-48	32-bit	1536

Table 7 Some of the differences in the different CUDA enabled architectures.

For instance, code written with the amount of register per SM in oldest CUDA enabled series in mind, the GT80, will run just as well on the present GT200 and future GF100 series architectures. The reverse is not true. In other words, there is no reason to fear that hardware optimized code will have difficulties running smoothly on future architectures, though there might of course be room for new optimizations based on the new architecture.

The same scalability does generally not apply to CPUs where some CPUs of an architecture will have an L3 cache where as other CPUs of the same architecture may not. It is hence not possible to "write once – run fully optimized everywhere" on regular CPUs, but is so for CUDA so far. In other words, optimizing code to a large extent according to the present hardware capabilities should not give rise to any issues in future hardware. This makes it possible to simply upgrade ones hardware, and still have old code run without needing to rewrite the old code. Optimizations to better suit the new hardware, however, will often be possible.

10 CUDA Optimizing

When writing efficient CUDA code, it is essential to know how to best exploit the benefits of CUDA but also to avoid its weaknesses. The following section details coding and optimization guidelines, starting with the highest priority guidelines and then working down towards the lower prioritized ones.

10.1 High priority optimizations

The highest priorities when designing and optimizing are

- find ways of parallelizing the algorithm
- ensuring global writes are coalesced whenever possible as described in 7.3.1.1
- avoiding different execution paths as described in 9.6.1
- minimizing the amount of transfers between on-chip and off-chip
- the data transfers between the host and device.

Details regarding the latter two are described below.

First, the amount of transfers between on-chip and off-chip should be minimized. One way to do this is to use the shared memory as a sort of manually managed cache so that data from global is read onto the chip only once and then reused by the threads on the SM. Another, less intuitive consequence of trying to minimize global reads is that it can often be better to recalculate certain values on-chip instead of for instance using look-up tables stored off-chip. As the size of shared memory is limited, considerations for using it should be considered early on in the design of the algorithm and its implementation so that one does not realize this optimization possibility too late in the process.

Second, in similar fashion to transfers between global and shared memory, transfers between the CPU and the GPU should also be kept to a minimum due to the low bandwidth between the two, as described in 7.3.6. Besides using the data transferred between the two as efficiently as possible and designing algorithms to maximize the use of transferred data, there is also an option called streaming which makes it possible to transfer data between the host and device while at the same time executing a kernel, as shown in Figure 21 below:

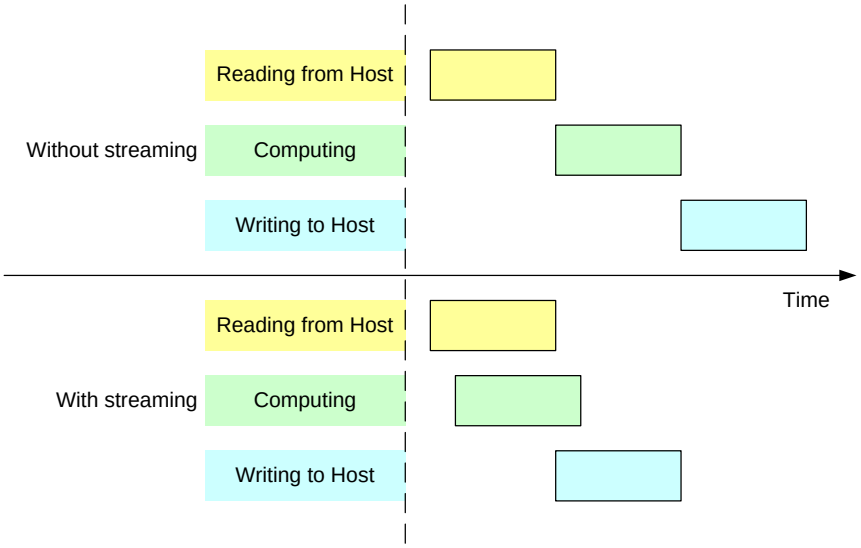

Figure 21 Streaming.

Streaming can for example be done if the kernel simply adds a constant value to elements of a matrix. As soon as a small piece of the matrix has been transferred to the device, a kernel can start computing, even though the whole matrix has not yet been transferred to the device. With the coming Fermi architecture, detailed in 13, streaming both ways simultaneously and running several kernels at the same time will be possible, further hiding the host-device transfers [19].

Streaming is thus also a way of hiding some of the latency in transferring data between the host and device, similar to hiding inter-GPU communication by writing having multiple blocks per SM so that some blocks can transfer data to and from global memory while other blocks compute.

Another way of hiding the latency between the GPU and the CPU is to have several kernels that follow each other in a chain-like fashion without needing to write back data to the CPU. This might for instance occur if several filters are to be added in series on a signal input. One of these kernels on its own might not justify using the GPU, but if several of them are computed one after the other, the total effect might be beneficial enough to motivate use of the GPU. An example is shown in Figure 22 below.

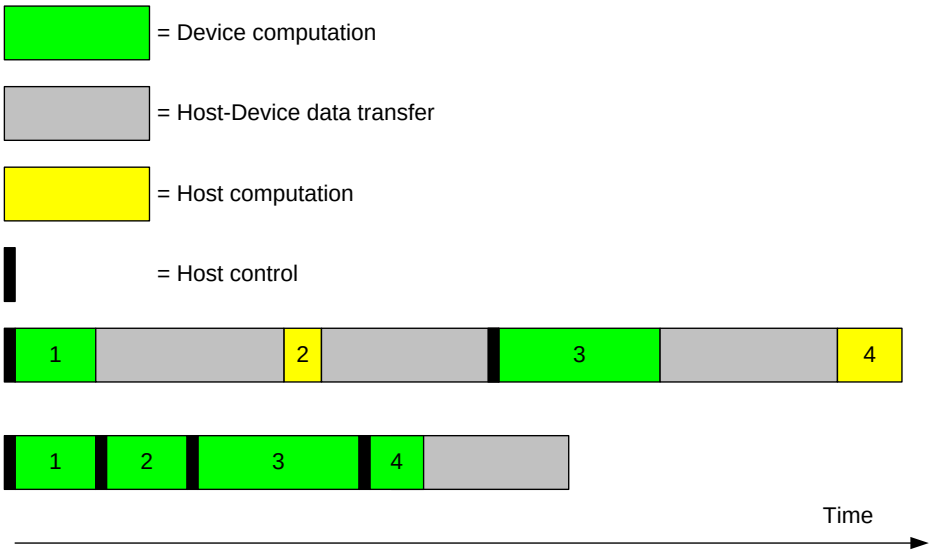

Figure 22 The benefit of keeping computations on the device.

By linking several kernels after one and other as in Figure 22 it is possible to cut down on the amount of time spent in expensive data transfers between the CPU and GPU. Note that even though computation *2* takes a longer time to compute on the device than on the host, it is still beneficiary to compute it on the device simply reduce the amount of expensive host-device data transfers.

10.2 Medium priority optimizations

Beside design and optimization guidelines of high priority, there are also several of medium importance, namely

- avoiding bank conflicts in shared memory as described in 7.3.2.1
- maintaining at least 18.75% occupancy in order to hide register latency dependencies as described in 7.3.3
- keeping thread blocks to a multiple of 32[11]
- using the “fast math” compiler flag when precision requirements can be relaxed.

The “fast math” flag ensures the compiler uses faster but less accurate version of functions such as sin, cos, inverse and similar.

[11] This not guaranteed to be the optimal size, as for example in [23] where 61 threads per block, not 64, was the optimal solution. But it does usually apply.

One aim when optimizing code is to keep the number of threads per block to a multiple of 32, as this is the size of a warp, the structural element constituting a block. If a block does not consist of a multiple of 32, say 40 threads, threads will be wasted. Having 40 threads mean that 2 * 32 - 40 = 16 threads will be wasted. If there are multiple concurrent blocks on an SM, a minimum of 64 threads per block should be used [20]. It is recommended to start testing blocks of either 128 or 256 threads if possible [21].

Finally, when optimizing instructions, one should reflect on the importance of accuracy over speed. The most significant choice to make is to choose between single or double precision. As described in 7.2, there are eight single precision FPU but only one double precision, and hence a program without any issues of low arithmetic intensity will run roughly eight times slower. Also, doubles take twice the register space compared to floats and also easily give rise to bank conflicts, as described in 7.3.2.1, potentially further degrading performance. Important to recall is that the float for "one half", for instance, is written 0.5f in C, not 0.5 which is the double precision for one half. If one enters a double into a function, CUDA will auto-convert any other value related to the computation to a double also [22].

If single precision is good enough, there are still more optimization alternatives, all at the cost of accuracy. There are, for example, several functions that in single precision mode have a fast equivalent with lower accuracy, such as exp, sin and sqrt.

Furthermore, there is the option of unrolling loops, just like on a regular CPU implementation. When using CUDA, unrolling a loop can be the difference between fitting an extra block on an SM or not, as unrolling a loop will free the use of the counting variable for the loop not just for one thread but for every active thread in the SM [23], in effect saving one register per active thread in the SM.

10.3 Low priority optimizations

Beside the high and medium priority guidelines, there are also some less important once detailed below.

First, there is less support for integer operations in CUDA than for floats. For instance, in one clock cycle, 8 single precision float multiplications can be computed per SM in one clock cycle where as only 2 32-bit integer multiplications can. If, however, one uses 24-bit integers, the throughput is also 8 computations per clock cycle. One theory for this behaviour is that colours are often stored as 24-bit integers, and as graphic cards are usually used for graphics, 32-bit integers are not shown the same amount of support.

Second, modulo operations and integer division are also very costly if they are performed with base two, i.e. with bit shifting [24].

As can be seen from the discussions above, there is a lot to keep in mind when designing algorithms and writing CUDA-code. But following the guidelines and performing performance tests should go a very long way in producing well performing code.

10.4 Advanced optimization strategies

The previously mentioned optimizations are those that Nvidia readily support and have on document. There are, however, further optimizations that can achieved through good hardware knowledge and understanding of the algorithm to be optimized.

10.4.1 High register usage and heavy threads: an alternative to Nvidia's programming philosophy

Nvidia's *CUDA Best practises guide 2.3* recommends the use of a "lightweight threading model" [25]. However, an alternative programming philosophy, most notably advocated by GPU programming guru Vasily Volkov, is to use heavy threads instead by use of the thread local registers [26]. The technicalities and difficulties of implementing such a heavy-thread programming philosophy are detailed below. Also, possible reasons for why advocates their light-weight thread recommendations are also given.

Often, the programmer can be limited by the shared memory being too small, leading to a bottleneck in many applications. There is however on-chip memory that often isn't used very effectively by the common CUDA programmer, namely the register memory. There are 16384 32-bit registers on each SM meaning that one can store 16384 floats on-chip. Compare that to the maximum 16384 byte / 4 byte/float = 4096 float that the shared memory can contain. If one manages to utilize the registers well, one can thus quadruple the amount of memory on-chip. This can in many applications, were for example many matrix operations are involved, lead to a significant performance increase.

For example, instead of storing a 64 by 64 matrix in shared memory, one may allocate 128 threads that each storing a 128 float long column (at most) of the matrix. If the operations being performed on the matrix are done column-wise, this method is not a problem since all the data is thread local. Allocating this much per thread block is, however, usually not the way to go. If for example each thread instead stores a 16 element long vector, one can in theory fit 8 thread blocks on each SM, thus increasing the occupancy and performance as we can fit more data on each SM. An illustration of how this thread local memory usage might look is shown in Figure 23 below.

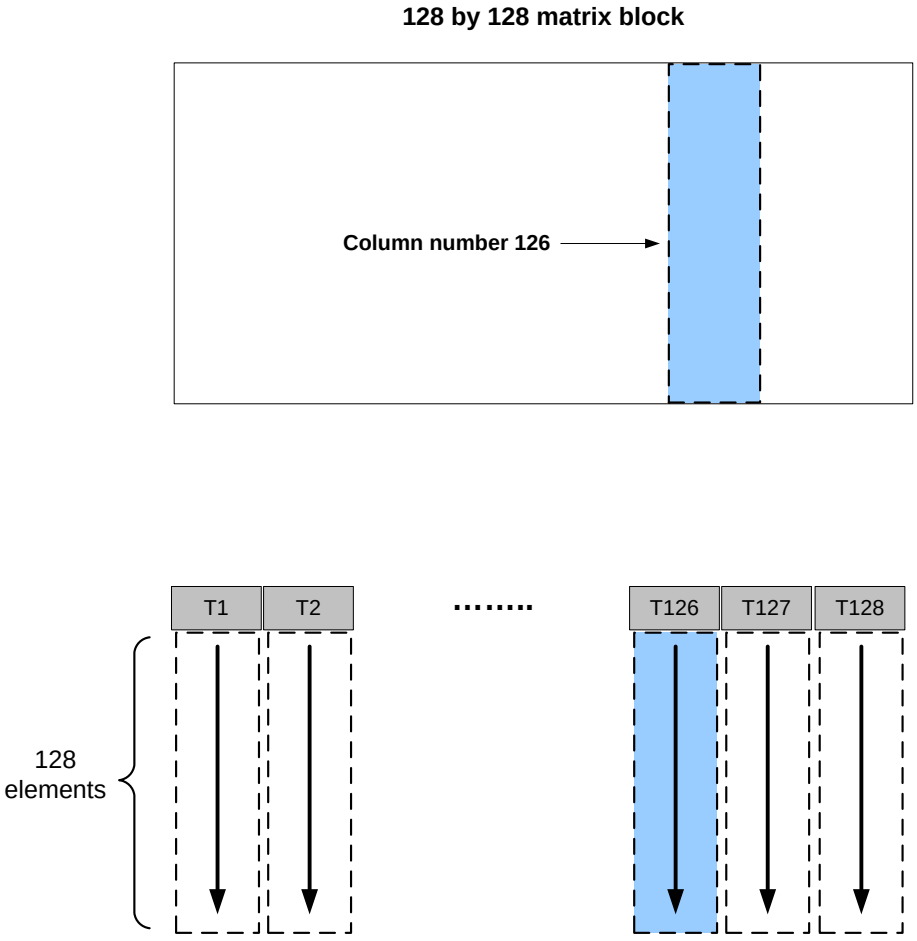

Figure 23 Example of how registers can be used to store a part of a matrix as thread local data.

The major gain being made here is that we are able to fit more data on-chip. Another positive side effect of this is that the registers don't suffer from any bank conflicts that might sometimes be unavoidable when using shared memory. In many applications, this increase in available on-chip memory may lead to significant performance increases of up to 2-3 times over an already well written CUDA kernel[12].

The downside however is that there is no simple way to allocate per thread arrays without having the registers spill over into slow local memory as accessing per thread arrays with anything but constant integers[13] automatically places the array in local memory which resides on the global off-chip memory.

[12] As can be seen in the results section for both STAP and Picture Correlation algorithms.
[13] threadIdx.x, blockIdx.y and similar are regarded as constant integers by the compiler as they are constant per thread, though will differ for different threads and blocks. This makes it possible to access arrays with use of threadIdx.x without the compiler placing the array in local memory, instead storing it in thread local register.

The solution to this is to resort to aggressive loop unrolling as well as some hard coding, making it possible to access thread local arrays though constant integers. To perform the unrolling, the loop depth must be known beforehand. If the problem is well defined, one may use templates or constant integers to make sure the compiler is able to do the unrolling without the registers spilling over into global memory. Once the loop depth is known, the studious programmer might typically choose to adapt his or her grid dimensions based on this parameter. While loop unrolling sounds trivial, the loops can in many applications involve many parameters which makes unrolling fiendishly difficult due to the demand for constant integers. This is a “No pain, no gain” optimization.

The difficulties involved in high register usage and the fact that it is easy to loose generality of the code are possibly the reasons why Nvidia does not recommend fewer heavy threads and instead opting for more light threads.

10.4.2 Global memory bank conflicts

Another possible performance hamper which is not described in the programming and best practices guides are global memory bank conflicts, known as partition camping. Partition camping is very similar to the shared memory bank conflicts described in 7.3.2.1, but applied to global memory. Global device memory is also divided into memory banks or partitions, but in much larger 256 byte partitions with a total of 6 partitions for the FX 4800. This means that the first 256 bytes and the next 1535-1792 bytes are in the same partition.

Partition camping can become an issue when multiple warps are reading or writing to the same partition, causing the operations to be serialized one after another. Hints of this issue often show up when the performance of the code varies for different dimensions of the problem being computed. Such an example is when doing the matrix transpose, when the output is being written the once row-wise elements are being written column-wise into the same global memory partition.

There are a few known solutions to this issue. One is to simply pad the data in order to avoid reads and writes being done simultaneously from the same partition. In the case of for example the matrix transpose, one can choose to offset the output matrix such that all the elements are written with an offset of 256 bytes. If one for example has a large matrix of floats, one might allocate an extra column of 64 floats which would be a sufficient offset to make sure that elements (i, j) and (i+1, j) will be placed in different partitions.

10.5 Optimization conclusions

Generally, the difficulty of writing reasonably optimized CUDA code that produces a speedup factor of say 10-50 compared to CPUs with similarly power and price depends a lot on the algorithm at hand. For instance, many radar signal processing algorithms are embarrassingly parallel and have simple, localized memory patterns and are hence reasonably easy to write optimized code for. On the other hand, for example genetic algorithms can be difficult to optimize due to their often non-localized or varying memory patterns; QRD is difficult due the many serial parts and on-chip memory demands; optimization algorithms are difficult due to the large amount of branching; and so on. But if one understands the hardware, adapts the algorithm to fit the hardware, and follows the optimization guidelines, writing reasonably optimized CUDA code is fairly simple if the algorithm suits CUDA.

An example of how the algorithm chosen might affect the final performance level can be seen in the QRD computation. When computing a QRD on the CPU, the “fast givens rotations” algorithm is often implemented. “Fast givens rotations” is a faster version of “givens rotations” due to it not calculating any expensive inverses, but with the cost of more instructions. An idea for a GPU version might thus be to use the regular “givens rotations” as the GPU handles inverse calculations very well compared to the CPU, meaning the GPU implementations would perform fewer instructions than the CPU. However, if one was to go down this route, one would sadly be off course.

Probably, the most efficient QRD for the GPU is to use so called “house holder transformations”. “Householder transformations” do not suit the CPU as well as “fast givens rotations”, but due to their parallelism and SIMD-like nature, they suit the GPU very well. Hence, even though one might have a very good understanding of “givens rotations”, one has already taken a wrong step if one is searching for an optimized QRD for the GPU. Often, the best CPU algorithm is not the best algorithm to implement on the GPU.

Furthermore, achieving a higher speedup of say 100+ is a lot more difficult than the initial factor of 10-50. Here, the advanced optimization details in 10.4 are often necessary. Also, one might have to resort to restructuring the input data in such a way as to eliminate bank conflicts further down the line of computations. Hence, even though the first order of magnitude might come fairly cheap, the second order of magnitude of speedup can be a lot more difficult to obtain.

11 OpenCL

"OpenCL is a framework for writing programs that execute across heterogeneous hardware platforms consisting of CPUs, GPUs"[14] and other processors, such as the Cell Broadband Engine or the coming Larrabee and Fusion processors. It also executes across software platforms. As OpenCL executes on across various hardware platforms, it is not solely aimed at GPUs in the way CUDA is. Also, OpenCL has less a set of strict specifications for handheld and embedded platforms. Despite this, CUDA and OpenCL have similar architecture specifications and memory model, programming model, and also a very similar C-based syntax.

The cross hardware platform capabilities of a general OpenCL program are intended to work through compiling kernels at runtime (or JIT, Just In Time) and, depending on the underlying hardware, adapt the code so that it executes optimally on the hardware, irrespective of if the code executes on a CPU, GPU, Cell-like processor or any combination of the three, similar to how OpenGL works for graphics. For instance, an OpenCL kernel run on a regular desktop PC should utilize both the GPU and the CPU simultaneously as the OpenCL drivers will compile the kernel to execute on all OpenCL available hardware. It is the same OpenCL code that runs on both the GPU and the CPU, but adapted at runtime to suit the hardware.

However, programs can also be compiled offline, i.e. already be compiled prior to system start-up. "The program binary can consist of either or both:

- Device-specific executable(s), and/or,
- Implementation-specific intermediate representation (IR) which will be converted to the device-specific executable." [27]

In other words, it is possible to compile programs for a device or devices before hand, thus removing the need to include and OpenCL compiler in the system and minimizing start-up time of the OpenCL implementation.

Further similarities between the CUDA and OpenCL are that they both are based on the idea of a host in control, and one or several devices that do the heavy computing. In both CUDA and OpenCL, the devices execute a kernel initiated from the host. Also, the hardware specifications and memory hierarchy of OpenCL seem to fall in line with present CUDA offerings, as can be seen in Table 8 below:

[14] http://en.wikipedia.org/wiki/Opencl

	CUDA 1.3 hardware	OpenCL 1.0
Shared memory	16 KB	16 KB (minimum)
Constant memory	64 KB	64 KB (minimum)
Dimensions of block	3	3 (minimum)

Table 8 CUDA and OpenCL resemble each other in many aspects, as is shown in the table above.

Both use CUDA and OpenCL use the model of global, shared, register and constant memory in the same manner. Also, both CUDA and OpenCL use a grid like structure, which consists of data-parallel blocks that in turn consists of threads, though the exact terms differ. The lack of recursion is yet another similarity between the two.

The future of both CUDA and OpenCL and their relation is unclear. Though OpenCL has a broad support on paper, notably Intel and Microsoft, among many others, are lacking in any active support. Developer environments and similar support is also lacking as described in 12.2, and investigations into the performance of how identical OpenCL kernels run on different hardware platforms do as of yet not exists, hence the success of the aim of having created a cross hardware platform framework at all can so far be put into question.

In essence, the difference between the two is that CUDA has more flexibility, maturity and developer support than OpenCL, and also several well developed libraries, numerous white-papers, and also adds hardware capabilities at a faster pace than OpenCL. On the other hand, OpenCL has, in theory, a larger hardware base and a more open attitude to revealing future plans and developments. However, it stands to reason to believe that Nvidia will be able to push forward new hardware capabilities at a faster pace than the Khronos Group, seeing as Nvidia does not have to come to a multi-corporate cross hardware functionality agreement for every part of the CUDA specifications which is a must for OpenCL.

The future of both CUDA and OpenCL are somewhat unclear. What CUDA lacks in openness, OpenCL lacks in maturity, recognition and adoption. The lack of a good developer support as well as adoption might well place OpenCL in a “chicken or egg”-situation where the lack of developer support leads to too few developers taking OpenCL seriously, in turn further leading to a stall in developer support. However, with AMD, Apple and possibly also IBM pushing for further adoption and support of OpenCL, the maturity and developer support of OpenCL might well increase rapidly in the near future.

On the other hand, CUDA, unlike OpenCL, has reasonably large developer base and Nvidia has repeatedly pointed out its commitment to GPGPU computing, mainly through the use of CUDA, not OpenCL. It thus seems reasonable to believe that the support and adoption of CUDA is set to increase further, at least in the near future, unless a broad and rapid increase in support and adaption of OpenCL occurs, or the coming CUDA architecture Fermi for some reason fails to be adopted.

The programming challenges and the developer environment of CUDA and OpenCL, or the lack there of, are described in 12.1 and 12.2 below respectively.

12 Experience of programming with CUDA and OpenCL

12.1 CUDA

Developing for CUDA is reasonably straight forward. There are good developer environments for Windows, Linux and Mac systems, a fair amount of relevant documentation, and also active developer forums. There are also basic libraries for FFTs, basic matrix operations and similar.

Developing in either Windows Vista or Windows 7 has the extra benefit of a developer tool called Nexus, which allows for GPU memory inspection, placing breakpoints inside GPU code and similar. These functions do as of yet not exist for either Linux or Mac systems. However, users of all systems are able to run in emulated GPU mode to investigate code. This is an essential development tool as the developer otherwise must resort to print statements. The latter is however sometimes necessary as the GPU and the GPU emulator do not always produce the same result, due to various reasons. This has recently been simplified as it is now possible to print directly from the GPU instead of having to write code to copy values to the CPU and then print from there.

Furthermore, a CUDA FORTRAN compiler has been developed by The Portland Group, and hence porting and developing FORTRAN code to use the benefits of CUDA GPUs is now possible without resorting to C.

Developing for CUDA basically consists of two parts: Formulating a parallelized solution in relation to the hardware constraints, and then the necessary coding.

The first part necessitates a good understanding of the CUDA hardware and the algorithm at hand, and also the ability to divide the problem into data parallel groups. The programmer hence has to deal with many of the parallel programming hazards such as synchronization issues as with a regular parallel C code, though the authors believe this to be simpler in CUDA than with for instance OpenMP or MPI, partly due to CUDA's relatively simple parallel nature; partly due to the necessary connection to the parallel CUDA hardware.

The second, coding part of the process is not much more of an issue than writing ordinary C code. Certain CUDA related API calls and similar quickly become as natural as for regular CPU C code. Also, as stated earlier, emulated debugging is very helpful in presenting a quick and easy simulation of the GPU.

Considering CUDA development as a whole, it is as straight forward and uncomplicated as writing efficient parallel code can be, though it does necessitate the writing of C code, taking the hardware into account, and writing of parallel code. CUDA is no free lunch, nor is it expensive.

Furthermore, the coming CUDA architecture called Fermi will increase the simplicity and flexibility of code further by implementing a unified address space for thread local, block local and global memory with regard to load and store operations. This in turn leads to increased C++ capabilities as function pointers and virtual functions will be supported.

Estimating the time it would take for an experienced C-programmer to write reasonably well optimized CUDA programs is for many reasons difficult to do, especially considering that C is one of the most commonly used languages with very varied kinds of implementations, hence many interpretations of “experienced C-programmer” are possible. It is however possible to construct a profile for what a C-programmer should have experience in to have a good start to programming CUDA.

First, the C-programmer must be comfortable with considering hardware constantly throughout the development process as it is the hardware that decides the algorithm to use. The GPU architecture is not as forgiving as a CPU architecture. Hence, having developed say a text editor might not be of much help. Second, the programmer has to have at least basic knowledge of developing algorithms and be able to do so within the CUDA’s hardware constraints as simply taking your average CPU algorithm and pasting it into CUDA will not give a good performance at all. Lastly, basic understanding of parallel and/or high performance programming is recommended. The latter of the two might apply more due to parallel programming generally being viewed as reasonably difficult, though CUDA seems to be viewed as an easy way to enter the world of parallel programming.

If a C-programmer fits the profile reasonably well, he or she should be able to set up the environment, read the documentation and be up and running with basic CUDA programs within maybe 4 weeks.

It should be noted that during this investigation, several other sources have been found with results that we would interpret as less than satisfactory, yet the authors are often impressed with a factor of 2-5 times speedup, and often seemingly disregard the most basic optimization guidelines.

12.2 OpenCL

Developing for OpenCL is as of yet not a simple task. The lack of any development environment with the possibility of emulating and hence debugging code on any underlying hardware means the programmer is basically stuck with a text editor and print statements. Print statements are not only often insufficient, but unlike CUDA, OpenCL does not allow for print outs directly from the device; instead the data must first be transferred to the host, a time consuming and exceedingly boring task. Another issue that easily arises without a debugger is system crashes. In CUDA, out of bounds memory access are easily detected when emulating the GPU. As no such possibility exists in OpenCL, the application must be run to see if it performs as expected. If one has happened to write out of bounds, it is not uncommon for the system to crash. This does not help developer productivity.

Beside the lack of a good developer environment, there is as of yet only one library intended for computing, namely Apple's recently released FFT library. The FFT library has not been implemented in this investigation, nor has any good investigation of it been found as of yet.

Though Nexus does as of yet not have support for OpenCL, Nvidia claims to be working for OpenCL integration in a future version. If and when Nexus receives OpenCL support, it should be of great benefit to developers as one should no longer have to code in the dark.

Furthermore, no FORTRAN compiler or similar is known to the authors to be in development.

Developing for OpenCL is similar to CUDA in regard to parallelising and the problem and writing C code, with a few notable of exceptions. First, general optimization guidelines for various hardware such as GPUs, CPUs, Cell-like processors and similar do not exists, or as written in the OpenCL specifications, "It is anticipated that over the coming months and years[,] experience will produce a set of best practices that will help foster a uniformly favourable experience on a diversity of computing devices." Hence, if one is writing code that is to be optimized not just for a GPU architecture, but several GPU architectures, CPU architectures, Cell-like processors, DSPs and more, one is basically in the dark, possibly performing serious research and breaking new ground. Nvidia has however released a "Best practices guide" for writing OpenCL code for Nvidia GPUs. The general ideas likely also fall in line with similar recommendations for AMD GPUs.

The second, coding part of the process is not much more of an issue than writing ordinary C code, except for the lack of way to debug ones code without the use of print statements. The OpenCL related API calls and similar are less obvious than the CUDA counterparts, though the terminology is in many cases more suitable.

Considering OpenCL development as a whole, it is clearly less efficient than developing for CUDA. Also, considering the lack of clear examples where OpenCL code runs well on let alone different GPUs, not counting other hardware, puts the goal of a heterogeneous cross platform and hardware framework into question. Hopefully though, the commitment of AMD, Apple and others will sort out the seemingly immature developer situation, and as more people in industry and academia test OpenCL for various kinds of problems and hardware, guidelines for writing efficient cross hardware platform kernels will hopefully develop.

For this investigation, OpenCL development was performed through first writing the CUDA kernels, and then more or less implementing “copy and paste”, and then literary translating from CUDA C syntax to OpenCL C syntax. This process, though, is not as simple as it sounds.

Due to the reasons stated above, it is as of yet difficult to recommend OpenCL for more than research, unless one is interested in writing cross GPU-vendor PC applications, such as “the creation of multimedia and creativity software products” [28] as Adobe Systems intends to do. Notably though “Adobe [will be] going to CUDA first. The plan is probably equal to all plans that we [have] heard [of] so far: go to CUDA in order to completely unlock the GPU potential and only then port to OpenCL, as Apple's and AMD's OpenCL toolkits mature, sometime in 2011.” [29]

12.3 Portability between CUDA and OpenCL

As shown in benchmarks 14.3 and 14.10, basic CUDA and OpenCL kernels seem to perform similarly on the same hardware. As the OpenCL code was basically just a port from the CUDA code, portability between CUDA and OpenCL is hence a possibility to some extent. However, the performance of the kernel implemented in this investigation might not work as well on AMD GPUs, let alone Cell-like processors and similar. OpenCL cross hardware platform performance is as of yet unknown. However, as stated in 12.2, if one intends to produce OpenCL code to run on the GPU, it is recommended to first write a fully functioning and optimized CUDA implementation, and then more or less copy the solution to OpenCL.

The conversion from CUDA to OpenCL took roughly the same amount of time as writing the actual CUDA code itself. This is mainly due to the very rough edges in the OpenCL development, and also the large difference of code on the host side between CUDA and OpenCL. When possible errors occur, they can often take a long time to find due to the lack of a debugger.

13 Fermi and the future of CUDA

Nvidia has as of recently [30] unveiled its latest, though not yet released architecture, called Fermi. Fermi makes way with many of the limitations of the present CUDA hardware, for instance allowing up to 16 kernels to run simultaneously instead of only one, greatly improving support for double precision, ECC support, a true cache hierarchy, more shared memory, faster atomic operations, C++ support, and also allows for recursion. Also, besides adding increasing double precision and ECC support which will certainly be appreciated by the HPC market, Nvidia have been very vocal and active in their aim for Fermi to take a significant place in the HPC market and in servers for heavy number crunching.

Moreover, both Intel and AMD are slowly moving small integrated graphics chips onto the CPU chipset. How this might affect future CUDA implementations is hard to say, seeing as Nvidia lacks any CPU products, AMD producing their own graphic cards and CPUs, and Intel trying to produce their own discrete graphic cards known as Larrabee, hence likely limiting both AMD and Intel interests in pushing for integrated Nvidia GPU support.

14 Overview of benchmarks

14.1 High Performance Embedded Computing: HPEC

The HPEC benchmark suite[15] consists of 10 benchmarks, some of which are tested here. They are used as a benchmark to test different sets of hardware and algorithms against each other. However, the benchmarks also come with data sets and sizes that are to be tested for. These data sets are only tested for CFAR and SAR. In the other benchmarks implemented below, a substantially larger number of data sets are tested for, often spanning several orders of magnitude in workload. The reason for this is both that the presented data sets are in many cases too small if not several data sets are run simultaneously to accurately evaluate the GPU, and also it is very difficult to draw any conclusion and see larger patterns if only one or two data sets are benchmarked. Hence, the HPEC algorithms are implemented for many different data sets and sizes. The HPEC suit benchmarks included in this investigation are: TDFIR, FDFIR, QRD, SVD, CFAR, and CT.

[15] For further details regarding the HPEC benchmark, see http://www.ll.mit.edu/HPECchallenge/

14.2 Other radar related benchmarks

For radar signal processing applications, there are also several other benchmarks that are of interest[16]. These include: Bi-cubic interpolation, Neville's algorithm, SAR, STAP and FFT. These are, like the HPEC benchmarks, tested for a wide range of data sets except the STAP benchmark which has 2 fixed data sets.

14.3 Time-Domain Finite Impulse Response: TDFIR

Time-Domain Finite Impulse Response, TDFIR, is an algorithm used in for instance pulse compression in radar signal processing. It consists of a large number of scalar products, on for each element in each input vector, as is shown in Figure 24.

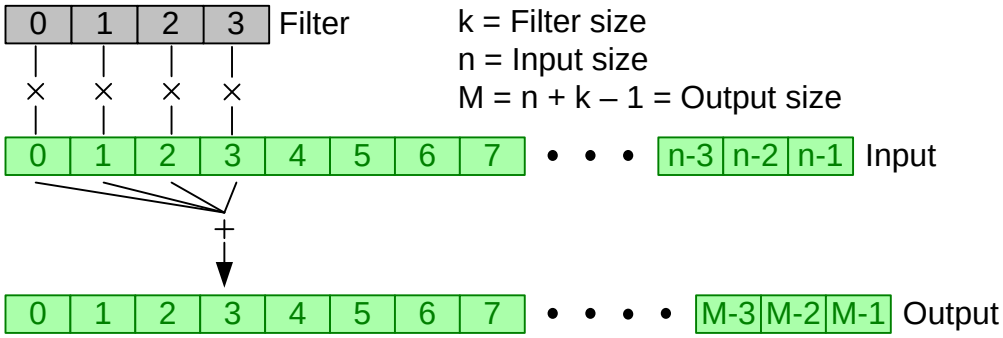

Figure 24 An illustration of the TDFIR algorithm.

The relevant parameters are the length and number of input vectors, and the length and number of filters. The number of output vectors is equal to the number of filters. All data is complex.

14.4 Frequency-Domain Finite Impulse Response: FDFIR

Frequency-Domain Finite Impulse Response, FDFIR, is an algorithm used in long filters. It is similar to TDFIR, but includes an FFT transform of both the input and the output vector. All data is complex.

[16] For further details regarding these benchmarks, see SMW report 4/0363 – FCP 104 1180 Uen

14.5 QR-decomposition

The QRD, or QR-decomposition, is used to solve linear systems of equations. In radar applications such as STAP, the QRD is performed on several rectangular matrices, roughly of size [200, 100]. This should be done as quickly as possible for a large number of identically sized matrices. The matrix is complex.

14.6 Singular Value Decomposition: SVD

The SVD, or Singular Value Decomposition, has many applications in signal processing. In radar applications, the SVD is performed on several rectangular matrices, roughly of the size [200, 100]. This should be done as quickly as possible for a large number of identically sized matrices. The matrix is complex.

14.7 Constant False-Alarm Rate: CFAR

Constant False-Alarm Rate, or CFAR, is an algorithm that finds a target in an environment with varying background noise. It consists of comparing the value in one cell, cell C, to the sum of several nearby cells, T, as shown below in Figure 25 below.

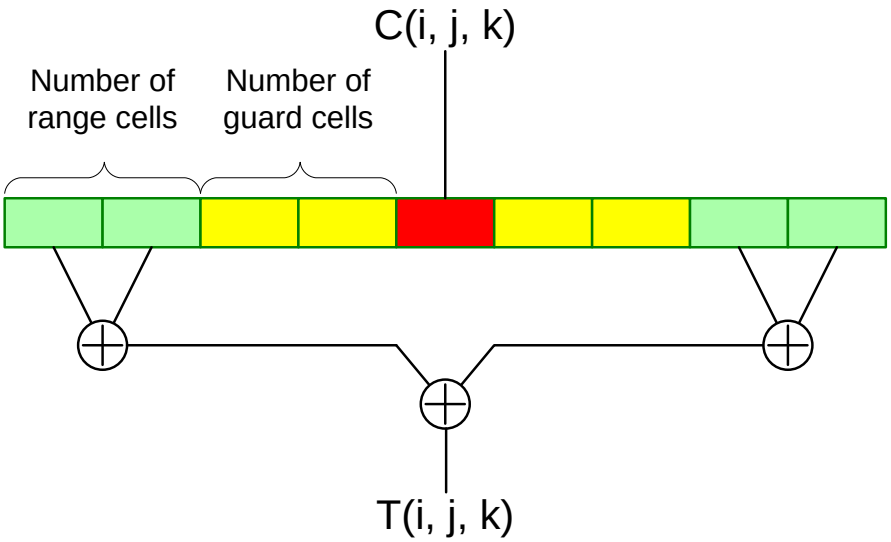

Figure 25 An illustration of the CFAR algorithm.

14.8 Corner Turn: CT

Corner Turn, or CT, is used in radar signal processing to reorder data in memory to allow faster access for other signal processing algorithms. It consists of transposing a matrix in memory; in other words switching between row-major and column-major, as is shown in Figure 26.

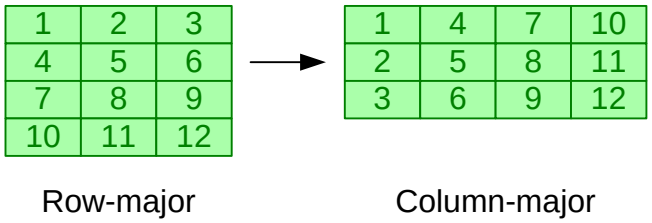

Figure 26 An illustration of the CT algorithm.

14.9 INT-Bi-C: Bi-cubic interpolation

Bi-cubic interpolation is a common interpolation algorithm not only used in signal processing. It consists of performing 4 cubic interpolations of four sets of four identically spaced values, and then performing a fifth cubic interpolation of the previous 4 interpolations.

Note that Bi-cubic interpolation produces a smooth surface with a constant derivative.

14.10 INT-C: Cubic interpolation through Neville's algorithm

Neville's interpolation is a less common interpolation algorithm. It consists of performing 6 linear interpolations from 4 unevenly spaced values. A simplified illustration is shown in Figure 27 below. Note that the 4 values at the top are not evenly spaced. They are interpolated in three steps until only the final interpolated result remains, the value at the bottom.

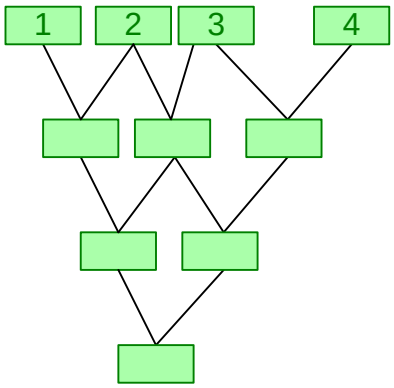

Figure 27 An illustration of Neville's interpolation.

Note that this is a linear approximation only. No smooth derivative is achieved as with Bi-Cubic interpolation.

14.11 Synthetic Aperture Radar (SAR) inspired tilted matrix additions

The SAR image produces a high-resolution image from several measurements taken while flying over the target area. One essential part of this algorithm is performing tilted matrix additions, the reason being that tilted matrix additions have complicated memory access patterns. They are performed as show in Figure 28, in which the two matrices *A* and *B* are summed to produce *C*.

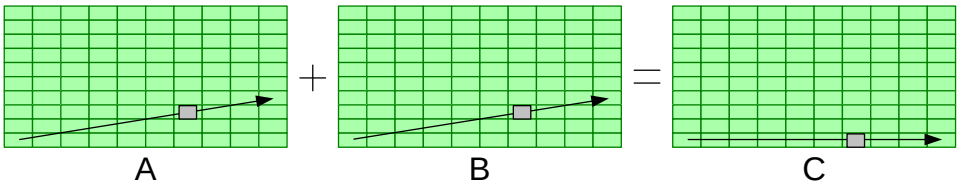

Figure 28 An illustration of SAR inspired tilted matrix addition.

For the benchmark, values are wrapped around in the sense that when a tilted line reaches the top of the matrix, it continues from the bottom.

14.12 Space-Time Adaptive Processing: STAP

Space-Time Adaptive Processing is a state of the art algorithm in real-time radar signal processing implementations. It continuously adapts by weighting incoming signals depending on noise and other conditions. Two versions of the compute intensive parts where implemented, the first using covariance matrix estimation, the second using QRDs. Both of the benchmarks did this for a number of matrices in parallel.

14.12.1 Using covariance matrix estimation

A less complicated version of STAP is using covariance matrix estimation. It consists of estimating several covariance matrices that are computed in parallel.

14.12.2 Using QRD

A more effective and also more complicated method to achieve high numerical accuracy and stability is to instead do QRDs over all Doppler channels. This is more compute intensive than when doing a covariance matrix estimation and a bit less trivial to parallelize. It consists of performing hundreds of QRDs in parallel.

14.13 FFT

The Fast Fourier Transforms, FFT, is a commonly used algorithm in signal processing, and it is benchmarked for a wide range of lengths and numbers of input vectors.

14.14 Picture Correlation

Picture correlation compares images with different shifts to find which configuration has the best correlation as shown in Figure 29 below**.**

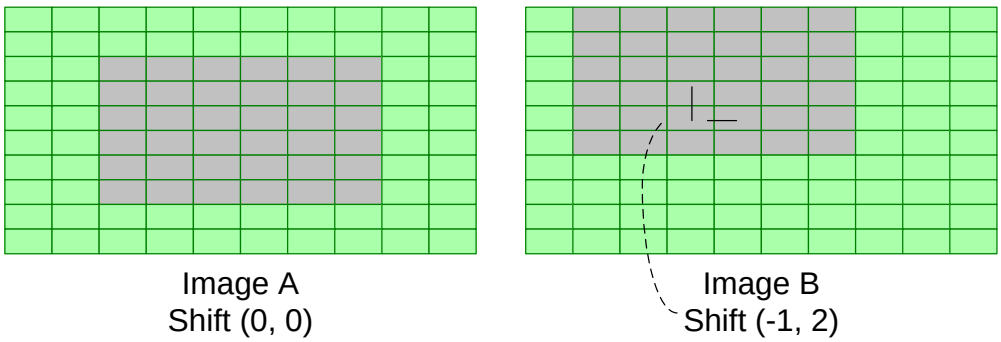

Figure 29 Picture correlation, compares images with different shifts.

14.15 Hardware and software used for the benchmarks

The GPU and CPU hardware used in the tests are shown in Table 7 and Table 8 below respectively.

Graphics board	Nvidia Quadro FX 4800
GPU	GTX 260
Memory	1.5 GB GDDR3
Memory Bandwidth	76.8 GB/s (16 PCI lanes)
Number of SMs	24
Number of SPs	192
Clock frequency	1.204 GHz
On-chip memory per SM	16 KB
Registers per SM	16K 32-bit
Instructions	FMAD

Table 9 The GPU hardware used in all tests.

CPU	Intel Core2Duo E8400
Memory	4GB (2 x 2 GB DIMM) 800Mhz DDR2
Front Side Bus	1333 MT/s
Number of cores	1 (the second core is turned off in all tests)
Clock frequency	3.00 GHz
Cache memories	6 MB L2, 32 KB L1 instruction, 32 KB L1 data
Instructions	X86 SSE4.1

Table 10 The CPU hardware used in all tests.

For all GPU benchmarks, Windows XP SP3 32-bit was used. For all the CPU benchmarks except for FFT, Linux Suse 9.11 32-bit was used. For the CPU FFT benchmark, Windows XP SP3 32-bit was used.

All GPU benchmarks were compiled for compute capability 1.3, using drivers 191.66.

All CPU benchmarks were run **with only one core** running to make interpretation of the results easier as the compilers' and OS's degree of optimizing the code for two cores then becomes irrelevant. All Linux CPU benchmarks were compiled with GCC using the flags 'xc', 'ansi', 'lm' and 'O3', allowing the compiler to add SSE instructions where it deemed possible. The Windows benchmarks where compiled with 'O3'. For the QRD and SVD benchmarks, Intel's MKL (Math Kernel Library) was used, run in windows, beside the HPEC implementations.

Also, all tests are done for **single precision only**. **GFLOPS** and **GB/s** are computed with **base 10** throughout the benchmarks, **not 1024**.

14.16 Benchmark timing definitions

These parameters have been used when measuring performance of the different benchmarks.

- **Kernel time** refers to the time of running the GPU kernel only, **not** including any transfer between host and device.
- **Total GPU time** refers to the time of running the GPU kernel only **and** any transfer between host and device.
- **Kernel Speedup** is how much faster the GPU executes a kernel compared to the CPU.
- **Total GPU speedup** is how much faster the GPU executes a kernel compared to the CPU **including** any transfers between host and device.
- **Peak performance** is either the maximum GFLOPS or maximum throughput achieved for what can be considered radar-relevant data sizes. GFLOPS is a metric for how many billion floating point operations per second are being performed. The way performance or throughput is calculated is detailed in each respective benchmark. As the aim with performance is to see how hard the GPU is working, the transfer times between host and device are not included; only kernel time is relevant.

15 Benchmark results

15.1 TDFIR

Maximum kernel speedup over CPU: 100 times
Maximum GPU performance achieved: 182 GFLOPS
Maximum CPU performance achieved: 2.6 GFLOPS

Results:
The highest kernel speedup of the GPU implementation was about 100 times faster than the benchmark code run on the CPU. The highest speedup including all memory transfers was about 37 times.

Implementation:
The TDFIR algorithm is detailed in 14.3. The GPU version of the algorithm was implemented by exploiting the embarrassingly parallel nature of the problem, splitting different parts of the input into data parallel blocks.

Figure 30 and Figure 31 below show the performance and speedup respectively for three different filter lengths, using a single filter for varying input vector lengths.

Prepared (also subject responsible if other)		No.		
SMW/DD/GX Jimmy Pettersson, Ian Wainwright		5/0363-FCP1041180 en		
Approved	Checked	Date	Rev	Reference
SMW/DD/GCX Lars Henricson	DD/LX	2010-01-27	A	

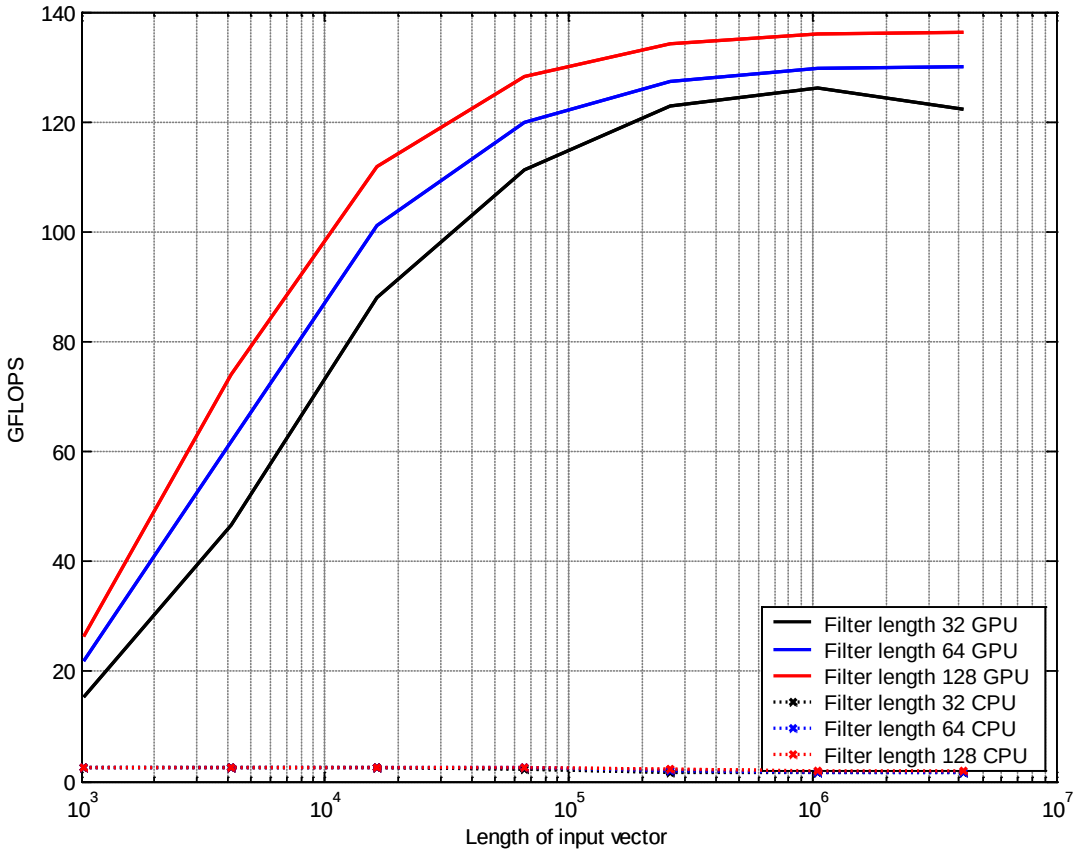

Figure 30 Performance for TDFIR for a single filter for varying input lengths.

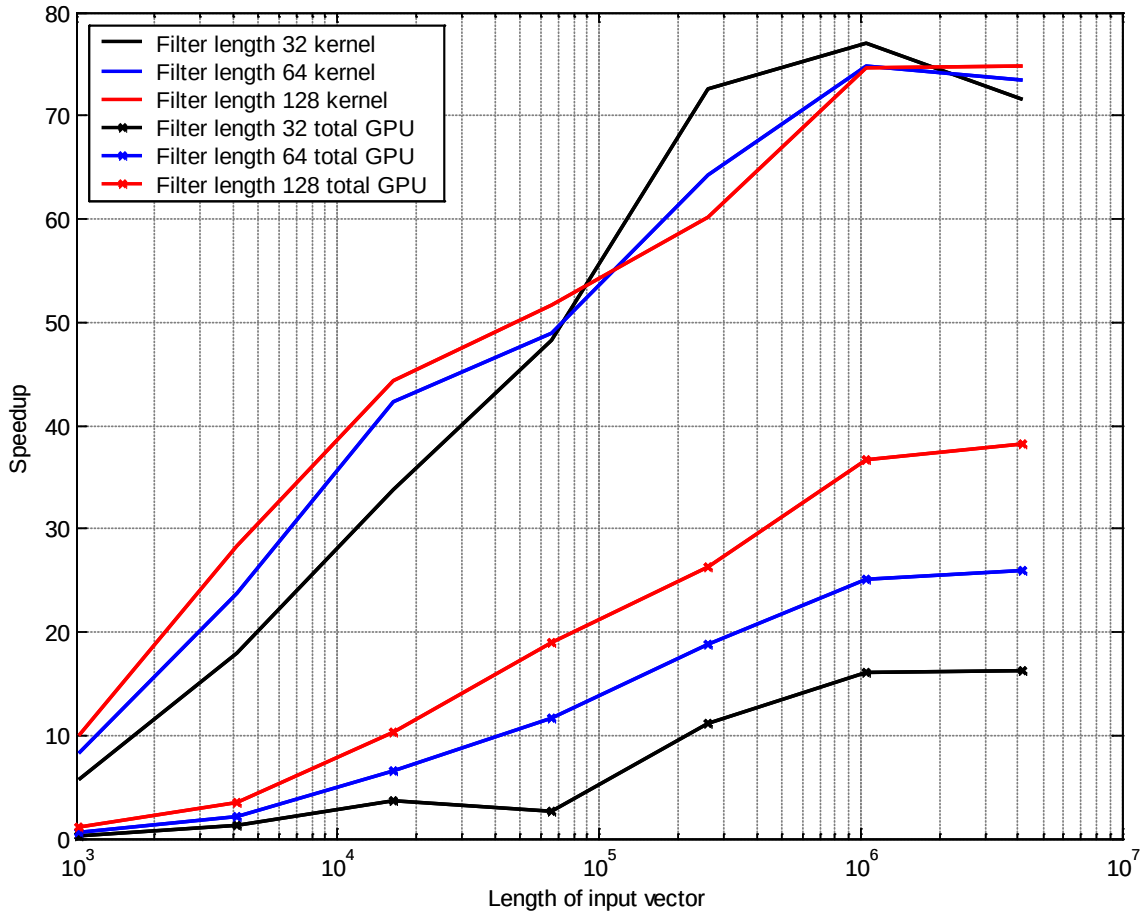

Figure 31 Speedup for TDFIR for both the kernel and the total GPU time using a single filter.

The GPU clearly benefits from a larger problem size as larger input vectors give rise to a higher performance. Seeing as the filter length is linearly related to the AI, increased filter length produces a higher performance. The CPU implementation is largely input vector size-independent, though a slight decrease in performance is measured for increased input vector size, and is completely filter length independent. Hence, as the GPU unlike the CPU benefits from both increased data size and increased AI, a higher speedup is seen for larger and heavier data sets.

Figure 32 and Figure 33 below show the performance and speedup respectively for three different filter lengths, using a varying number of filters for a single input vector consisting of 4096 elements.

Prepared (also subject responsible if other)		No.		
SMW/DD/GX Jimmy Pettersson, Ian Wainwright		5/0363-FCP1041180 en		
Approved	Checked	Date	Rev	Reference
SMW/DD/GCX Lars Henricson	DD/LX	2010-01-27	A	

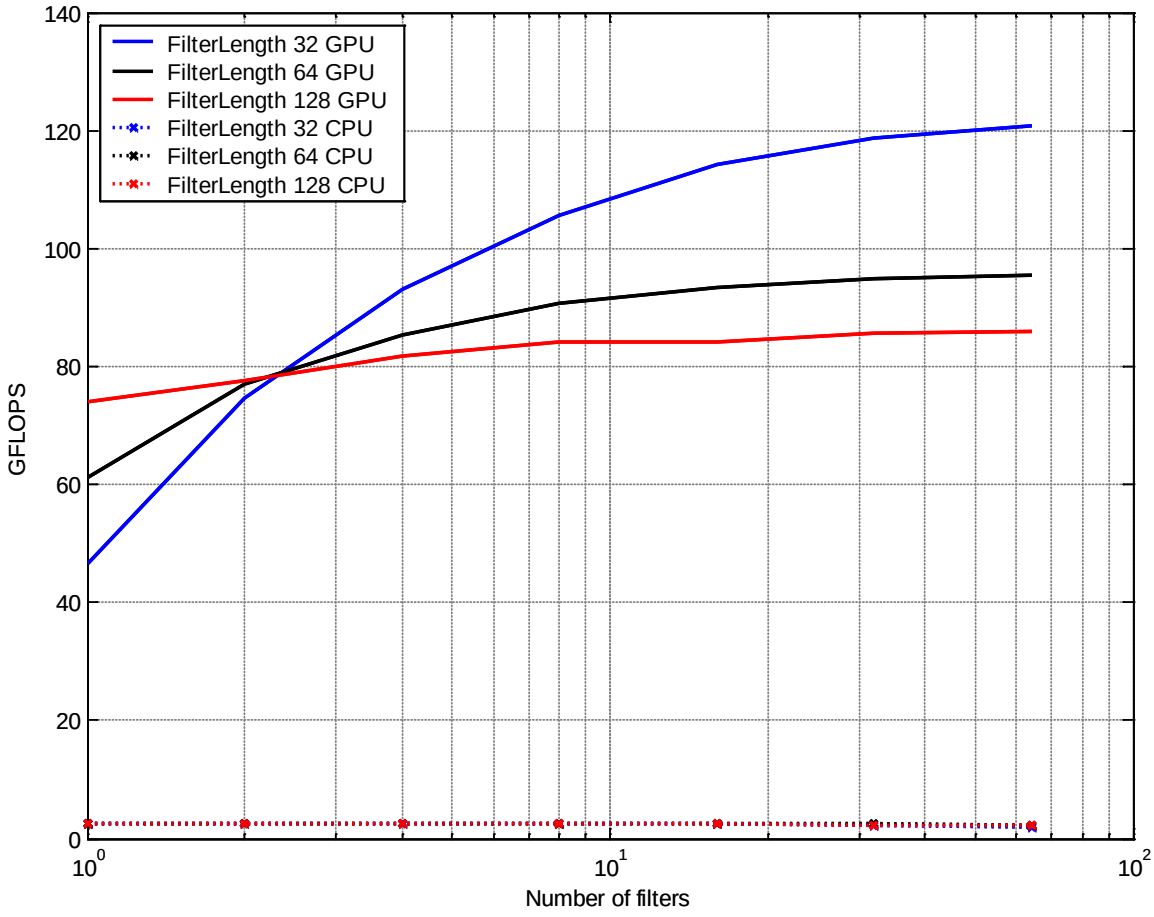

Figure 32 Performance for TDFIR for a 4096 elements long vector for a varying number of filters.

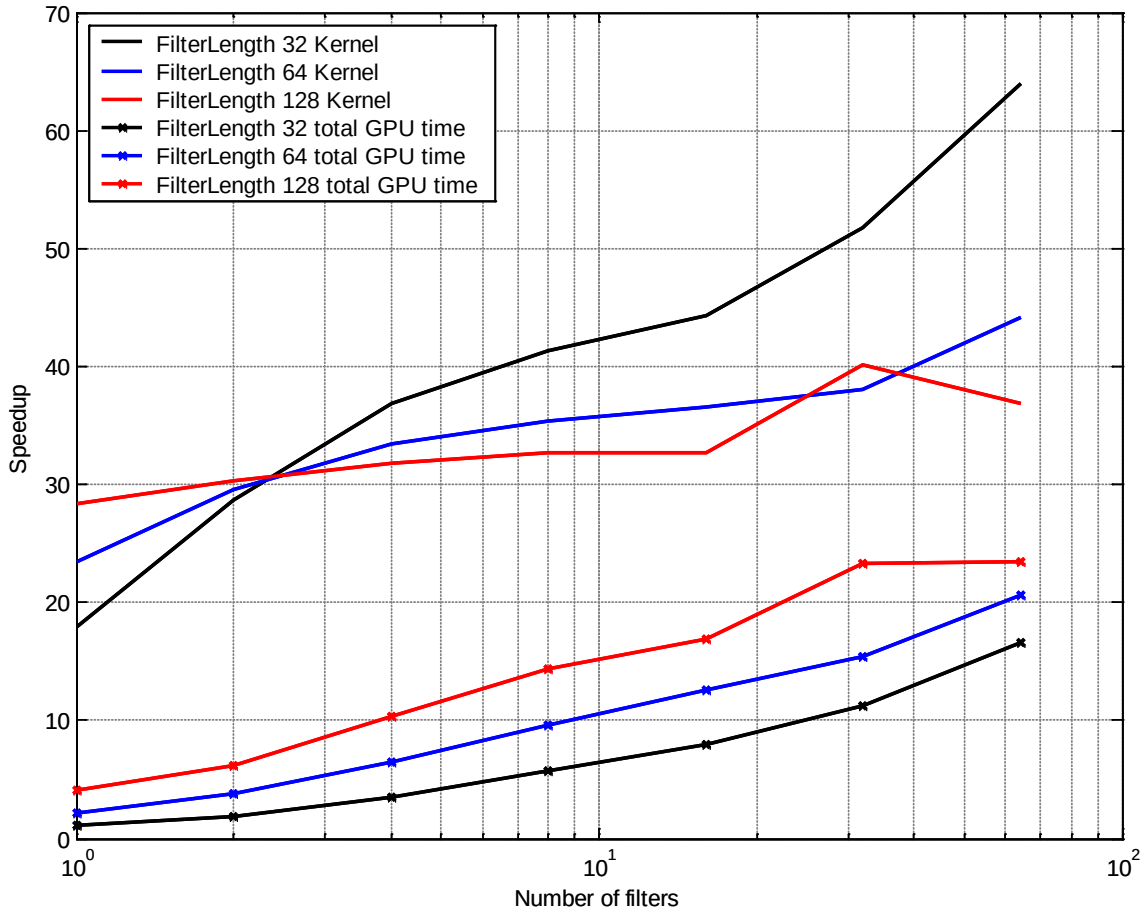

Figure 33 Speedup for TDFIR for both the kernel and the total GPU for a 4096 elements long vector for a varying number of filters.

Again, The GPU clearly benefits from problem sizes. However, increased filter length is not beneficiary for the whole range of number of filters. This is due to the limited size of the constant cache on each SM in which the filter resides. As the constant cache is only 8 KB, where as the size of constant memory on in global memory is 64 KB, it is not possible to fit all filter data on-chip for certain numbers of filters. This is most apparent when comparing the 128 element long filter to the 32 element long filter for 1 filter and 64 filters. The total AI is always higher for the 128 element long filter and hence it has a higher performance for few filters. However, as the number of filters grows, the 128 element long filters do not all fit on-chip. This leads to an increase in global memory accesses compared to the shorter 32 element filter, which gives rise to the cross in the graph where the shorter filter overtakes the longer filter in terms of performance. A more detailed explanation of this the difference between local and global AI is given in 9.4.

Finally, some very heavy data sets were also tested for, using multiple filters of varying lengths. The performance and speedup results are shown in respectively.

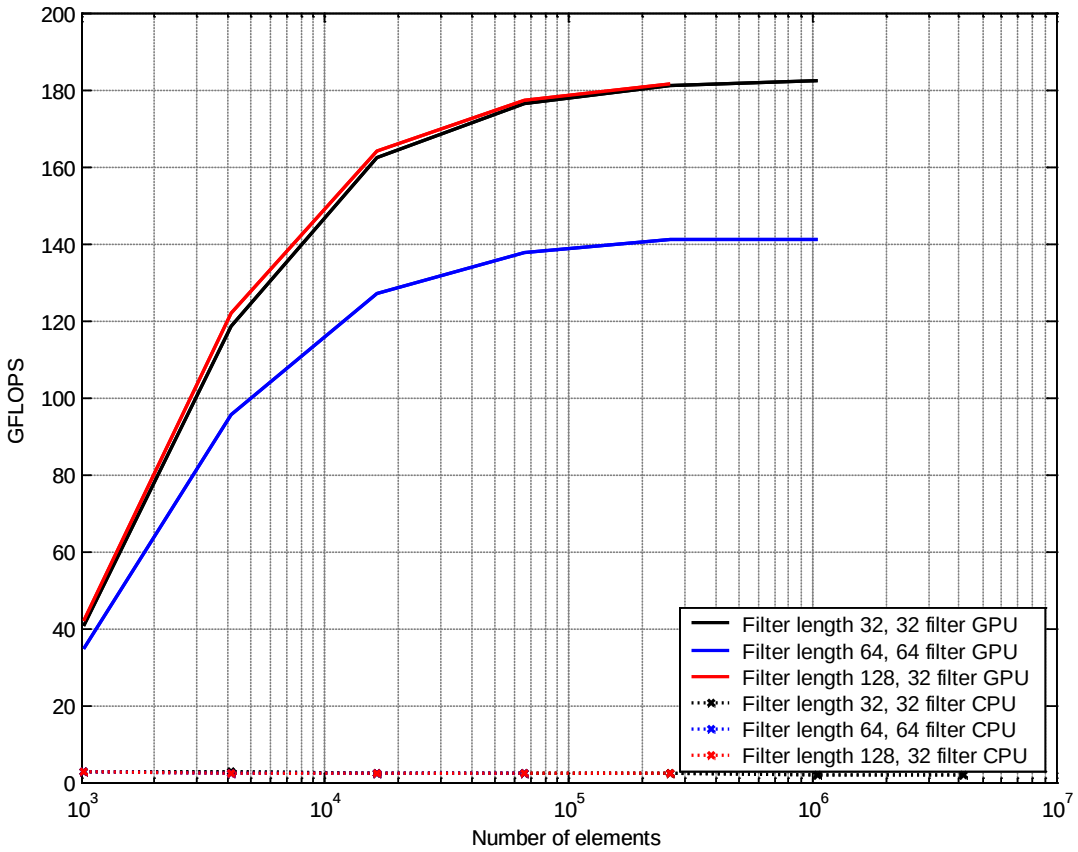

Figure 34 Performance for TDFIR for heavy data sets.

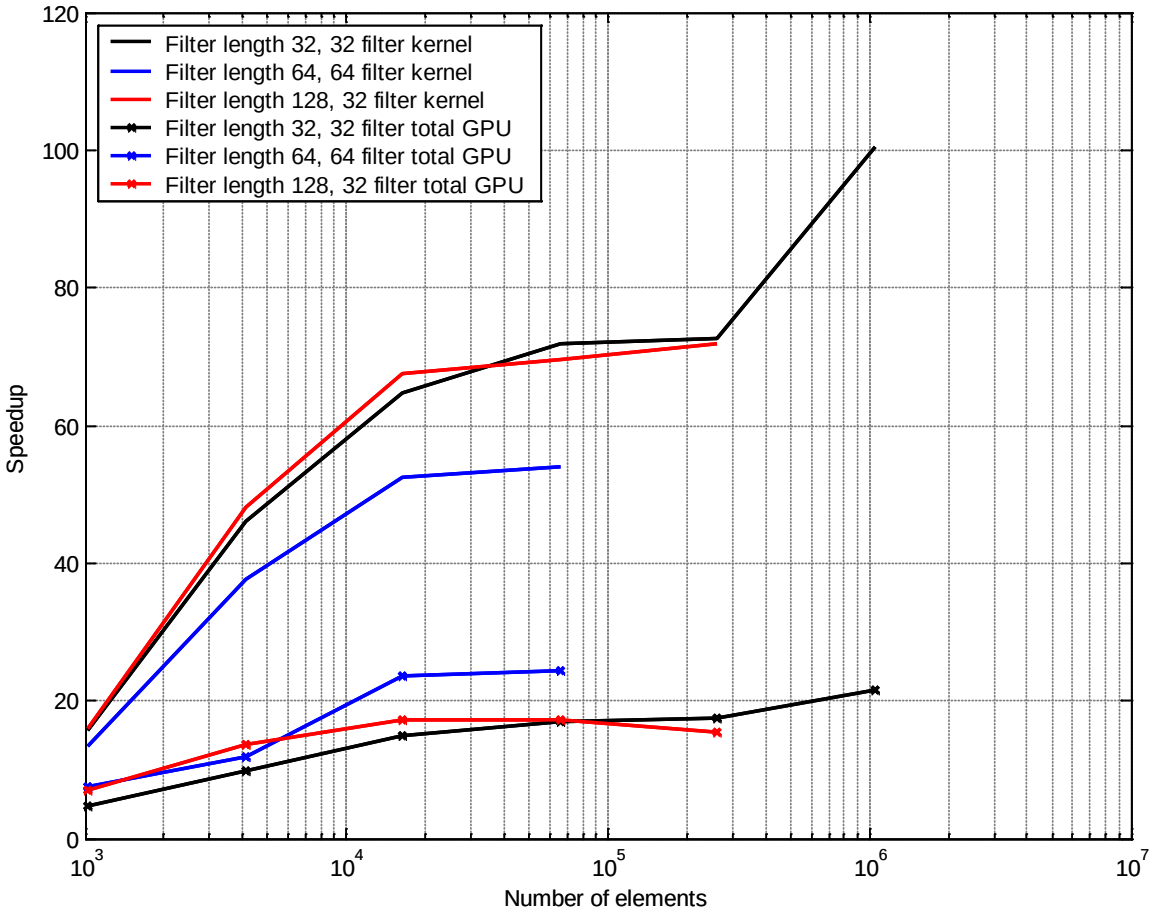

Figure 35 Speedup for TDFIR for heavy data sets.

Again, increased AI and data size leads to improved speedup for the GPU.

The performance was estimated by multiplying the length of the filters times the number of filters times the input length times the eight operations it takes to compute a complex multiplication and addition. This estimation was then divided by either the kernel time or the CPU run time.

15.2 TDFIR using OpenCL

Maximum kernel speedup over CPU: 65 times
Maximum GPU performance achieved: 116 GFLOPS
Maximum CPU performance achieved: 2.6 GFLOPS

Results:
The highest kernel speedup of the GPU implementation was about 65 times faster than the benchmark code run on the CPU.

Implementation:
TDFIR using OpenCL was implemented in the same way as the CUDA implementation in 15.1.

The OpenCL performance results compared to the CUDA implementation for varying input lengths for 3 filters are shown in Figure 36 below.

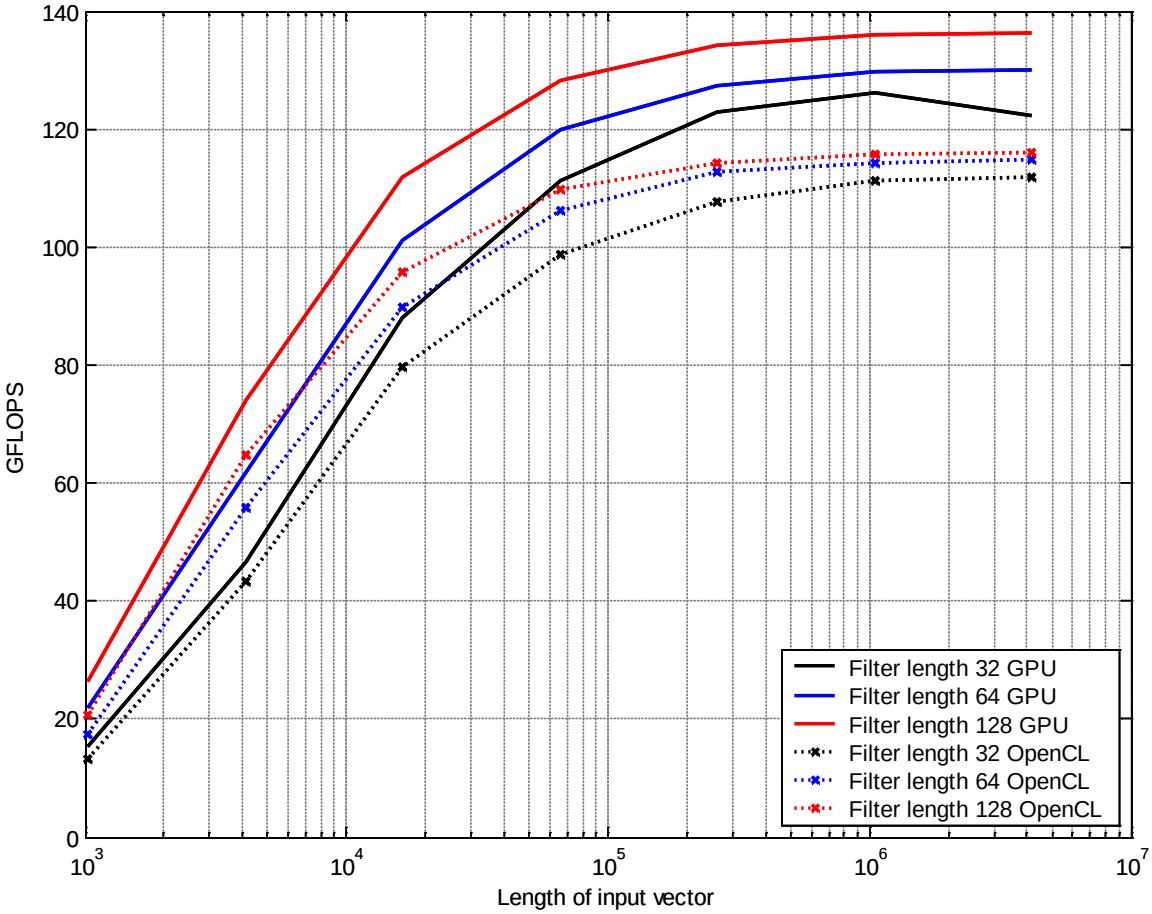

Figure 36 Performance for TDFIR using OpenCL for various filter lengths.

As can be seen in Figure 37, the OpenCL version's performance closely resembles that of the CUDA implementation, though it is always slightly lower. This is interesting considering that the kernels and hardware are basically identical. Also, as the difference in performance seems to increase in with the input length, it is unlikely that the difference in performance is due to a difference in start-up time between the two. The difference could be due to the fact that the OpenCL drivers are not of the same quality as the CUDA ones. However, it is clear that from a performance perspective, CUDA and OpenCL can perform similarly.

15.3 FDFIR

Maximum kernel speedup over CPU: Not measured
Maximum GPU performance achieved: 160 GFLOPS
Maximum CPU performance achieved: Not measured

Results:
Due to the benchmark HPEC CPU code not being able to perform the FDFIR computations on several input vectors in a reasonable manner, no comparison to the CPU was made.

Implementation:
FDFIR was implemented similar to TDFIR, but with an FFT on the input vector and inverse FFT on all the output vectors.

The performance is shown for a varying number of input vectors, number of filters, and varying both the number of vectors and number of filters in Figure 37, Figure 38 and Figure 39 respectively. No speedup results are shown because the HPEC CPU code could not handle several input vectors at a time.

Prepared (also subject responsible if other)		No.		
SMW/DD/GX Jimmy Pettersson, Ian Wainwright		5/0363-FCP1041180 en		
Approved	Checked	Date	Rev	Reference
SMW/DD/GCX Lars Henricson	DD/LX	2010-01-27	A	

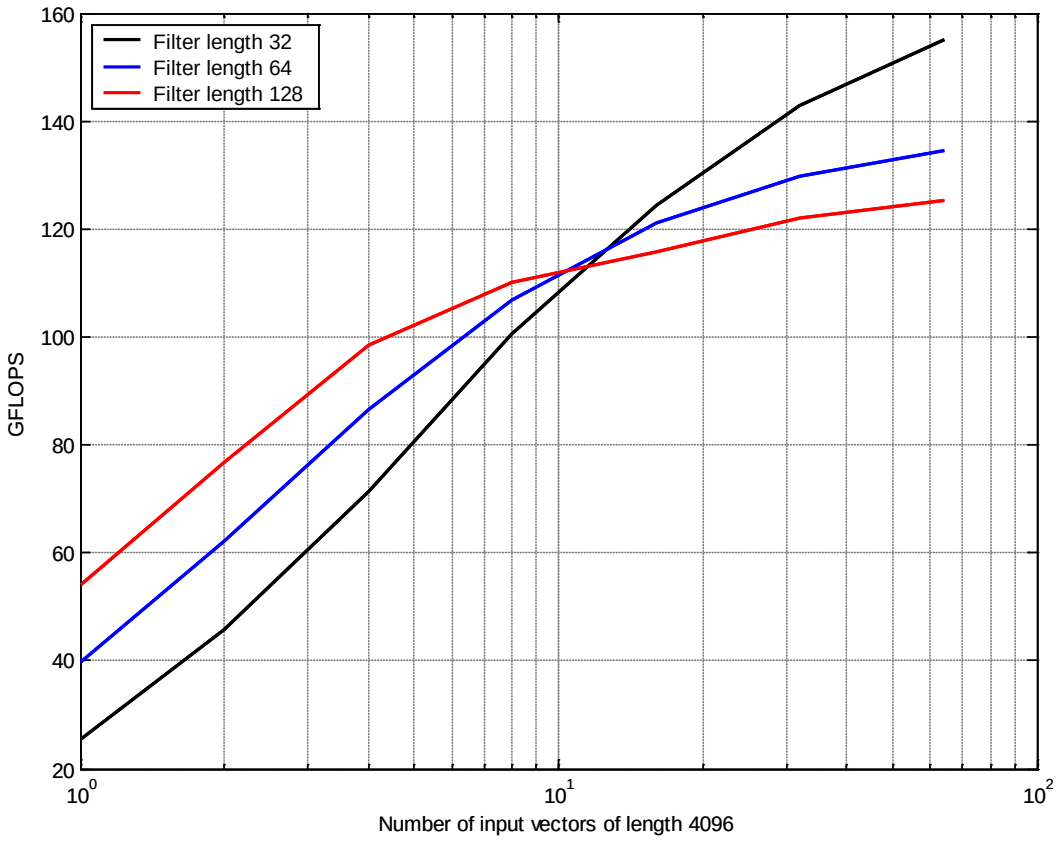

Figure 37 Performance for FDFIR for various numbers of filters.

Prepared (also subject responsible if other)		No.		
SMW/DD/GX Jimmy Pettersson, Ian Wainwright		5/0363-FCP1041180 en		
Approved	Checked	Date	Rev	Reference
SMW/DD/GCX Lars Henricson	DD/LX	2010-01-27	A	

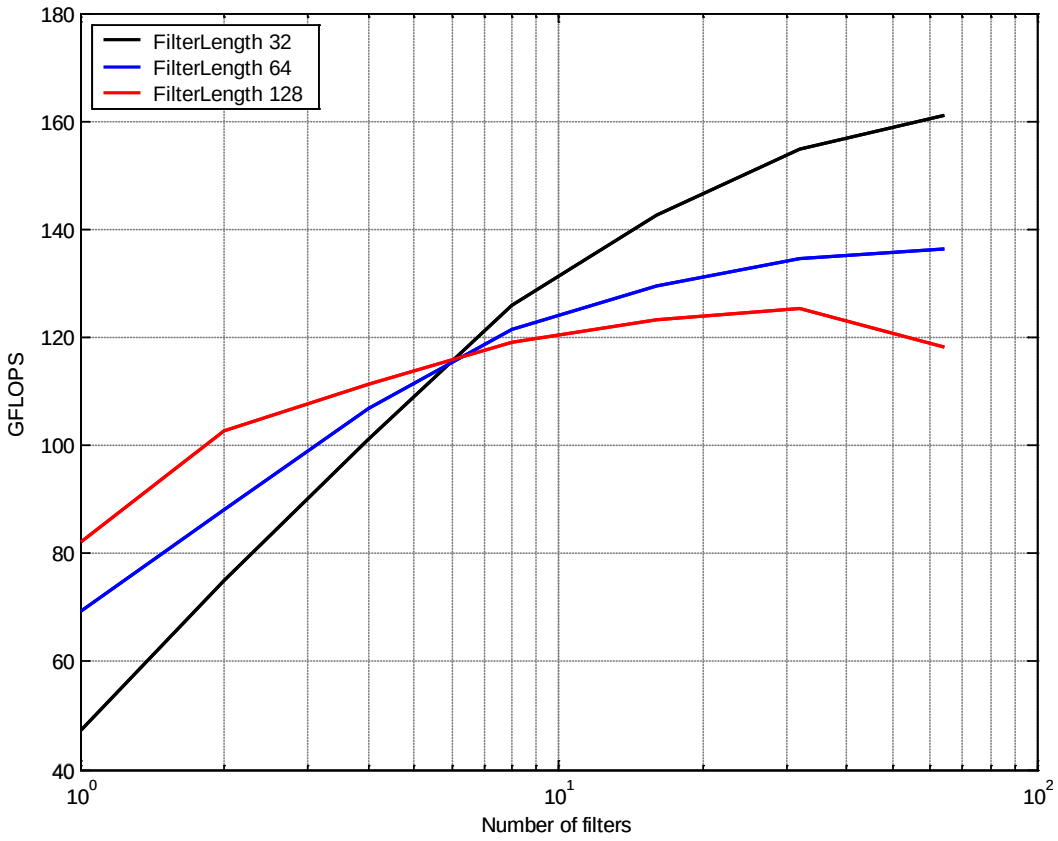

Figure 38 Performance for FDFIR for a 4096 elements long vector for a varying number of filters.

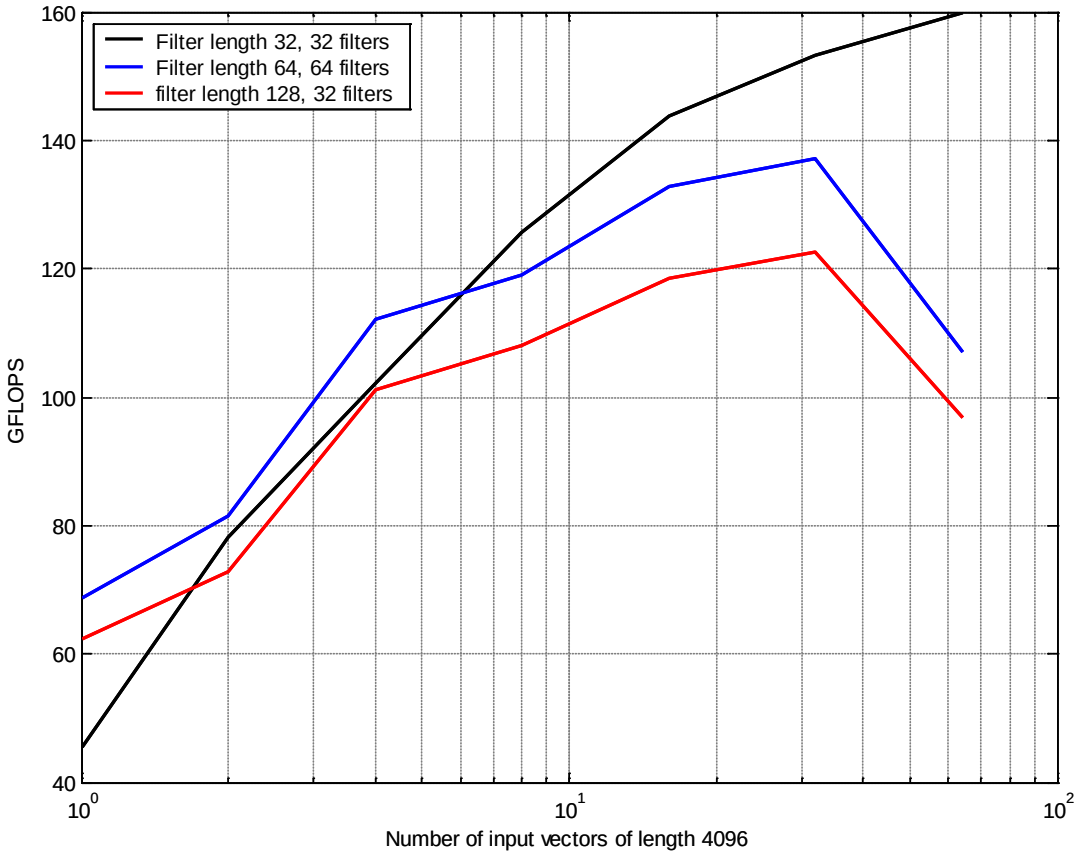

Figure 39 Performance for FDFIR for a varying number of 4096 elements long vectors for a varying number of heavy filters.

Clearly, the GPU benefits from increased problem size and increased AI. A closer inspection of the FFT algorithm is given in 15.13. As with TDFIR, the performance does not always increase with increased AI as can be seen in Figure 37 and Figure 38. The reason for this is that the filter is places in the on-chip constant cache. For larger filters, the cache is quickly filled which means that values have to be re-read into the cache, hence leading to lower performance for larger and more numerous filters.

The notable drop for two of the lines in Figure 39 is presumably due to the operating system needing to update the screen as when looking at the CUDA Visual Profiler output, a large gap can be seen in which the kernel is idle for a period of time. This effect occurred roughly 90% if the time the benchmark was run for large sizes.

The performance was calculated by summing the number of FLOP in the FFT calculations and the TDFIR part. This sum was then divided by the kernel time.

15.4 QR-decomposition

Maximum kernel speedup over CPU: 1
Maximum GPU performance achieved: 190 GFLOPS
Maximum CPU performance achieved: 3.5 GFLOPS

Results:
The highest kernel speedup of was over 200 times faster than the benchmark code run on the CPU. The highest speedup including all memory transfers was also over 200 times. However, these results were achieved for very large matrices not applicable to radar signal processing. Hence, a benchmark against Intel's MKL is also performed.

Implementation:
The QRD algorithm is detailed in 14.5. On the GPU, the benchmark was implemented by using the CULAtools library. Single, large QRDs are very suitable for CUDA, given that they have a complexity of n^3. However, the CULAtools library can only compute one QRD at a time since each QRD computes as a separate kernel and only one kernel can run at a time. This makes using CULAtools useful only for large matrices, not many small ones.

Performance, and speedup vs. both. HPEC and MKL are shown in Figure 40, Figure 41 and Figure 42 respectively.

Prepared (also subject responsible if other)		No.		
SMW/DD/GX Jimmy Pettersson, Ian Wainwright		5/0363-FCP1041180 en		
Approved	Checked	Date	Rev	Reference
SMW/DD/GCX Lars Henricson	DD/LX	2010-01-27	A	

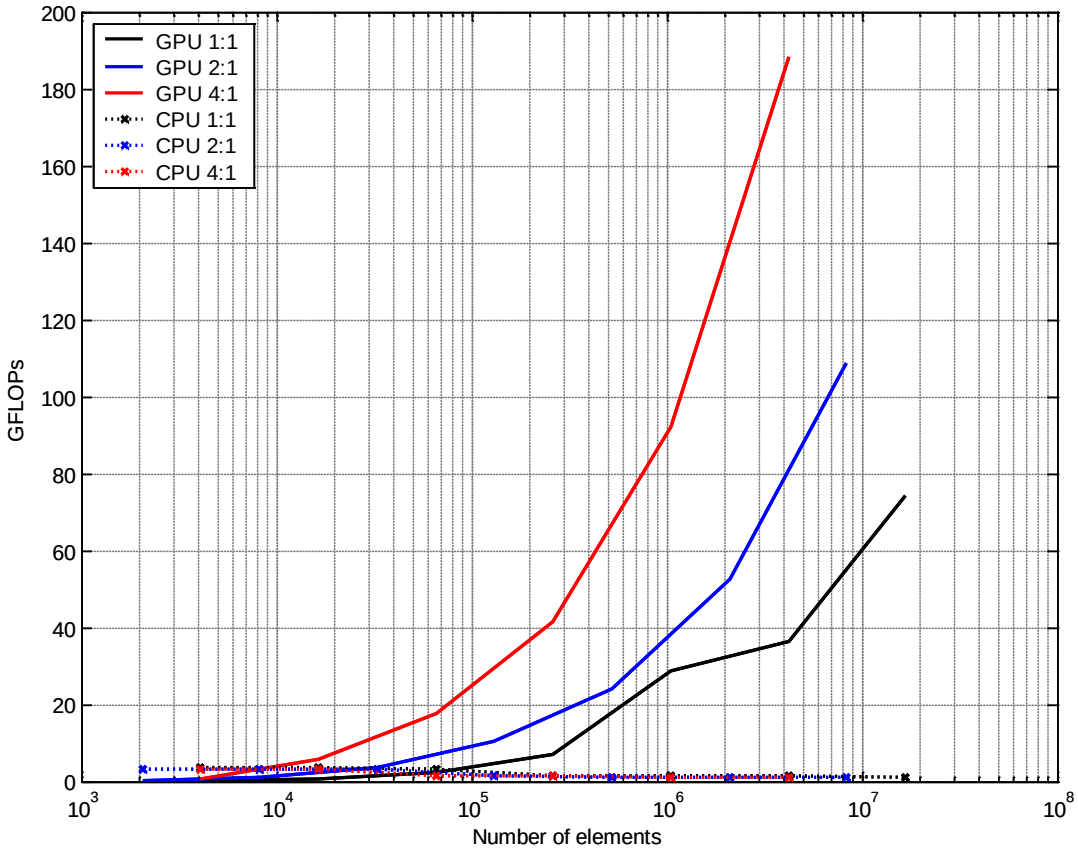

Figure 40 Performance of QRD for different row by column ratios.

As can be seen from Figure 40, the GPU greatly benefits from increased matrix size. However, for square matrices smaller than 150*150, the CPU is faster.

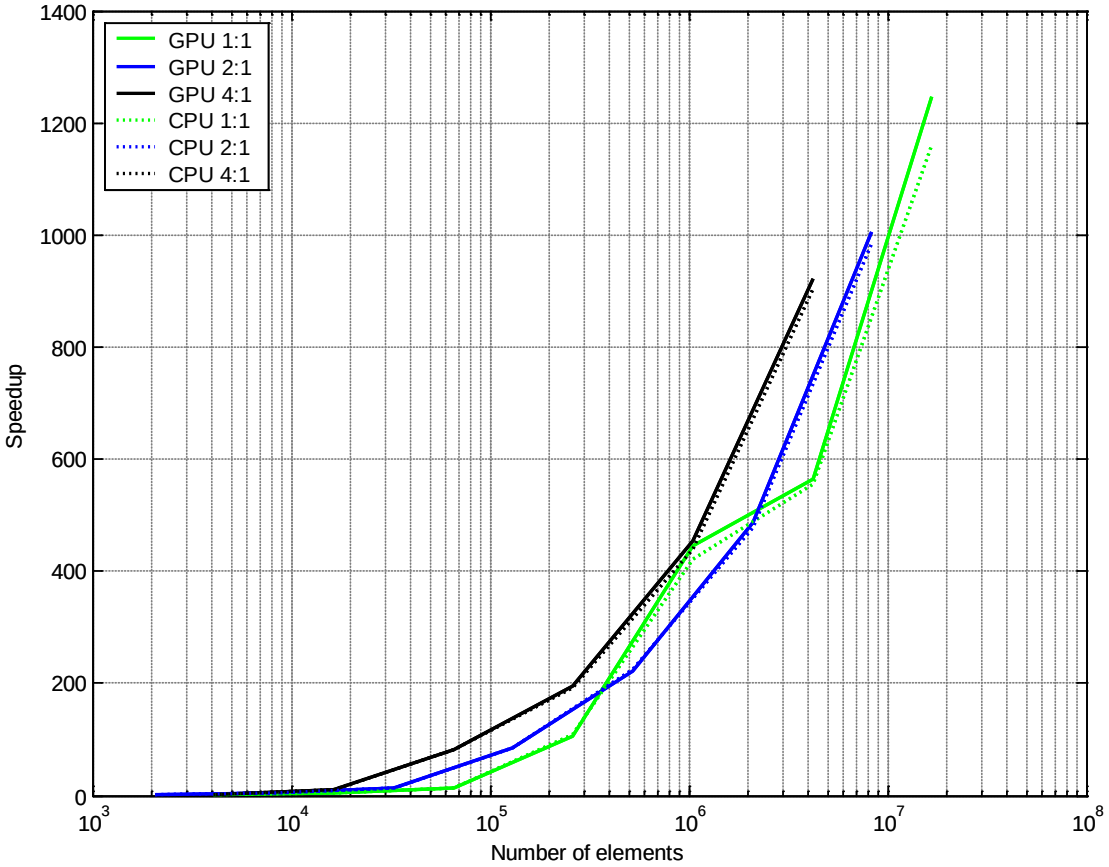

Figure 41 Kernel speedup for QRD against HPEC CPU implementation.

The incredible speedup shown in Figure 41 was achievable partly due to the fast GPU implementation but mainly due to the HPEC code performing poorly to say the least. A more relevant comparison is made against Intel's MKL, shown in Figure 42 below.

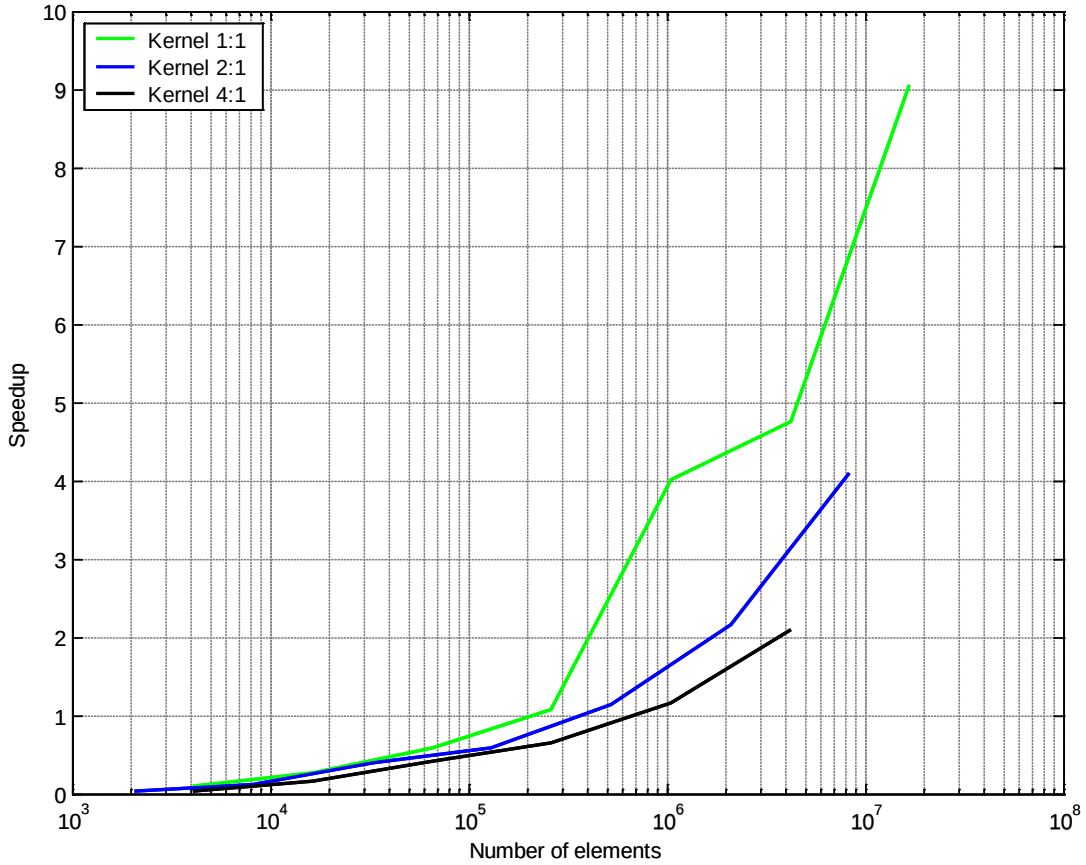

Figure 42 Kernel speedup for QRD against Intel's Math Kernel Library (MKL) CPU implementation.

As can be seen Figure 42, the GPU performs relatively better for larger matrices. However, in the range relevant for radar applications, roughly [200, 100] or 2 * 10^4 elements, the CULAtools library is slower. Also, as the CULAtools library only can compute one matrix at a time, the CULAtools QRD implementation is not suitable for real-time computations where many small QRD are to be computed simultaneously.

$$workload = 16 N_{row}^2 N_{col} - \frac{8}{3} N_{col}^3$$

$$performane = \frac{workload}{time}$$

15.5 Singular Value Decomposition: SVD

Maximum kernel speedup over CPU (HPEC): 60
Maximum GPU performance achieved: 33 GFLOPS
Maximum CPU performance achieved: 1.6 GFLOPS

Results:
The highest kernel speedup was 60 times faster than the benchmark code run on the CPU. The highest speedup including all memory transfers was 59. However, these results were achieved for very large matrices not applicable to radar signal processing.

Implementation:
The SVD algorithm is detailed in 14.6. On the GPU, the SVD benchmark was implemented by using the CULAtools library. Single, large SVDs are suitable for CUDA, given that they have a complexity of n^3. However, the CULAtools library used can only compute one SVD at a time since each SVD computes as a separate kernel and only one kernel can run at a time.

Performance, speedup vs. both. HPEC and MKL, and lastly the performance for a square matrix for larger sizes running on the GPU are shown in Figure 43, Figure 44, Figure 45 and Figure 46 respectively.

Prepared (also subject responsible if other)		No.		
SMW/DD/GX Jimmy Pettersson, Ian Wainwright		5/0363-FCP1041180 en		
Approved	Checked	Date	Rev	Reference
SMW/DD/GCX Lars Henricson	DD/LX	2010-01-27	A	

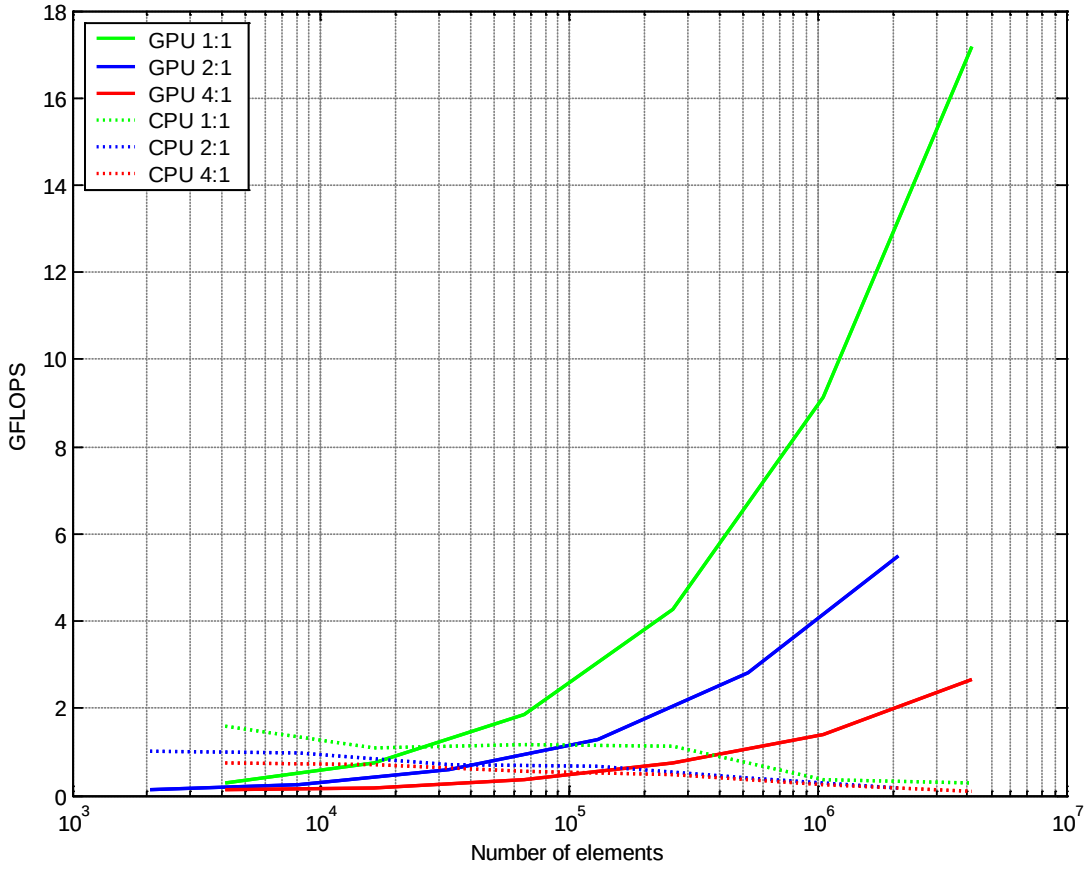

Figure 43 Performance for SVD for different input sizes and varying row by column ratios.

Prepared (also subject responsible if other)		No.		
SMW/DD/GX Jimmy Pettersson, Ian Wainwright		5/0363-FCP1041180 en		
Approved	Checked	Date	Rev	Reference
SMW/DD/GCX Lars Henricson	DD/LX	2010-01-27	A	

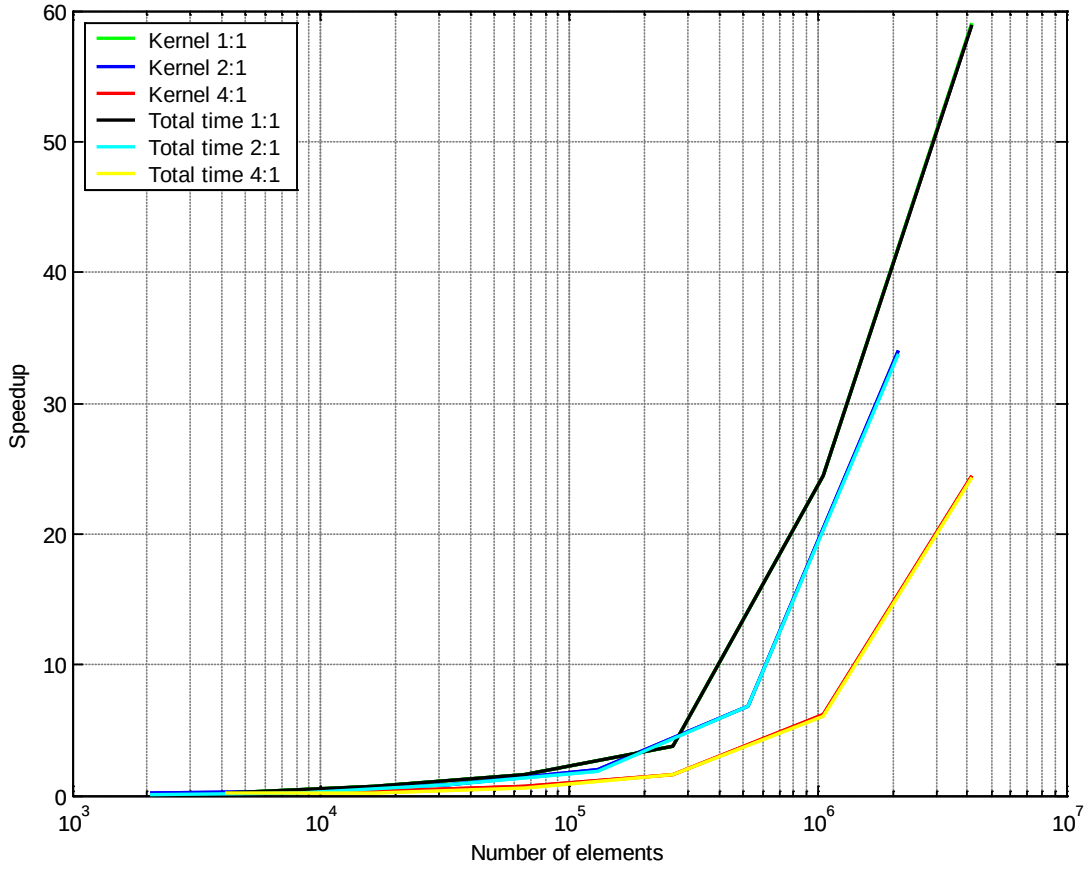

Figure 44 Speedup for SVD against HPEC for both the kernel and the total GPU time for different input sizes and varying row by column ratios.

Prepared (also subject responsible if other)		No.		
SMW/DD/GX Jimmy Pettersson, Ian Wainwright		5/0363-FCP1041180 en		
Approved	Checked	Date	Rev	Reference
SMW/DD/GCX Lars Henricson	DD/LX	2010-01-27	A	

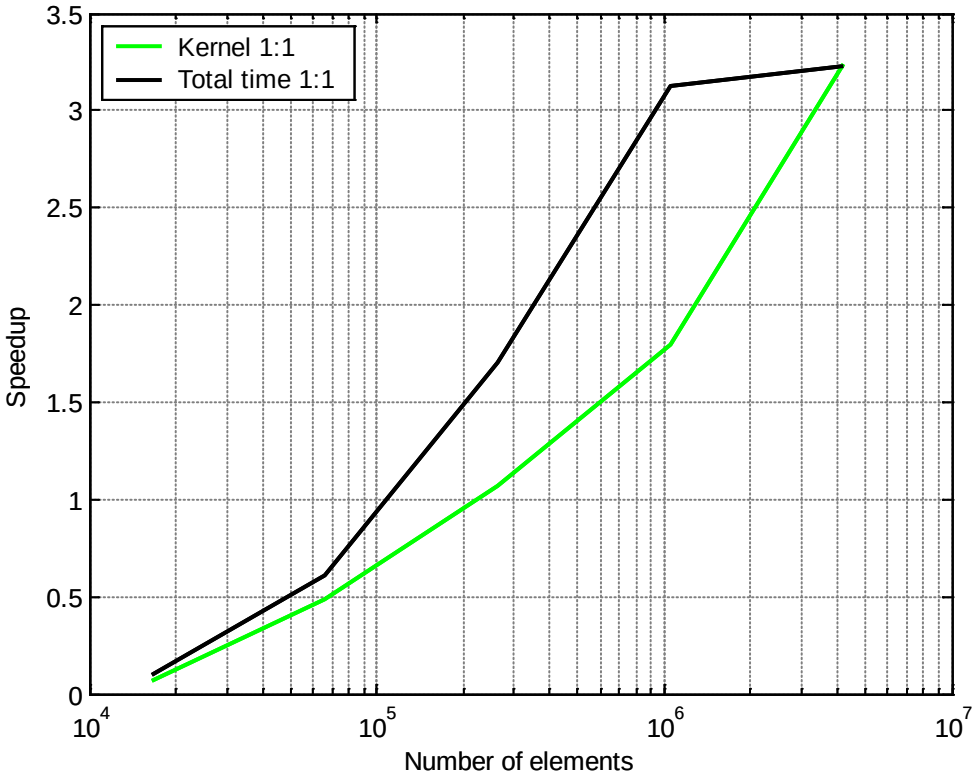

Figure 45 Speedup for SVD against MKL for both the kernel and the total GPU time for different input sizes for square matrices.

From the figures above, we can first see that the GPU benefits from increased problem sizes, while also performing worse for smaller ones. Also, the CPU performs worse as the matrix size increases. This is likely due to the matrix not fitting in the L2 cache. This is clearly noticeable in Figure 43 when the matrix size increases from roughly 10^5 elements, or 0.8 MB, to 10^6 elements, or 8 MB. As the matrix no longer fits in the 6 MB L2 cache, the CPU must constantly read and write to the CPU RAM, which is very costly. Also, it is easy to see that the lion's share of the time is spent computing, not transferring data between the host and device, as both the kernel and the total time are near identical.

Figure 46 Performance for SVD for a large square matrix of varying size.

Even though the GPU performs reasonably well for large matrices, in the range relevant for radar applications of roughly [200, 100] or $2 * 10^4$ elements, the CULAtools library is not faster. Also, as the CULAtools library only can compute one matrix at a time, the CULAtools SVD implementation is not suitable for real-time computations where many small QRD are to be computed simultaneously.

The performance was estimated according to

$$Workload = \frac{112}{3} * {N_{col}}^3 + 24 * {N_{col}}^2 + 31 * N_{col}$$

$$performance = \frac{workload}{time}$$

15.6 Constant False-Alarm Rate: CFAR

Maximum kernel speedup over CPU: 30
Maximum GPU performance achieved: 60 GFLOPS
Maximum CPU performance achieved: 2.6 GFLOPS

Results:
The highest kernel speedup was 30 times faster than the benchmark code run on the CPU. The highest speedup including all memory transfers was 1.1.

Implementation:
The CFAR algorithm is detailed in 14.7. The implemented version of the algorithm is suitable for the GPU as it is embarrassingly parallel. However, the AI is not especially high. As the total GPU speedup is only 1.1, this kernel by itself does not justify a GPU implementation. This is due to the low bandwidth host-device transfer. However, for radar applications, it is to be noted that CFAR and STAP use the same data, so the cost of transferring data to the GPU can be amortized over both kernels.

The data sets used in the benchmark are shown in Table 11 below.

	Number of beams	Number of range gates	Number of Doppler bins	Number of guard cells
Data set 1	16	64	24	4
Data set 2	48	3500	128	8
Data set 3	48	1909	64	8
Data set 4	16	9900	16	16

Table 11 Data sets used in CFAR.

The speedup, performance, and also a comparison to results produced by Georgia Tech for the various data sets are shown in Figure 47, Figure 48 and Figure 49 respectively.

Prepared (also subject responsible if other)		No.		
SMW/DD/GX Jimmy Pettersson, Ian Wainwright		5/0363-FCP1041180 en		
Approved	Checked	Date	Rev	Reference
SMW/DD/GCX Lars Henricson	DD/LX	2010-01-27	A	

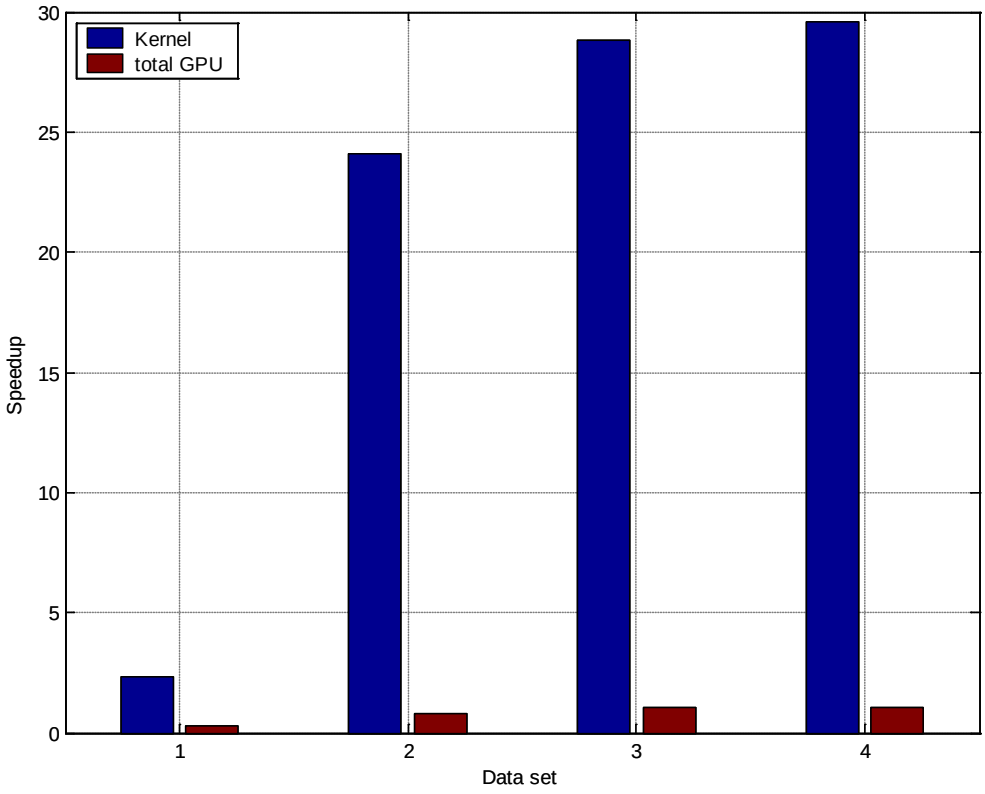

Figure 47 Speedup for CFAR for both the kernel and the total GPU time for the different data sets.

To receive a decent speedup, a sufficiently high workload is needed. Hence, the small workload of Data set 1 leads to a smaller speedup than the other data sets.

Prepared (also subject responsible if other)		No.		
SMW/DD/GX Jimmy Pettersson, Ian Wainwright		5/0363-FCP1041180 en		
Approved	Checked	Date	Rev	Reference
SMW/DD/GCX Lars Henricson	DD/LX	2010-01-27	A	

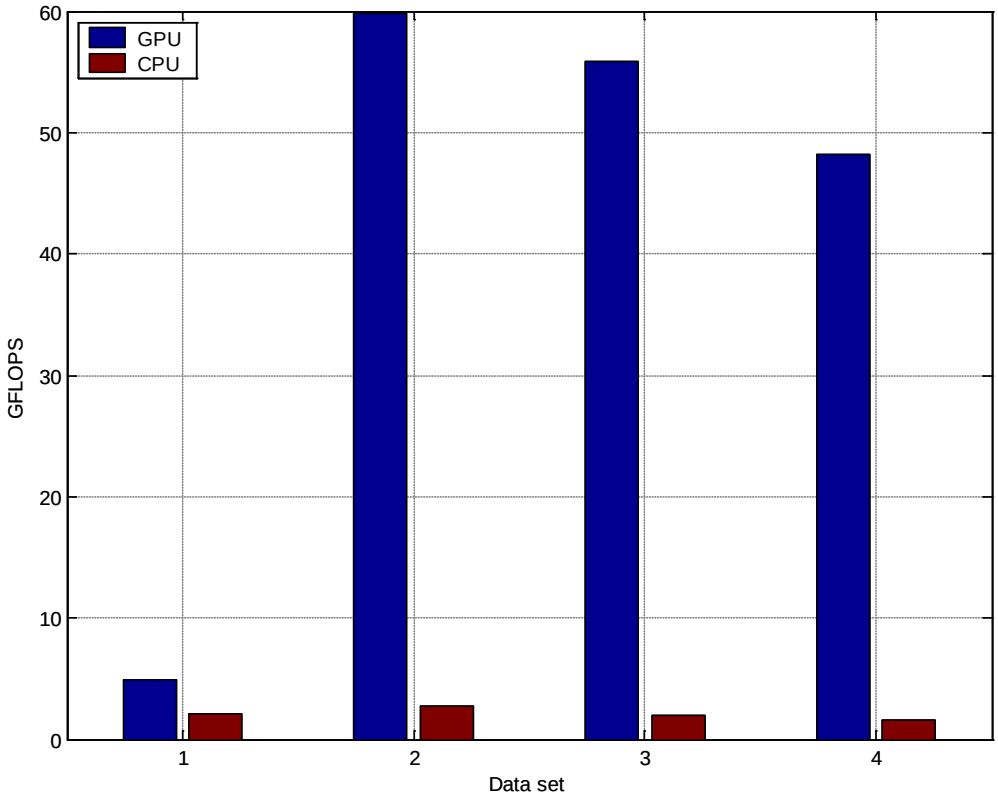

Figure 48 Performance for CFAR for different data sets.

The performance was estimated using MIT's MATLAB code which also generates the data sets tested. It includes a function which gives the workload depending on how the data set is designed.

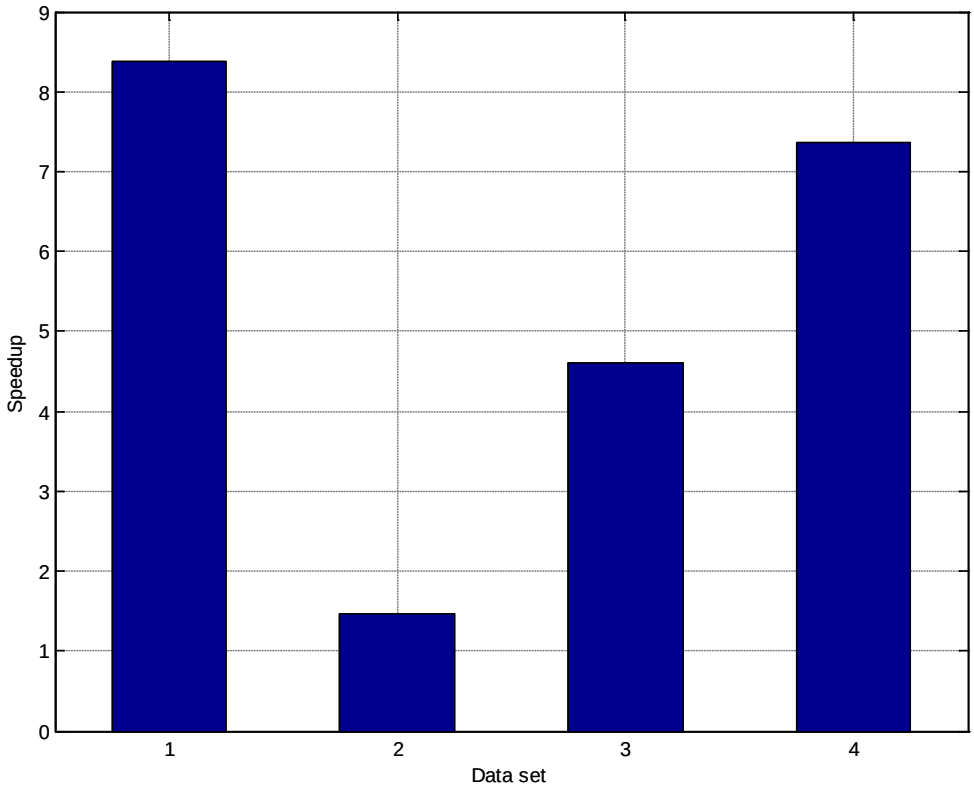

Figure 49 The speedup over Georgia Tech's CUDA implementations for CFAR.

Worth noting when comparing to Georgia Tech's test results [31] is that the level of speedup varies notably for the different data sets. This might well be due to the difficulties with non-coalesced global memory access as described in 7.3.1.1 for the GPU Georgia Tech utilizes. Other than the difficulties with non-coalesced memory access, the differences are lower theoretical performance reaching 345 GFLOPS compared to the GTX 260's 480, and a higher theoretical bandwidth, reaching 86.4 GB/s compared to the GTX 260's 76.8. The variations in speedup might also be due to implementation differences.

15.7 Corner turn

Maximum kernel speedup over CPU: 260 times
Maximum GPU throughput achieved: 28[17] GB/s
Maximum CPU throughput achieved: 4.1 GB/s

[17] As explained below, this represents 75% of the theoretical bandwidth being used.

Results:
The highest kernel speedup was 260 times faster than the benchmark code run on the CPU. The highest speedup including all memory transfers was 7. The obvious bottleneck here is the device to host transfer.

Implementation:
The CT algorithm is detailed in 14.8. On the GPU, the implemented version of corner turn involves many interesting features of dealing with both on-chip and off-chip memory. The features that have to be considered are coalesced transposing, bank conflicts, and maybe most interesting due to its limited documentation, partition camping. After addressing these issues, a very high percentage of the GPUs theoretical bandwidth is used. The throughput, which is half the effective bandwidth used as data is transported to and from the GPU, kernel and total speedup are shown in Figure 50, Figure 51 and Figure 52 respectively.

Prepared (also subject responsible if other)		No.		
SMW/DD/GX Jimmy Pettersson, Ian Wainwright		5/0363-FCP1041180 en		
Approved	Checked	Date	Rev	Reference
SMW/DD/GCX Lars Henricson	DD/LX	2010-01-27	A	

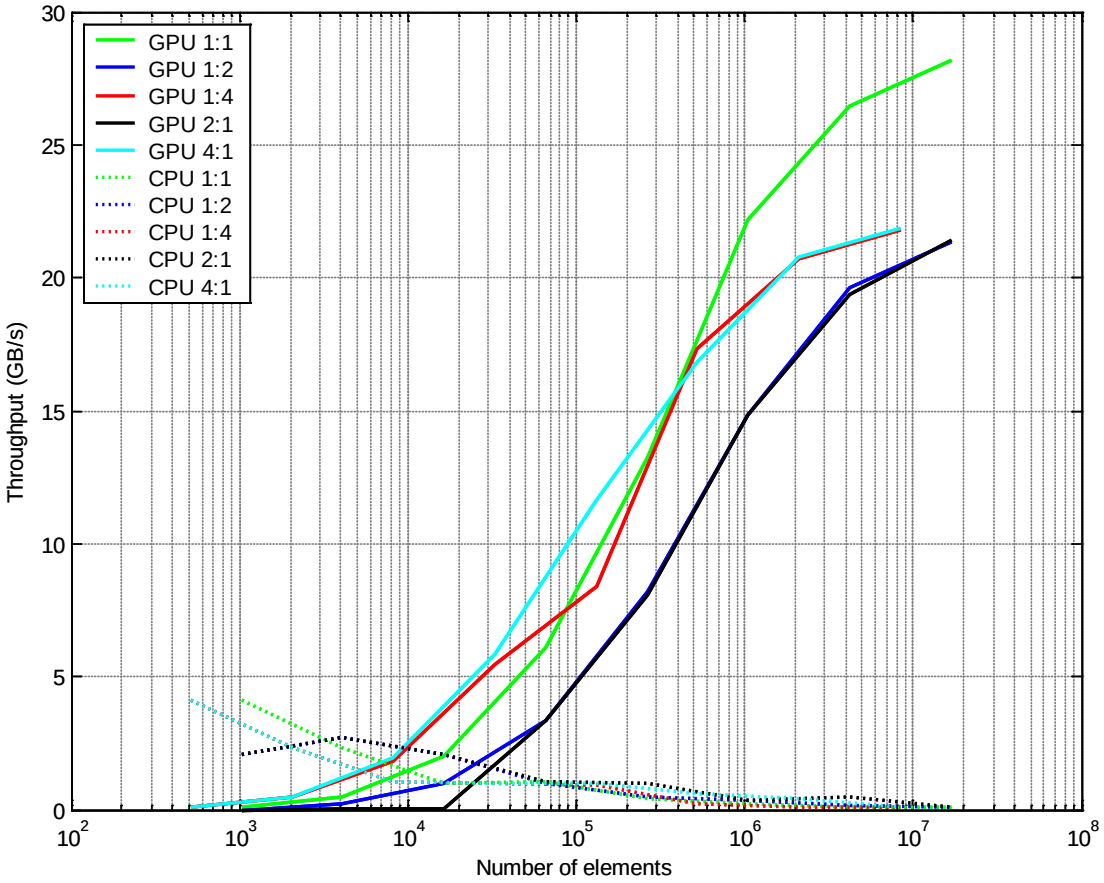

Figure 50 Throughput for CT for different input sizes and varying row by column ratios. Note that the effective bandwidth is twice that of the throughput for this algorithm.

Prepared (also subject responsible if other)		No.		
SMW/DD/GX Jimmy Pettersson, Ian Wainwright		5/0363-FCP1041180 en		
Approved	Checked	Date	Rev	Reference
SMW/DD/GCX Lars Henricson	DD/LX	2010-01-27	A	

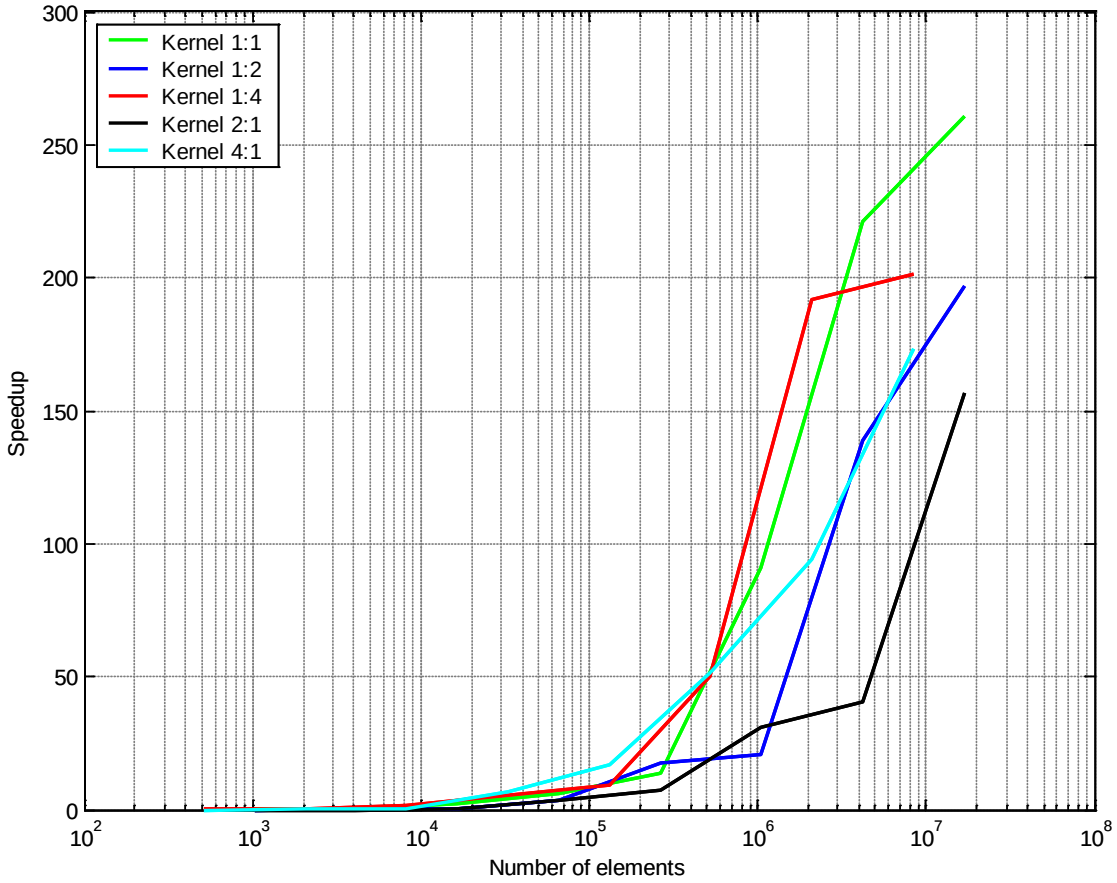

Figure 51 Kernel speedup for CT for different row by column ratios.

Prepared (also subject responsible if other)		No.		
SMW/DD/GX Jimmy Pettersson, Ian Wainwright		5/0363-FCP1041180 en		
Approved	Checked	Date	Rev	Reference
SMW/DD/GCX Lars Henricson	DD/LX	2010-01-27	A	

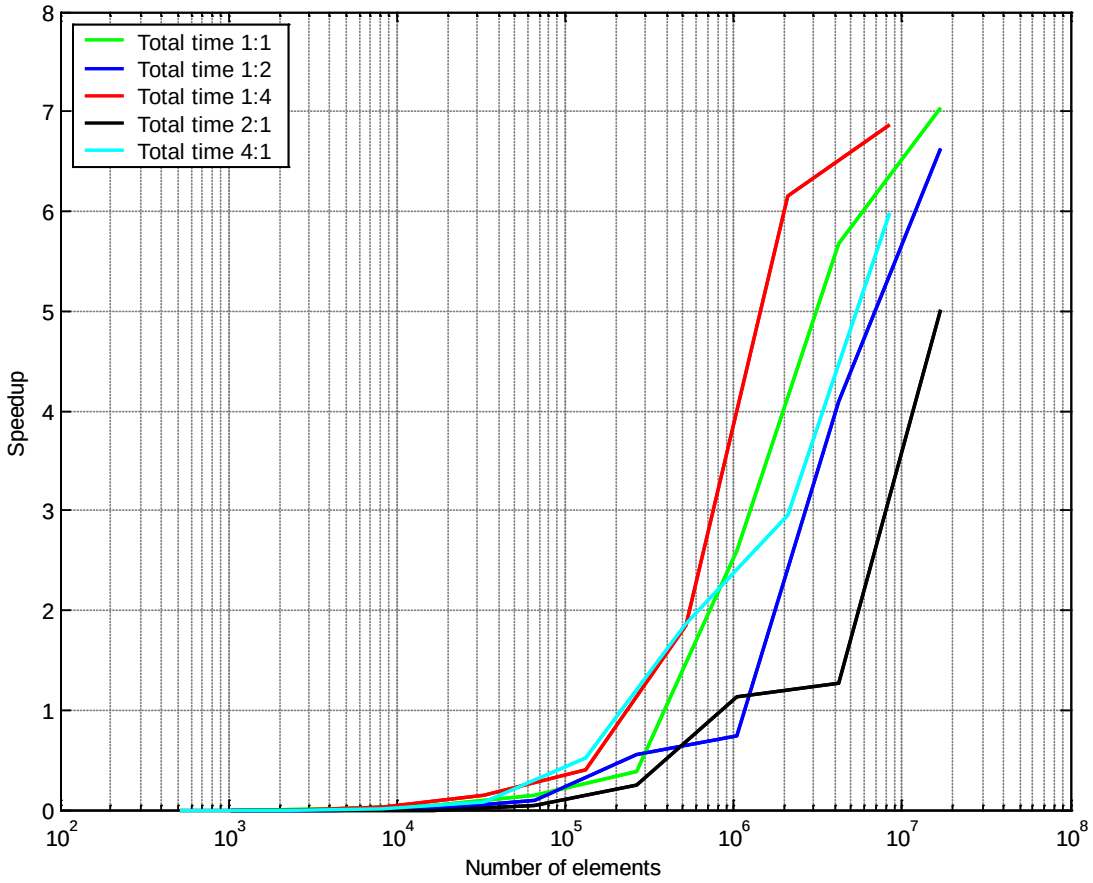

Figure 52 Total GPU speedup for CT for different input sizes and varying row by column ratios.

As seen above, the GPU has a clear advantage over the CPU for larger matrices. A sharp turn in the speedup is seen around the 6 MB mark, roughly 10^6 elements. This is likely due to the matrix not being able to fit in the L2 cache, in which case the CPU must read and write to the CPU RAM unnecessarily often due to the non-cache blocked nature of the CPU code. The CPU performance could in theory hence be increased by working on small blocks at a time instead of using the naïve implementation as the HPEC code does.

When including the data transfers between host and device, as shown in Figure 52, the pattern looks a lot like the kernel speedup shown in Figure 51, but more compressed. This is due to the fact that the AI is constant for corner turn and the problem behaves identically irrespective or the matrix size; the complexity is linear to the problem size. Hence including the memory transfers only adds a constant factor to the time. This constant factor added to all the kernel times is basically only data size dependant, and hence equal for all kernels of the same size. It is also substantially larger than the kernel time, which is why the earlier large ratios between different kernels are compressed when measuring the total time.

Furthermore, the GPU implementation performs near identically for mirrored ratios, i.e. 1:2 to 2:1, and 1:4 to 4:1. The CPU however does not, though why it is so is not clear.

Worth noting is that this kernel is only run in combination with other algorithms, transposing data in memory to increase access speed to memory for them. Hence, the total GPU speedup can largely be seen as irrelevant as the time to transfer data to the GPU will be shared between all algorithms using the data.

The throughput was estimated by dividing the data size by the time taken to complete the corner turn. In other words, the throughput is half of the effective bandwidth as the matrix is transferred twice; once in each direction. The throughput is hence the amount of data corner turn in GB per second, not the total amount of data transferred.

The 28 GB/s throughput is actually 56 GB/s of effective data transferred. This is roughly 75% of the graphics board's theoretical peak of 78.6 GB/s.

15.8 INT-Bi-C: Bi-cubic interpolation

Maximum kernel speedup over CPU: 32
Maximum GPU performance achieved: 100 GFLOPS
Maximum CPU performance achieved: 3.5 GFLOPS

Results:
The highest kernel speedup was 32 times faster than the benchmark code run on the CPU. The highest speedup including all memory transfers was 3.

Implementation:
The Bi-cubic interpolation algorithm is detailed in 14.9. On the GPU, several variants were implemented with different amounts of data reuse.

The speedup and performance for the various implementations for various numbers of interpolations are shown in Figure 53 and Figure 54. Without going in to too much detail, higher implementation numbers mean a higher degree of data reuse; i.e. a higher AI.

Prepared (also subject responsible if other)		No.		
SMW/DD/GX Jimmy Pettersson, Ian Wainwright		5/0363-FCP1041180 en		
Approved	Checked	Date	Rev	Reference
SMW/DD/GCX Lars Henricson	DD/LX	2010-01-27	A	

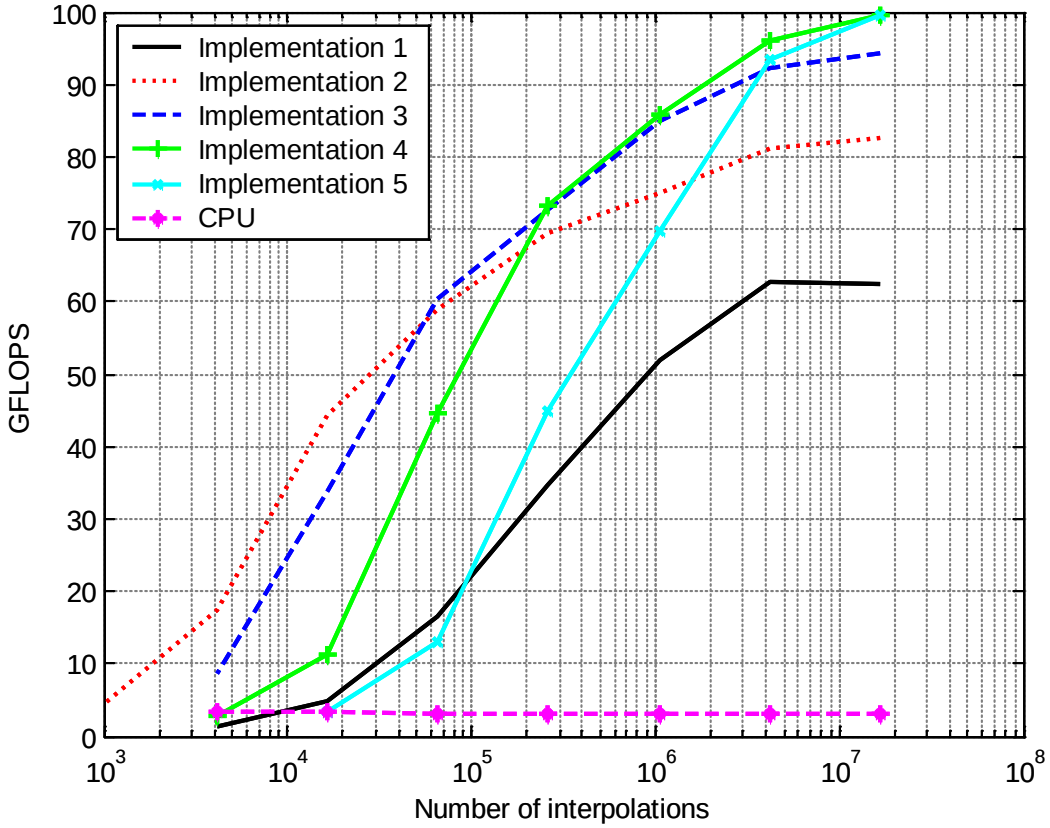

Figure 53 Performance for INT-Bi-C for the 3 different implementations for various numbers of interpolations. Higher implementation number indicates a higher reuse of data.

Prepared (also subject responsible if other)		No.		
SMW/DD/GX Jimmy Pettersson, Ian Wainwright		5/0363-FCP1041180 en		
Approved	Checked	Date	Rev	Reference
SMW/DD/GCX Lars Henricson	DD/LX	2010-01-27	A	

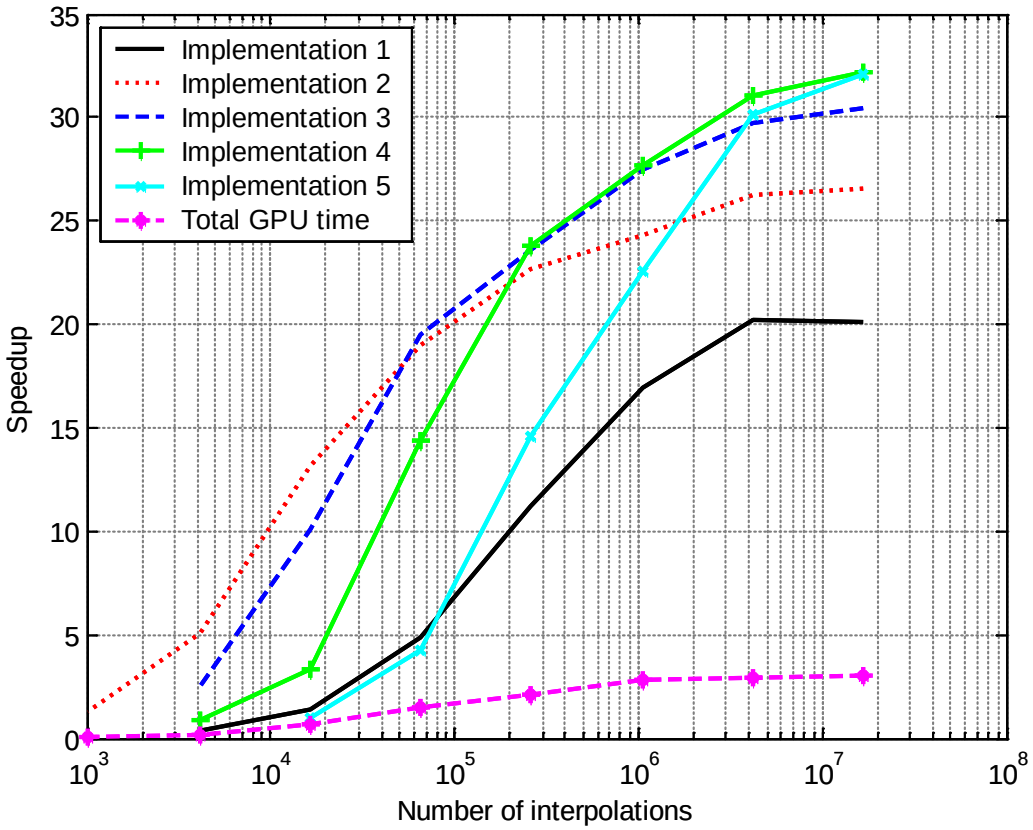

Figure 54 Speedup for INT-Bi-C for both the kernel and the total GPU time for the 3 different implementations for various numbers of interpolations. Higher implementation number indicates a higher reuse of data.

The different speedups can be attributed to the different AI for the various implementations. Also, larger problems give higher performance and speedup factors for the GPU, not the CPU.

The implemented versions of the algorithm are very suitable for CUDA as they are all embarrassingly parallel as all interpolations are independent of each other, have nice global memory access patterns, and some have a reasonably high AI. The total GPU speedup of 3 also justifies bi-cubic interpolation as a stand-alone kernel for large problems, even if data is not used in other kernels or streamed.

The performance was estimated from their being 5 interpolations of 4 elements per bi-cubic interpolation; 4 in the x-dimension, followed by 1 in the y-dimension. Each of the 5 interpolations consists of 14 floating point operations. This gives a total of 70 operations per bi-cubic interpolation. This estimation was then multiplied by the number of bi-cubic interpolations and then divided by either the kernel time or the CPU run time.

15.9 INT-C: Neville's algorithm

Maximum kernel speedup over CPU: 25
Maximum GPU performance achieved: 100 GFLOPS
Maximum CPU performance achieved: 4.7 GFLOPS

Results:
The highest kernel speedup was 25 times faster than the benchmark code run on the CPU. The highest speedup including all memory transfers was 1.1.

Implementation:
Neville's algorithm is detailed in 14.10. On the GPU, an interpolation was performed for points 1-4, 2-5, 3-6, 4-7, 5-8, and so on, i.e. element 5, for instance, was used for four different interpolations. The *x* and *y*-values were hence stored in shared memory to minimize the number of global reads required. All the values were stored as long vectors; one for *x*-values, one for *y*-values and one for the points of interpolation. The values were stored linearly in memory to easily allow for coalesced memory reads.

The performance and speedup for various numbers of interpolations are shown in Figure 55 and Figure 56 below.

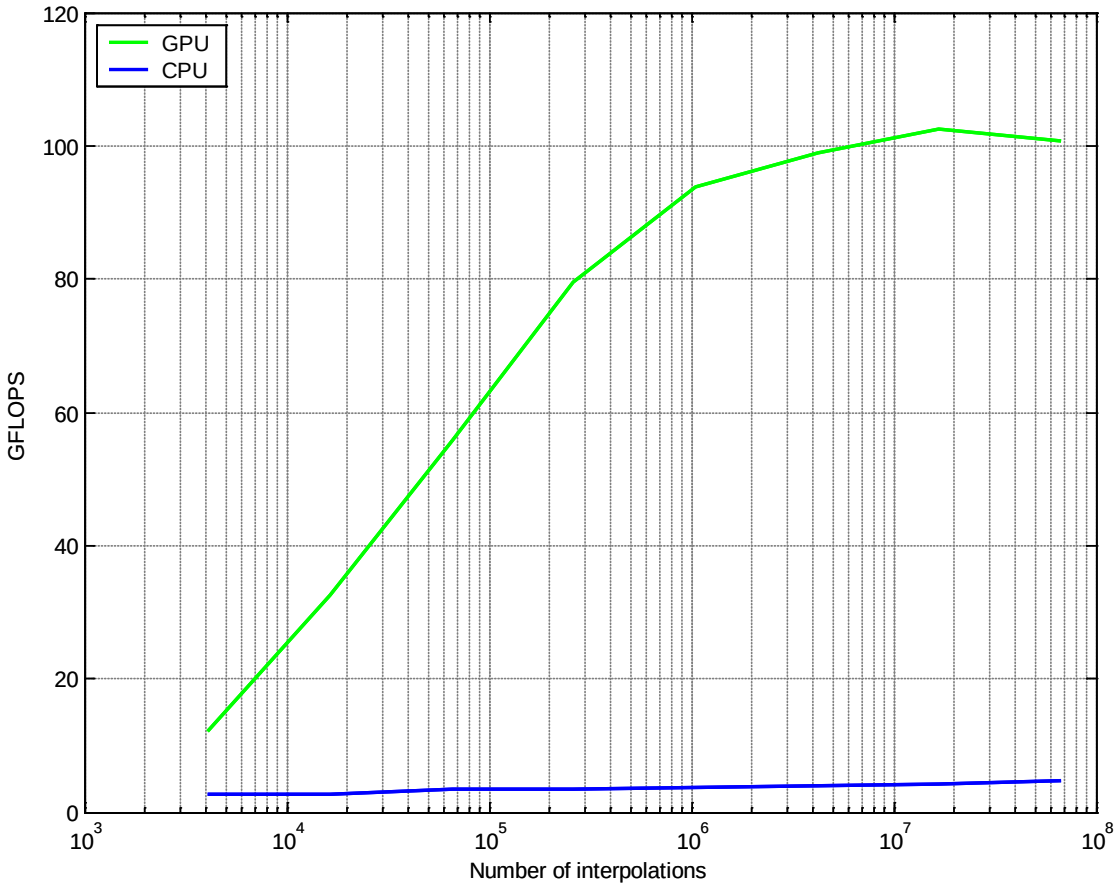

Figure 55 Performance for INT-C for a varying number of interpolations.

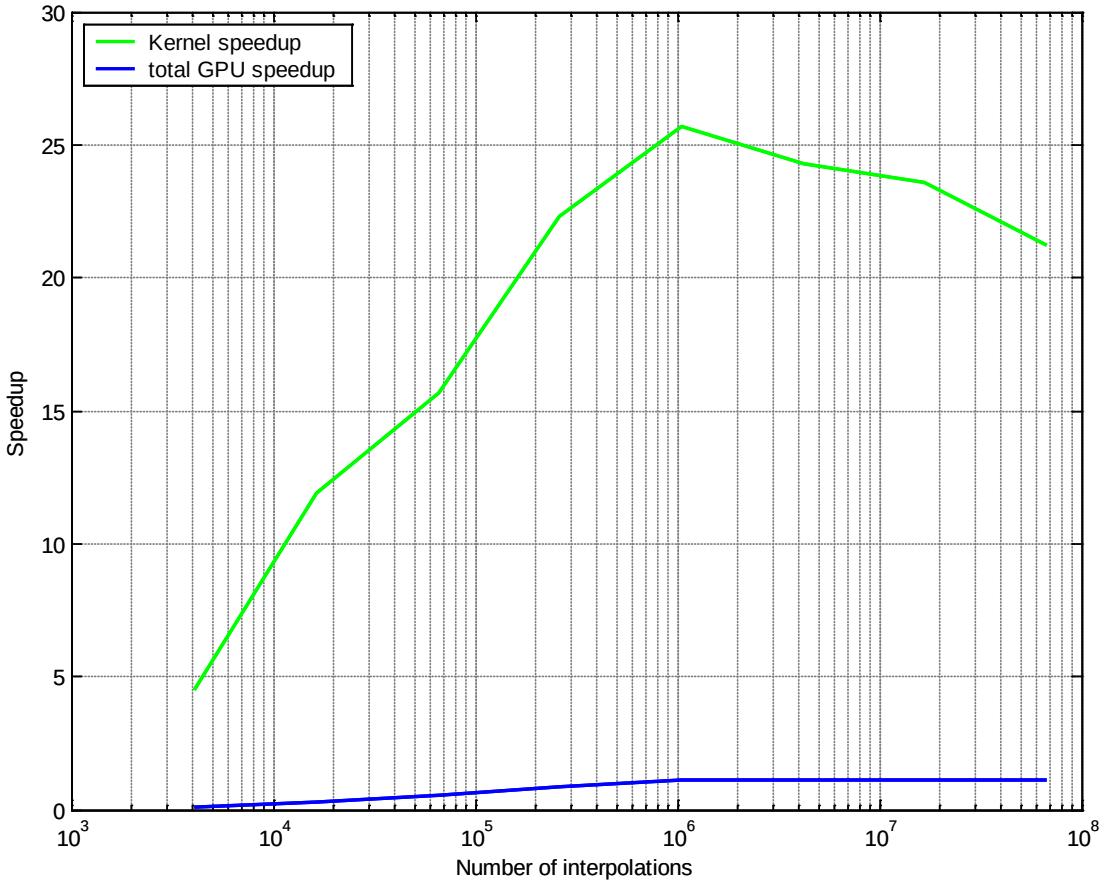

Figure 56 Kernel speedup for INT-C for a varying number of interpolations.

Clearly, the GPU benefits from an increased problem size. The fact that the kernel speedup decreases for the largest values are not due to the GPU running slower, but rather the CPU running faster, as can be seen in Figure 55.

The implemented version of the algorithm is very suitable for CUDA as it is embarrassingly parallel, has nice global memory access patterns, and has a reasonably high AI. However, the total GPU speedup is not high enough to justify its use as a stand-alone kernel when considering speedup only.

The performance was estimated from their being 34 operations per interpolation. This estimation was then multiplied by the number of interpolations and then divided by either the kernel time or the CPU run time.

15.10 Int-C: Neville's algorithm with OpenCL

Maximum kernel speedup over CPU: 18 times
Maximum GPU performance achieved: 70 GFLOPS
Maximum CPU performance achieved: 2.6 GFLOPS

Results:
The highest kernel speedup of the GPU implementation was about 18 times faster than the benchmark code run on the CPU.

Implementation:
Neville's algorithm using OpenCL was implemented in the same way as the CUDA implementation in 15.9.

The OpenCL performance results compared to the CUDA implementation are shown in Figure 57 below.

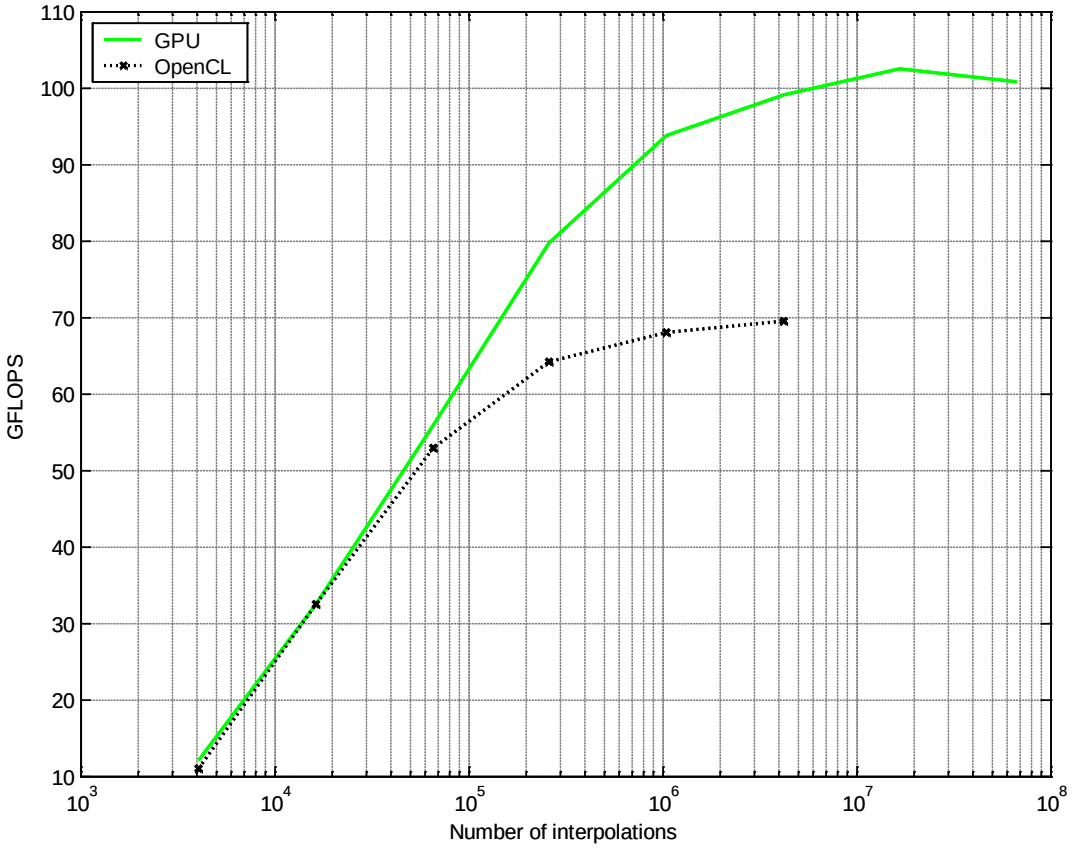

Figure 57 Performance for INT-C using OpenCL for various numbers of interpolations.

As can be seen in Figure 57, the OpenCL version's performance resembled that of the CUDA implementation only for smaller data sets. This is interesting considering that the kernels and hardware are basically identical. Considering this, it is unclear why the difference occurs in the first and also why the difference only occurs for larger problems, not smaller. The reason for this seems to be bandwidth related as detailed below.

Initially, both the OpenCL and CUDA versions behave similarly. The problem is of such small size that the greatest performance hamper is that the SMs are simply not fed with enough data because there is not enough of it. As off-chip bandwidth cannot be hidden in the first place due to the limited size, many bottle-necks related to bandwidth hiding would not be present. However, as the problem size increase, the performance between the two implantations starts to differ. This could be due to the fact that the OpenCL version for some reason cannot hide the off-chip latency equally good as the CUDA implementation. Also, it could be due to the OpenCL version coalescing less well. This would probably not show itself for smaller problems either as the bandwidth would not be the bottleneck then either. Both of these explanations to the problem could likely be due to the drivers, which have only recently been released in non-beta versions.

15.11 Synthetic Aperture Radar (SAR) inspired tilted matrix additions

Maximum kernel speedup over CPU: 83
Maximum GPU throughput achieved: 19[18] GB/s
Maximum CPU throughput achieved: 0.7 GB/s

Results:
The highest kernel speedup of was 83 times faster than the benchmark code run on the CPU. The highest speedup including all memory transfers was 2.7. All benchmarks tests were run for [2048, 2048] matrices.

Implementation:
The SAR benchmark is detailed in 14.11. In the GPU implementation, the data written from the host to the device had been transposed beforehand. The reason for this was that transposing the data meant that the direction of coalesced memory and the displacement of the addition are lined up in the same direction.

The speedup, throughput, and tilt dependence are shown in Figure 58, Figure 59, and Figure 60 below respectively.

[18] As explained below, this represents 75% of the theoretical bandwidth being used.

Prepared (also subject responsible if other)		No.		
SMW/DD/GX Jimmy Pettersson, Ian Wainwright		5/0363-FCP1041180 en		
Approved	Checked	Date	Rev	Reference
SMW/DD/GCX Lars Henricson	DD/LX	2010-01-27	A	

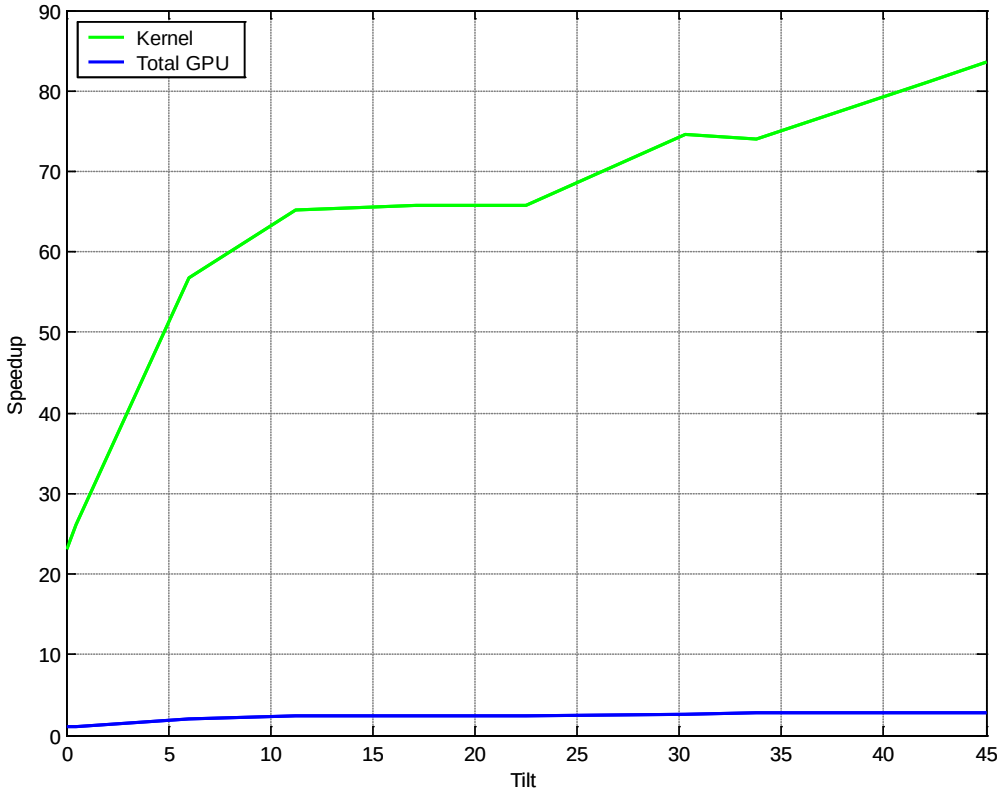

Figure 58 Kernel speedup for SAR for varying degrees of tilt.

As shown in Figure 58, a speedup of 83 was reached for the largest tilt. The increase in speedup for increased tilt is purely a factor of the CPU performing progressively worse due to cache misses. In fact, the GPU-implementation's throughput is tilt-independent, save for tilt = 0, as shown in Figure 59.

Prepared (also subject responsible if other)		No.		
SMW/DD/GX Jimmy Pettersson, Ian Wainwright		5/0363-FCP1041180 en		
Approved	Checked	Date	Rev	Reference
SMW/DD/GCX Lars Henricson	DD/LX	2010-01-27	A	

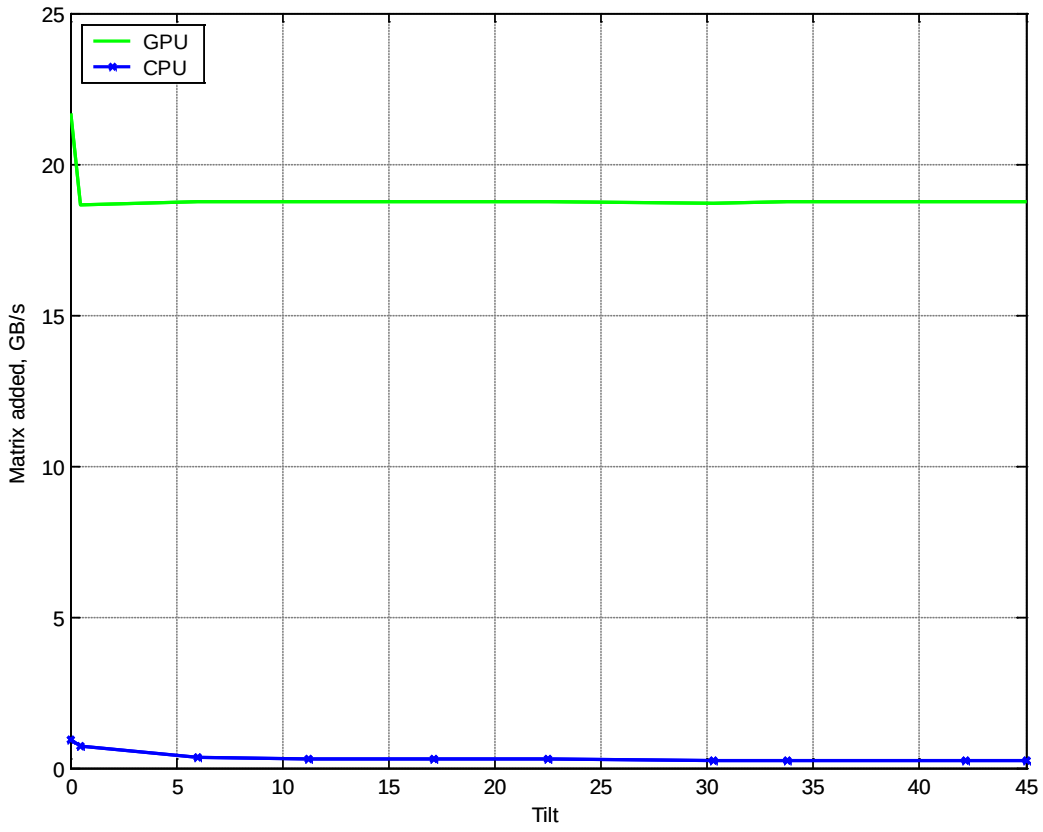

Figure 59 Throughput for SAR for varying degrees of tilt.

For all tilts save tilt = 0, the GPU has a constant throughput. This is because the C-matrix write is uncoalesced if the tilt is non-zero and has a constant efficiency loss of 50%, as described below. The CPU implementation, however, transfers less data progressively as the tilt increases due to the increased amount of cache misses, as can be seen in Figure 60

Worth noting is that the actual throughput according to the *CUDA profiler* for a tilt of 25 is:

- Global memory read throughput: 38 GB/s
- Global memory write throughput: 32 GB/s
- Global memory overall throughput: 70 GB/s

As the throughput of effective data, i.e. not counting wasted transfers due to uncoalesced memory access, is 53 GB/s but the actual memory transfers is 70 GB/s, the device wastes roughly 17 GB/s in uncoalesced transfers. This means that roughly 75% of the theoretical bandwidth is used.

The waste is due to the fact that the global writes of the C-matrix are shifted and, according to 7.3.1.1, which gives rise to uncoalesced memory access. In effect, this leads to a halving of the transfer efficiency of the C-write as the data has to be transferred twice; once for each of the two global memory segments the transfer writes to.

As the transfer efficiency for the write of C is halved, the three operations ‘read A’, ‘read B’ and ‘write C’, are now in practice ‘read A’, ‘read B’ and ‘write C twice’. Hence, the difference between the actual data transfer and the effective data transferred should be 4/3, or 1.33, which is indeed the case since 70 / 53 = 1.32.

The decline in throughput for various tilts compared to a tilt of 0 is shown in Figure 60.

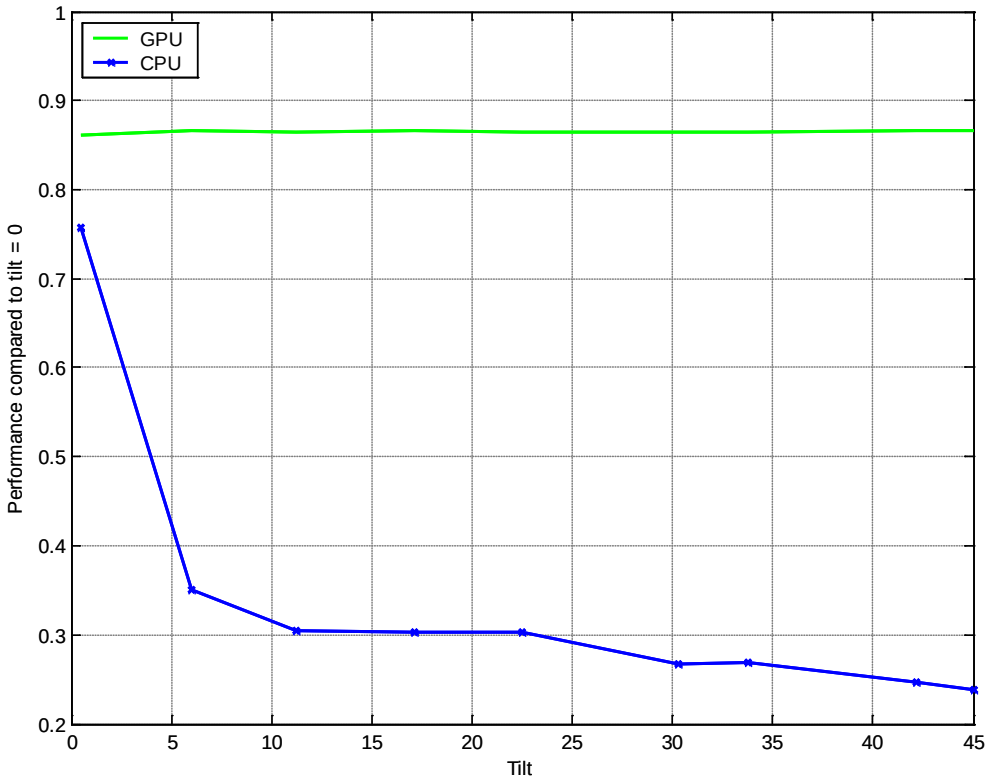

Figure 60 Performance decrease for SAR for various tilts.

The implemented version of the algorithm is very suitable for CUDA as it is highly parallel and has reasonably nice global memory access paters. Also, as can be seen in Figure 58, the implementation is suitable to use as a stand-alone kernel for 2048 * 2048 matrices as the total GPU speedup is at most 2.7.

The throughput was computed from dividing the amount of data in the result matrix by the time taken to perform the computation. The amount of data was 2048 * 2048 floats, i.e. 33.5 MB[19] of data. Note that the data used to calculate the throughput is only the data in the C-matrix, not the A- or B-matrices. The throughput is hence the amount of result matrix produced in GB per second, not the amount of data transferred.

15.12 Space Time Adaptive Processing: STAP

The data sets used in both the two STAP implementations are shown in Table 12 below.

[19] As noted in 14.16, a base of 10 instead of 1024 is used.

Data set	Doppler	Range	Range blocks	Beams	Order
MITRE	256	240	2	22	3
Extended	256	960	2	32	5

Table 12 The two data sets used to benchmark the STAP algorithm.

15.12.1 Using covariance matrix estimation

Maximum kernel speedup over CPU: 145
Maximum GPU performance achieved: 240 GFLOPS
Maximum CPU performance achieved: 1.76 GFLOPS

Results:
The highest kernel speedup was 145 times faster than the benchmark code run on the CPU. The highest speedup including all memory transfers was 89 times.

Implementation:
The STAP algorithm and is detailed in 14.12. On the GPU, two different programming philosophies were tested. The first one is presented in the CUDA programming guide by Nvidia in which they recommend the use of many lightweight threads. The second one is the alternative presented in 10.4.1 which instead recommends the use of heavier threads. Though the exact implementation differences are too complicated to be presented here, the main difference is that the first implementation utilizes shared memory, and the second uses the larger register memory. The latter implementation has a roughly three times higher performance than the former.

Performance and speedup for both implementations are shown in Figure 61 and Figure 62 respectively.

Prepared (also subject responsible if other)		No.		
SMW/DD/GX Jimmy Pettersson, Ian Wainwright		5/0363-FCP1041180 en		
Approved	Checked	Date	Rev	Reference
SMW/DD/GCX Lars Henricson	DD/LX	2010-01-27	A	

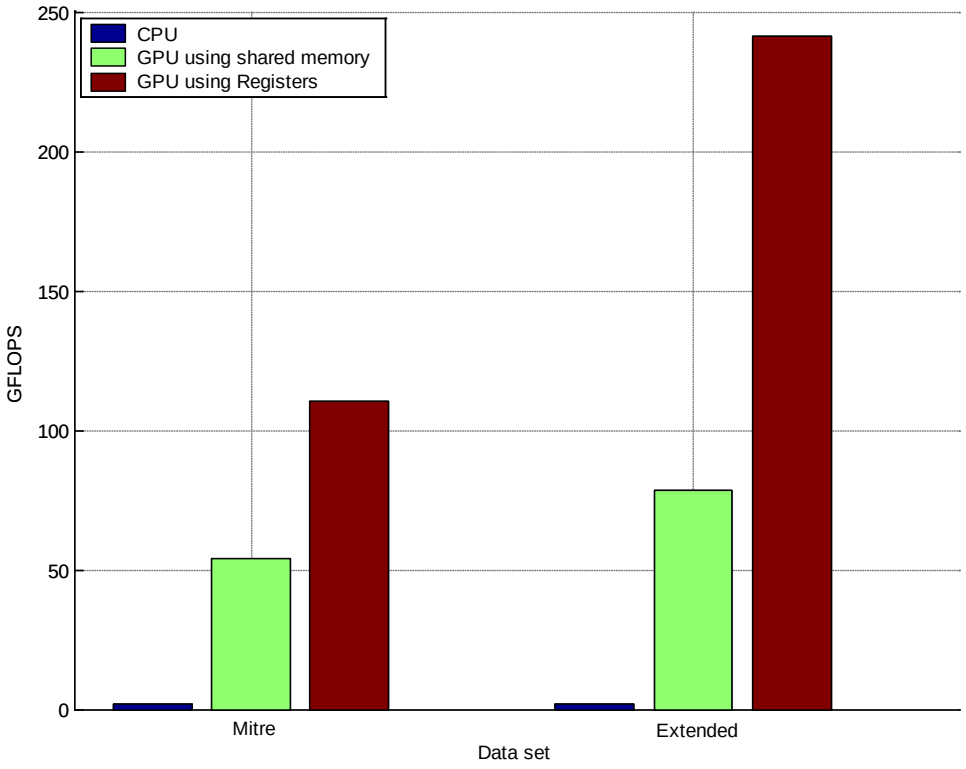

Figure 61 Performance for STAP with covariance matrix estimation for both the *Extended* and *Mitre* datasets.

Prepared (also subject responsible if other)		No.		
SMW/DD/GX Jimmy Pettersson, Ian Wainwright		5/0363-FCP1041180 en		
Approved	Checked	Date	Rev	Reference
SMW/DD/GCX Lars Henricson	DD/LX	2010-01-27	A	

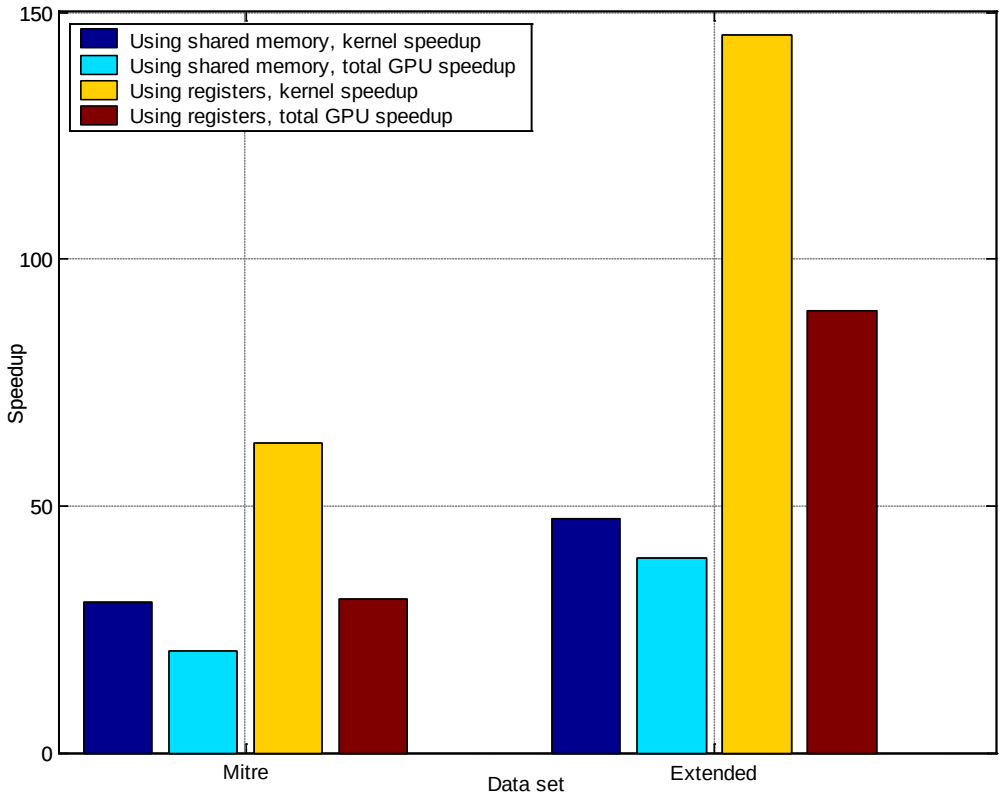

Figure 62 Speedup for STAP with covariance matrix estimation for dataset 1 and 2 with two different solution methods.

Clearly, the STAP algorithm suits the GPU very well. Higher AI and larger problem sizes increase the benefit further. Also, the deviation from Nvidia's programming philosophy in benefit for a more heavy-thread implementation is clearly beneficiary for the algorithm as the register implementation scores better than the shared memory one.

The two datasets used were named *MITRE* and *Extended* and are detailed in 14.12.

As can be seen in Figure 61 below the Extended dataset achieves 240 GFLOPS while the MITRE dataset achieves a more modest 110 GFLOPS. There are a number of reasons for this. One is that the Extended dataset is about 4 times bigger with an increased workload of about 25 which allows one to hide the on-chip latency better, another is that the Extended set requires 5 times more floating point operations per data read, and hence has a higher AI which makes hiding global memory access easier. A third reason is that the algorithm in its current form performs more effective data fetching for the extended dataset (due to better coalescing).

The performance was estimated as follows:

$$W_{covariance} = (N_{beams} * Order)^2 * 2$$
$$W_{total} = W_{covariance} * N_{doppler} * N_{range} * N_{range\_blocks}$$
$$Performance = \frac{W_{total}}{time}$$

15.12.2 Using QRDs

Maximum kernel speedup over CPU: 130
Maximum GPU performance achieved: 230 GFLOPS
Maximum CPU performance achieved: 1.76 GFLOPS

Results:
The highest kernel speedup of was 130 times faster than the benchmark code run on the CPU.

Implementation:
The STAP algorithm is detailed in 14.12. Though the implementation details are too many to present here, it is worth noting that the data sets used are just small enough to fit on-chip, if one utilizes both shared memory and the register memory to their limits. By doing this, the whole computation can be done on-chip. Also, as the STAP algorithm only needs the R part of the QRD, the QRD implemented here actually only computes R, not Q. The performance is shown in Figure 63.

Prepared (also subject responsible if other)		No.		
SMW/DD/GX Jimmy Pettersson, Ian Wainwright		5/0363-FCP1041180 en		
Approved	Checked	Date	Rev	Reference
SMW/DD/GCX Lars Henricson	DD/LX	2010-01-27	A	

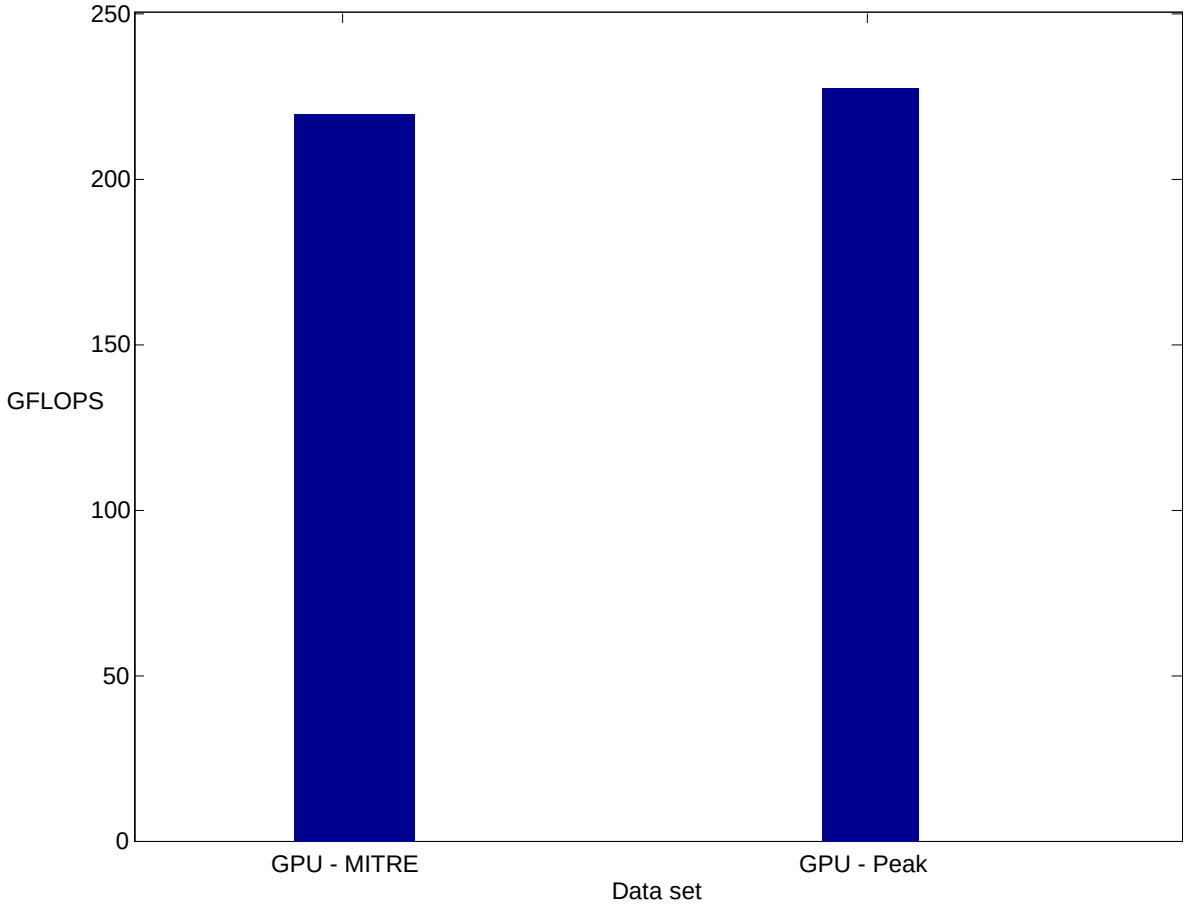

Figure 63 Performance for STAP using QRD for the two data sets.

Clearly, QRDs small enough to fit on-chip can suit the GPU very well. Note however that the implementation here does not compensate for poor condition number and similar, which the CULApack library can be expected to do.

The performance was calculated for doing the in-place transformation of A into R in A = QR. It is described by:

$$workload = 3 * N_{matrices} * N_{rows} * N_{cols} * \sum_{i=1}^{cols} (N_{rows} - i)$$

$$performance = \frac{workload}{time}$$

15.13 FFT

Maximum kernel speedup over CPU: 30
Maximum GPU performance achieved: 150 GFLOPS
Maximum CPU performance achieved: 8.1 GFLOPS

Results:
The highest kernel speedup of was 60 times faster than the benchmark code run on the CPU. The highest speedup including all memory transfers was 17.

Implementation:
The FFT algorithm is detailed in 14.13. On the GPU, the FFT benchmark was implemented by using Nvidia's CUFFT FFT library.

FFTs are only reasonably suitable for CUDA, given that they have a reasonably low complexity of only $5n\log(n)$. However, if a large number of them are to be computed, a good performance is easily achievable.

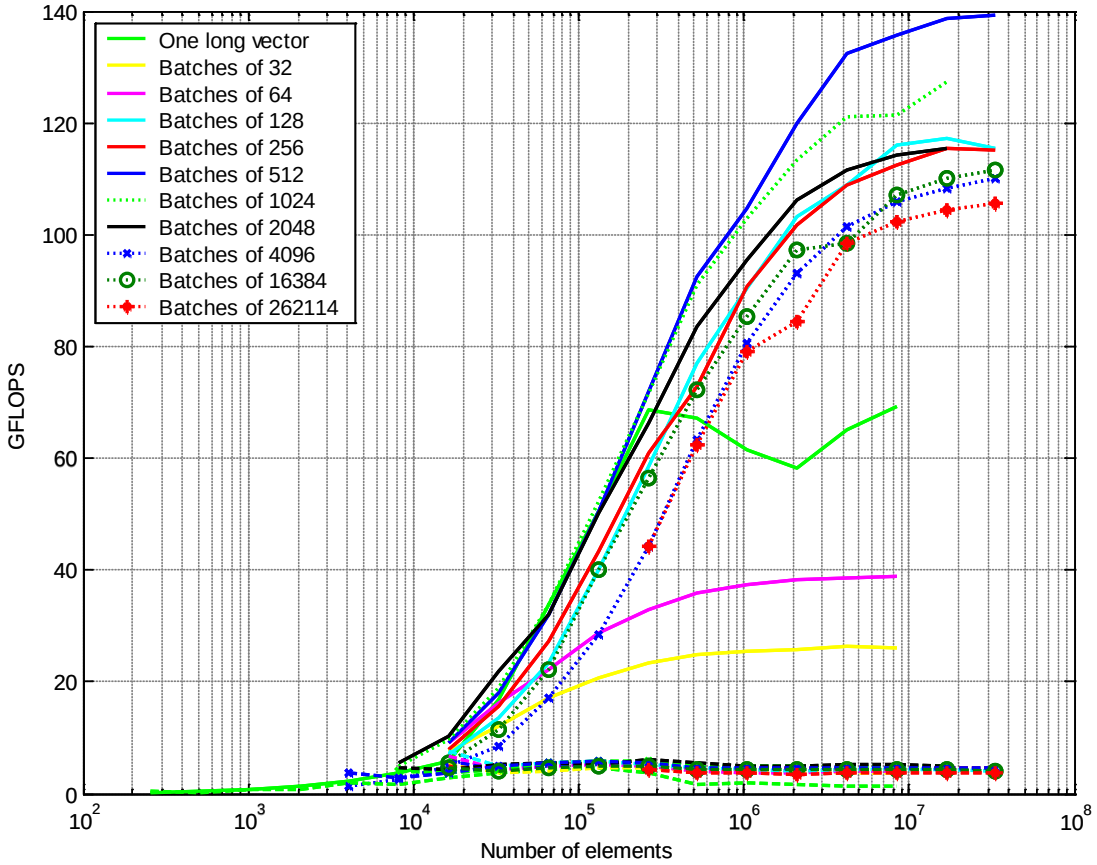

Figure 64 Performance for FFT for a given number of elements, either as one long vector or divided into batches.

From Figure 64, we see that the GPU benefits from larger problem sizes. Also, batches 32 of and 64 elements perform substantially worse compared to 128 elements and larger batches. This could also be due to the fact that the implementation might perform each batch as a block in the computational grid. If this is the case, then only 8 32-batch blocks can be active on an SM at any time as an SM can only handle 8 blocks simultaneously. As the AI of an FFT scales with its size, the AI of the smaller batches might simply be too low to be able to hide much of the on chip latency. Also, as an FFT is computed in a recursive fashion in which fewer and fewer elements are dealt with in each successive recursive step, eventually branching of a warp will occur. This branching will always occur at a certain size in the recursion irrespective of how large the batch is initially. However, for small batches, the branching part will be a relatively larger part of the whole computation, and hence give rise to a lower performance.

Furthermore, Figure 64 also shows us that both the GPU and the CPU implementation perform less well when computing one long input vector than the same number of elements divided into batches, despite the long input vector having a higher AI than the batched versions with the same number of elements. In the GPU case, this might be due to that all elements of the vector must communicate with each other in the sense that after the recursion has worked its way down the input vector, it must then work its way up through the input vector again. Hence, each element will in some manner influence the final value of all other elements. The FFT is in other words not embarrassingly data parallel and has a necessary small serial part. Hence, in the final steps of the recursion, the GPU will be idle to a large extent, in effect lowering the total performance as the most SMs will have to wait for the final, small serial computation to finish. The idle elements of the final recursion steps can, however, be hidden when using multiple batches as an SM can simply start computing for another batch instead of having to wait for an FFT that is in its final recursive steps.

In the CPU case, the slow-down for large long input vectors is likely due to the fact that the whole input vector does not fit on-chip as the dip in performance occurs at roughly 10^6 elements, or 8MB, where the 6 MB L2 cache is filled, and hence off-chip communication is necessary.

Prepared (also subject responsible if other)		No.		
SMW/DD/GX Jimmy Pettersson, Ian Wainwright		5/0363-FCP1041180 en		
Approved	Checked	Date	Rev	Reference
SMW/DD/GCX Lars Henricson	DD/LX	2010-01-27	A	

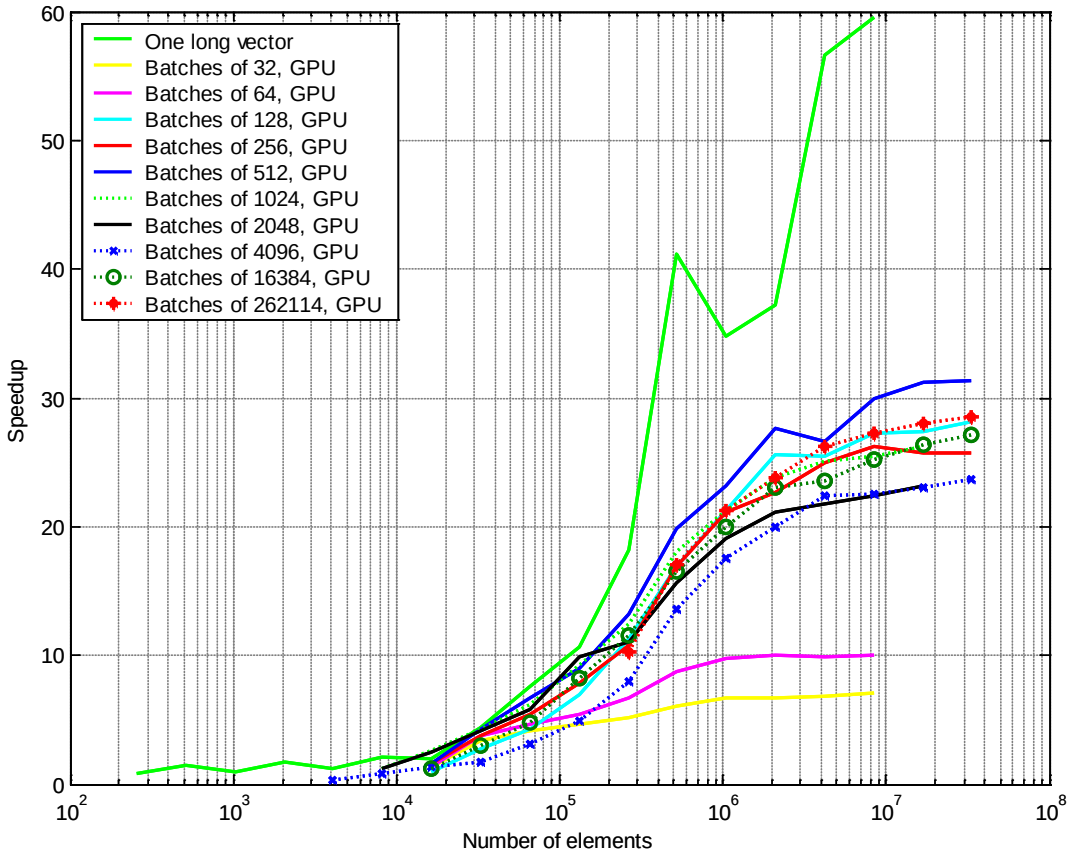

Figure 65 Kernel speedup for FFT against the FFTW algorithm.

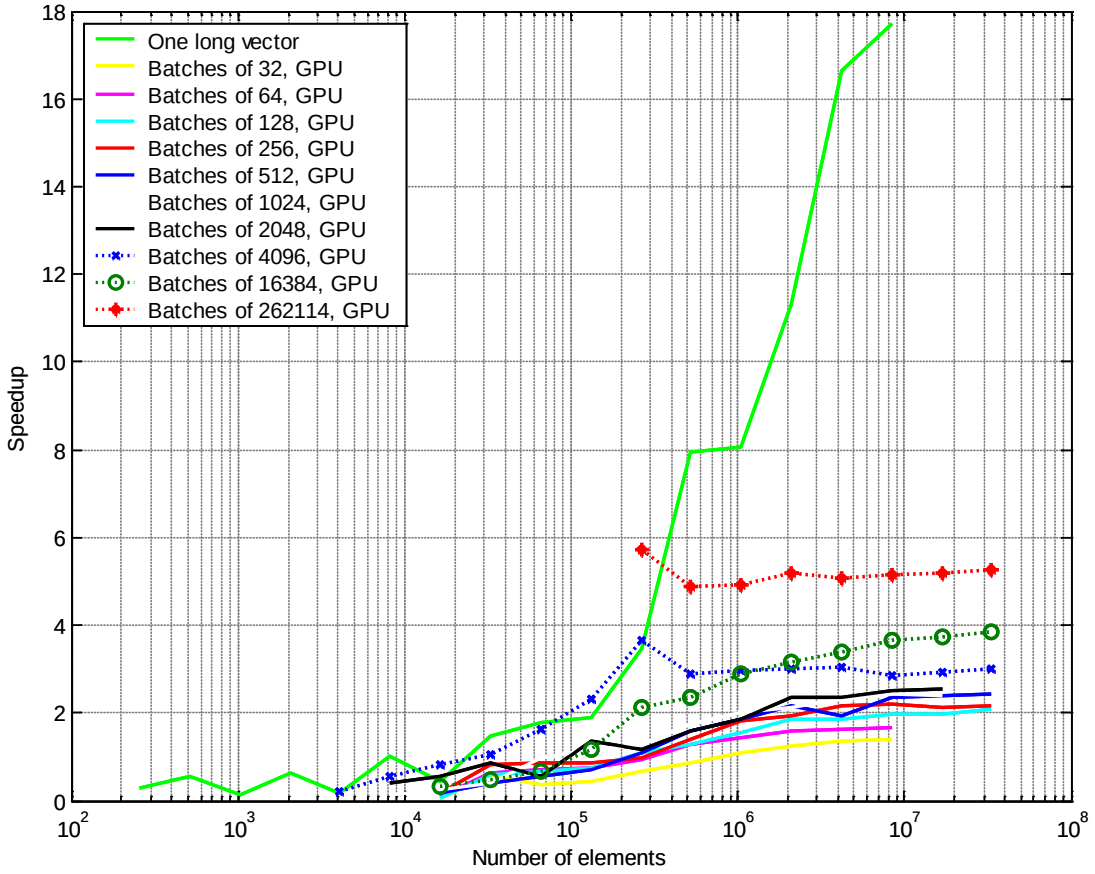

Figure 66 Total GPU speedup for FFT.

The kernel speed up shown in Figure 65 levels of at around 30 if we exclude the special unbatched case. However the total GPU speedup shown Figure 66 is not too significant, which means the FFT is best implemented in a chain of kernels rather than as a stand-alone one.

The performance was estimated by through applying the 5n log(n) * m as the number of operations, where *n* is the length of the vector and *m* is the number of input vectors, and simply dividing by the execution time.

15.14 Picture correlation

Maximum kernel speedup over CPU: 70
Maximum GPU performance achieved: 120 GFLOPS
Maximum CPU performance achieved: 2 GFLOPS

Results:
The highest kernel speedup of was 70 times faster than the benchmark code run on the CPU. The highest speedup including all memory transfers was 59.

Implementation:
Two implementations were tested for this benchmark; one utilizing shared memory, the other utilizing registers. Both implementations consisted of two kernels. The first kernel, the main one, computed the 625 correlation coefficients for small blocks in which the area of interest (AoI) was divided into. Each block then wrote its correlation vales to a previously allocated section in global memory, each block writing to a separate part. The second kernel, starting once all blocks of the previous kernel had ended as only one kernel can run at a time, and then simply looped over all the previous blocks' correlation coefficients. Despite its naïve and unoptimized implementation, the second kernel only occupied a fraction of the total time for both kernels, as the first kernel was very heavy.

The optimized implementation utilized many registers to maximize the amount of on-chip memory usage. The remaining bottleneck is still that there aren't very many floating point operations per memory fetch; this is by the nature of the benchmark.

Performance and speedup is shown below in Figure 67 and Figure 68 respectively.

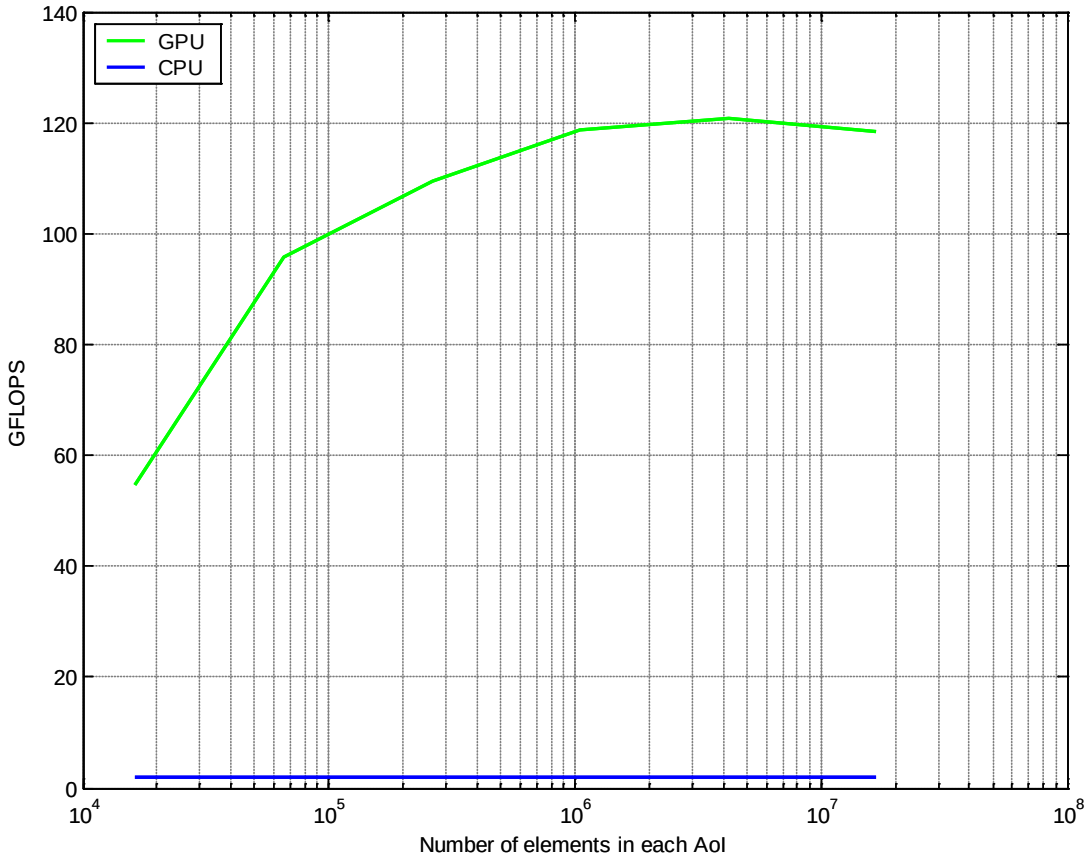

Figure 67 Performance for Picture Correlation for varying sizes of the Area of Interests (AoI).

Prepared (also subject responsible if other)		No.		
SMW/DD/GX Jimmy Pettersson, Ian Wainwright		5/0363-FCP1041180 en		
Approved	Checked	Date	Rev	Reference
SMW/DD/GCX Lars Henricson	DD/LX	2010-01-27	A	

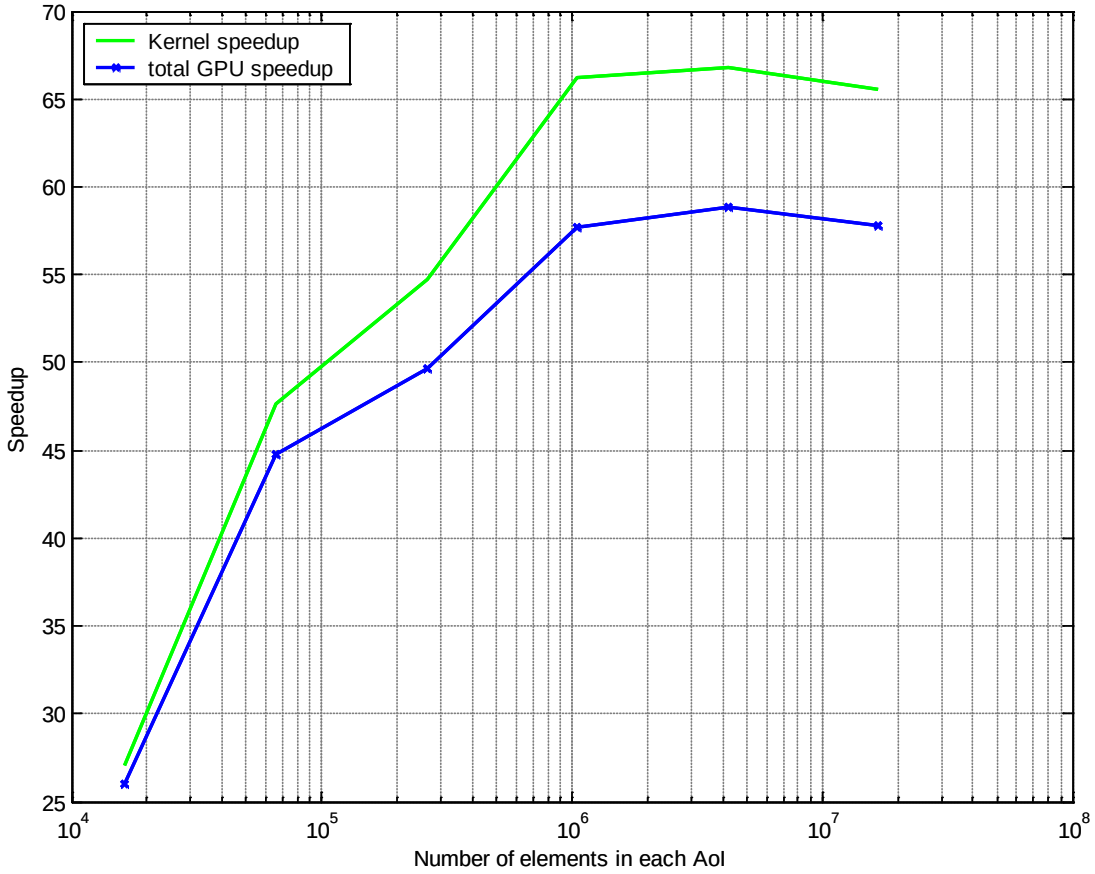

Figure 68 The kernel and total GPU speedup for Picture Correlation for varying sizes of the Area of Interests (AoI).

The performance was estimated by

$$workload = N_{complex-mult-add} * N_{rows} * N_{columns} * N_{configurations}$$

$$performance = \frac{workload}{time}$$

$$N_{complex-mult-add} \equiv 8$$

$$N_{configurations} \equiv 625$$

15.15 Benchmark conclusions and summary

The GPU clearly performs better for larger problems. Also, its performance relative to the CPU only increases with increased AI. Furthermore, the guidelines regarding the use of many lightweight threads should not always be followed. In fact, in all comparisons made, the use of heavy threads performed better than the use of light-weight ones. However, the actual coding for use of heavy threads is a lot more difficult than using the light-weight thread approach.

Worth noting is that many kernels use the same data, such as CFAR and STAP. Though CFAR on its own is unsuitable to run on the GPU due to the relatively high cost of transferring data between host and device, STAP, on the other hand, clearly performs a sufficient amount of calculations to motivate use of the GPU. Hence the CFAR has its data transferred to the GPU “for free” if the STAP algorithm is also used as they use the same data. Similarly to CFAR, the interpolation kernels and FFT also suffer from low total GPU speedup. However, they too are not used as stand-alone kernels in radar applications but rather in combination with other algorithms, and hence, their initial low feasibility for use on the GPU due to low total GPU speedup must be revaluated depending on the chain of kernels used.

Moreover, the radar signal processing algorithms tested here have either embarrassing internal parallelism, or parallelism can be found in using several input channels at once, several filters at once, or a combination. Hence, the radar signal processing benchmarks tested here have a data-parallel nature by default, fulfilling the basic criteria for GPU suitability.

A summary of all the implemented benchmarks including comments regarding feasibility is shown in Table 13.

Benchmark	Max GFLOPS achieved	Kernel Speedup for radar relevant sizes	Total GPU speedup for radar relevant sizes	Feasibility for radar applications
TDFIR	182	100	40	High
FDFIR	160	- [20]	- [20]	High
QRD using CULApack	190	- [21]	- [21]	Low, but improvements are possible as seen in STAP: QRD
SVD using CULApack	33	- [21]	- [21]	Low, but improvements might be possible.
CFAR	60	30	3	Medium. Should not be used as a standalone kernel due to low total GPU speedup
CT	28 GB/s[22]	50-100	1-2	High, as it is only used in a chain of kernels
Int-Bi-C	100	32	4	medium, as it is only used in a chain of kernels
Int-C	100	25	3	medium, as it is only used in a chain of kernels
SAR	19 GB/s[22]	80	5	High, as it is only used in a chain of kernels and its performance is independent of tilt.
STAP: Covariance matrix estimation	240	145	79	High
STAP: QRD	227	N/A	N/A	High, no code to compare against but the performance is very high.
FFT	140	30	18	High. Should not be used as a standalone kernel due to low total GPU speedup

[20] No CPU code was available for comparison.
[21] No radar relevant speedups because of small data sizes and where as the library used is intended for large matrices, hence the values are left empty.
[22] Both SAR and CT achieve roughly 75% of the theoretical bandwidth.

Picture Correlation	120	70	59	High

Table 13 Benchmark result overview.

16 Feasibility for radar signal processing

From the benchmark results, it is obvious that the CUDA GPU is substantially faster than the CPU and CPU code used, for radar relevant data sizes. For the most demanding benchmarks, a speedup of close to a factor of 100 is achieved. For performance oriented benchmarks, the peak is 250 GFLOPS, and for the two bandwidth oriented benchmarks, close to 75 % bandwidth of the theoretical bandwidth is achieved. Hence, from a pure performance perspective, GPUs are clearly feasible for radar signal processing.

Beside impressive performance, the GPUs energy efficiency (GFLOPS / W) is clearly better than for a regular CPU. Indeed Peter Cavil [32], General Manager at GE Fanuc Intelligence Platforms states SWaP; Size, Weight and Power, as one of the benefits of using CUDA GPUs. Due to the suitability of GPUs, GE Fanuc and Nvidia have recently entered an agreement to produce CUDA solutions for military and aerospace industry.

Development, also pointed out by Peter Cavil, is relatively simple. It does however necessitate a good understanding of the algorithm at hand, the CUDA hardware, and parallelizing of the algorithm, much like optimizing and parallelizing CPU implementations.

Despite the seemingly good energy efficiency compared to CPUs, GPUs can still produce a substantial amount of heat, making them potential hot spots for environments where cooling can be difficult. Flexibility in GPUs with varying performance and power does however exist as CUDA code performs equally optimized for all GPUs of the same architecture, as noted in 9.8, allowing for choice of smaller, more energy efficient GPUs if the there are strong power constraints.

Obsolescence can also be an issue. New architectures have appeared roughly every other year, which means purchasing old hardware might be difficult and leading to more than necessary upgrades and testing. However, as stated in 9.8.1, CUDA code will run safely on future hardware, hence rewriting code should not be necessary.

Furthermore, the fact that CUDA is a proprietary product owned solely by Nvidia can always present problems in the future it the company cancels development or if the company declares bankruptcy or similar, though this seems far from likely in the present climate.

17 Conclusions

17.1 Performance with the FX 4800 graphics board

From the benchmark results, it is obvious that the CUDA GPU is substantially faster than the CPU and CPU code used, for radar relevant data sizes. For the most demanding benchmarks, a speedup of close to a factor of 100 is achieved. For performance oriented benchmarks, the peak is 250 GFLOPS, and for the two bandwidth oriented benchmarks, close to 75 % bandwidth of the theoretical bandwidth is achieved.

Furthermore, good benchmark results are achieved for all but the QRD and SVD benchmarks where the CULApack library was used. However, considering the speedup of our STAP QRD, one can clearly put the result of the CULApack library into question, especially considering it is aimed at the HPC market for much larger matrices. A smaller on-chip SVD might also be possible to construct, similar to our own on-chip QRD. Hence, it is difficult to say if any of the implanted algorithms are unsuitable for CUDA when considering them for radar signal processing implementations.

17.2 Programming in CUDA and OpenCL

Developing for CUDA is reasonably straight forward. There is good support in the form of tools, documents, forums and white-papers, and what is needed from the developer is knowledge of C, the CUDA hardware, the algorithm at hand, and the means of parallelizing the algorithm in a data parallel manner.

Also, it is also possible to conclude that the CUDA programming guidelines should not be followed blindly. For instance, the desire for high arithmetic intensity should rather be specified as high local arithmetic intensity, and threads should not always be numerous and light but sometimes heavy at the cost of an increased number of threads.

OpenCL, on the other hand, is a lot more demanding when developing due to the immature state of developer tools, documentation, forums, and white-papers. However, as C for CUDA and OpenCL C are very similar, developing for OpenCL can be done by first writing a CUDA version, and then porting it to OpenCL.

17.3 Feasibility for radar signal processing

Beside the great performance and relative ease of development, there are other features that make CUDA a suitable choice for radar signal processing, and also some downsides.

First, the data-parallel nature of the investigated algorithms and radar signal processing in general means radar signal processing by default fulfils the primary criteria for use in GPUs, which means many algorithms will suit the GPU naturally.

Second, GPUs are more energy efficient than CPUs.

Downsides are the power density, potential hardware obsolesces problems, and the fact that CUDA is a proprietary standard in full control of one company, with all the associated risks involved.

18 Acknowledgements

We would especially like to thank our supervisors at Saab Electronic Defence Systems; Kurt Lind and Anders Åhlander for their vast input on radar signal processing algorithms, in-depth help regarding computer architectures and general support and assistance. We would also like to acknowledge our subject reviewer Sverker Holmgren, head of the cross-disciplinary research program in Computational Science at Uppsala University.

19 References

1 Multicore computing - the state of the art. Karl-Filip Faxén, Christer Bengtsson, Mats Brorsson, Hakan Grahn, Erik Hagersten Bengt Jonsson, Christoph Kessler, Björn Lisper, Per Stenström, Bertil Svensson, December 3, 2008

2 http://en.wikipedia.org/wiki/Comparison_of_AMD_graphics_processing_units#Radeon_Evergreen_.28HD_5xxx.29_series

3 John D Owens, David Luebke , Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Tim Purcell. A surevey of General-Purpose Computations on Graphics Hardware. Computer Graphics Forum, 26(1):80-113, March 2007.

4 http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543~114147,00.html

5 http://en.wikipedia.org/wiki/CUDA

6 http://www.khronos.org/news/press/releases/khronos_launches_heterogeneous_computing_initiative/

7 http://www.nvidia.com/object/win7_winvista_64bit_195.39_beta.html

8 http://www.nvidia.com/page/technologies.html

9 http://www.apple.com/macosx/technology/#opencl

10 http://www.tomshardware.com/news/intel-larrabee-gpgpu-gpu-cpu,7815.html

11 http://www.semiaccurate.com/2009/11/11/amd-uncovers-bulldozer-bobcat-and-fusion/

12 http://www.nvidia.com/object/io_1240981320317.html,

13 http://www.nvidia.com/object/tesla_computing_solutions.html

14 http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

15 http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242

16 CUDA Programming guide 2.3, page 82

17 CUDA Programming guide 2.3, page 89

18 http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242

19 http://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIA's_Fermi-The_First_Complete_GPU_Architecture.pdf

20 CUDA Best Practices Guide 2.3, 4.4 Thread and Block Heuristics

21 CUDA Best Practices Guide 2.3, page 41

22 CUDA Best Practices Guide 2.3, 4.4 Thread and Block Heuristics

23 Program Optimization Study on a 128-Core GPU. Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, and Wen-mei W. Hwu. Center for Reliable and High-Performance Computing. University of Illinois at Urbana-Champaign

24 CUDA Best Practices Guide 2.3, 5.1.1

25 CUDA Best Practices Guide 2.3, 1.1.2

26 http://www.cs.utk.edu/~dongarra/WEB-PAGES/cscads-libtune-09/talk21-volkov.pdf

27 OpenCL Specifications page 1.0.29, page 85

28 http://en.wikipedia.org/wiki/Adobe_Systems

Prepared (also subject responsible if other)		No.		
SMW/DD/GX Jimmy Pettersson, Ian Wainwright		5/0363-FCP1041180 en		
Approved	Checked	Date	Rev	Reference
SMW/DD/GCX Lars Henricson	DD/LX	2010-01-27	A	

29 http://www.brightsideofnews.com/news/2009/12/14/adobes-mercury-playback-engine-for-cs5-is-cuda-only!.aspx

30http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

31 PROGRAMMING METHODOLOGY FOR HIGH PERFORMANCE APPLICATIONS ON TILED ARCHITECTURES. Georgia Institute of Technology. DARPA Order No. V301/01, page 14

32 ELEKTRONIK I NORDEN, 14/5/2009