Master of Science Thesis in Information Coding
Department of Electrical Engineering, Linköping University, 2016

Comparison of Technologies for General-Purpose Computing on Graphics Processing Units

Torbjörn Sörman

Master of Science Thesis in Information Coding

Comparison of Technologies for General-Purpose Computing on Graphics Processing Units

Torbjörn Sörman

LiTH-ISY-EX–16/4923–SE

Supervisor: **Robert Forchheimer**
ISY, Linköpings universitet
Åsa Detterfelt
MindRoad AB

Examiner: **Ingemar Ragnemalm**
ISY, Linköpings universitet

Organisatorisk avdelning
Department of Electrical Engineering
Linköping University
SE-581 83 Linköping, Sweden

Abstract

The computational capacity of graphics cards for general-purpose computing have progressed fast over the last decade. A major reason is computational heavy computer games, where standard of performance and high quality graphics constantly rise. Another reason is better suitable technologies for programming the graphics cards. Combined, the product is high raw performance devices and means to access that performance. This thesis investigates some of the current technologies for general-purpose computing on graphics processing units. Technologies are primarily compared by means of benchmarking performance and secondarily by factors concerning programming and implementation. The choice of technology can have a large impact on performance. The benchmark application found the difference in execution time of the fastest technology, CUDA, compared to the slowest, OpenCL, to be twice a factor of two. The benchmark application also found out that the older technologies, OpenGL and DirectX, are competitive with CUDA and OpenCL in terms of resulting raw performance.

Acknowledgments

I would like to thank Åsa Detterfelt for the opportunity to make this thesis work at MindRoad AB. I would also like to thank Ingemar Ragnemalm at ISY.

Linköping, Februari 2016
Torbjörn Sörman

Contents

List of Tables

List of Figures

Acronyms

1D One-dimensional.

2D Two-dimensional.

3D Three-dimensional.

ACL AMD Compute Libraries.

API Application Programming Interface.

BLAS Basic Linear Algebra Subprograms.

CPU Central Processing Unit.

CUDA Compute Unified Device Architecture.

DCT Discrete Cosine Transform.

DFT Discrete Fourier Transform.

DIF Decimation-In-Frequency.

DWT Discrete Wavelet Transform.

EBCOT Embedded Block Coding with Optimized Truncation.

FFT Fast Fourier Transform.

FFTW Fastest Fourier Transform in the West.

FLOPS Floating-point Operations Per Second.

FPGA Field-Programmable Gate Array.

GCN Graphics Core Next.

GLSL OpenGL Shading Language.

GPGPU General-Purpose computing on Graphics Processing Units.

GPU Graphics Processing Unit.

HBM High Bandwidth Memory.

HBM2 High Bandwidth Memory.

HLSL High-Level Shading Language.

HPC High-Performance Computing.

ILP Instruction-Level Parallelism.

OpenCL Open Computing Language.

OpenGL Open Graphics Library.

OpenMP Open Multi-Processing.

OS Operating System.

PTX Parallel Thread Execution.

RDP Remote Desktop Protocol.

SM Streaming Multiprocessor.

VLIW Very Long Instruction Word-processor.

1 Introduction

This chapter gives an introduction to the thesis. The background, purpose and goal of the thesis, describes a list of abbreviations and the structure of this report.

1.1 Background

Technical improvements of hardware has for a long period of time been the best way to solve computationally demanding problems faster. However, during the last decades, the limit of what can be achieved by hardware improvements appear to have been reached: The operating frequency of the Central Processing Unit (CPU) does no longer significantly improve. Problems relying on single thread performance are limited by three primary technical factors:

1. The Instruction-Level Parallelism (ILP) wall
2. The memory wall
3. The power wall

The first wall states that it is hard to further exploit simultaneous CPU instructions: techniques such as instruction pipelining, superscalar execution and Very Long Instruction Word-processor (VLIW) exists but complexity and latency of hardware reduces the benefits.

The second wall, the gap between CPU speed and memory access time, that may cost several hundreds of CPU cycles if accessing primary memory.

The third wall is the power and heating problem. The power consumed is increased exponentially with each factorial increase of operating frequency.

Improvements can be found in exploiting parallelism. Either the problem itself is already inherently parallelizable, or reconstruct the problem. This trend manifests in development towards use and construction of multi-core microprocessors. The Graphics Processing Unit (GPU) is one such device, it originally exploited the inherent parallelism within visual rendering but now is available as a tool for massively parallelizable problems.

1.2 Problem statement

Programmers might experience a threshold and a slow learning curve to move from a sequential to a thread-parallel programming paradigm that is GPU programming. Obstacles involve learning about the hardware architecture, and restructure the application. Knowing the limitations and benefits might even provide reason to not utilize the GPU, and instead choose to work with a multi-core CPU.

Depending on ones preferences, needs, and future goals; selecting one technology over the other can be very crucial for productivity. Factors concerning productivity can be portability, hardware requirements, programmability, how well it integrates with other frameworks and Application Programming Interface (API)s, or how well it is supported by the provider and developer community. Within the range of this thesis, the covered technologies are Compute Unified Device Architecture (CUDA), Open Computing Language (OpenCL), DirectCompute (API within DirectX), and Open Graphics Library (OpenGL) Compute Shaders.

1.3 Purpose and goal of the thesis work

The purpose of this thesis is to evaluate, select, and implement an application suitable for General-Purpose computing on Graphics Processing Units (GPGPU).

To implement the same application in technologies for GPGPU: (CUDA, OpenCL, DirectCompute, and OpenGL), compare GPU results with results from an sequential C/C++ implementation and a multi-core OpenMP implementation, and to compare the different technologies by means of benchmarking, and the goal is to make qualitative assessments of how it is to use the technologies.

1.4 Delimitations

Only one benchmark application algorithm will be selected, the scope and time required only allows for one algorithm to be tested. Each technology have different debugging and profiling tools and those are not included in the comparison of the technologies. However important such tool can be, they are of a subjective nature and more difficult to put a measure on.

2

Benchmark algorithm

This part cover a possible applications for a GPGPU study. The basic theory and motivation why they are suitable for benchmarking GPGPU technologies is presented.

2.1 Discrete Fourier Transform

The Fourier transform is of use when analysing the spectrum of a continuous analogue signal. When applying transformation to a signal it is decomposed into the frequencies that makes it up. In digital signal analysis the Discrete Fourier Transform (DFT) is the counterpart of the Fourier transform for analogue signals. The DFT converts a sequence of finite length into a list of coefficients of a finite combination of complex sinusoids. Given that the sequence is a sampled function from the time or spatial domain it is a conversion to the frequency domain. It is defined as

$$X_k = \sum_{n=0}^{N-1} x(n) W_N^{kn}, k \in [0, N-1] \tag{2.1}$$

where $W_N = e^{-\frac{i2\pi}{N}}$, commonly named the twiddle factor [15].

The DFT is used in many practical applications to perform Fourier analysis. It is a powerful mathematical tool that enables a perspective from another domain where difficult and complex problems becomes easier to analyse. Practically used in digital signal processing such as discrete samples of sound waves, radio signals or any continuous signal over a finite time interval. If used in image processing, the sampled sequence is pixels along a row or column. The DFT takes input in complex numbers, and gives output in complex coefficients. In practical applica-

tions the input is usually real numbers.

2.1.1 Fast Fourier Transform

The problem with the DFT is that the direct computation require $\mathcal{O}(n^n)$ complex multiplications and complex additions, which makes it computationally heavy and impractical in high throughput applications. The Fast Fourier Transform (FFT) is one of the most common algorithms used to compute the DFT of a sequence. A FFT computes the transformation by factorizing the transformation matrix of the DFT into a product of mostly zero factors. This reduces the order of computations to $\mathcal{O}(n \log n)$ complex multiplications and additions.

The FFT was made popular in 1965 [7] by J.W Cooley and John Tukey. It found it is way into practical use at the same time, and meant a serious breakthrough in digital signal processing [8, 5]. However, the complete algorithm was not invented at the time. The history of the Cooley-Tukey FFT algorithm can be traced back to around 1805 by work of the famous mathematician Carl Friedrich Gauss[18]. The algorithm is a divide-and-conquer algorithm that relies on recursively dividing the input into sub-blocks. Eventually the problem is small enough to be solved, and the sub-blocks are combined into the final result.

2.2 Image processing

Image processing consists of a wide range of domains. Earlier academic work with performance evaluation on the GPU [25] tested four major domains and compared them with the CPU. The domains were Three-dimensional (3D) shape reconstruction, feature extraction, image compression, and computational photography. Image processing is typically favourable on a GPU since images are inherently a parallel structure.

Most image processing algorithms apply the same computation on a number of pixels, and that typically is a data-parallel operation. Some algorithms can then be expected to have huge speed-up compared to an efficient CPU implementation. A representative task is applying a simple image filter that gathers neighbouring pixel-values and compute a new value for a pixel. If done with respect to the underlying structure of the system, one can expect a speed-up near linear to the number of computational cores used. That is, a CPU with four cores can theoretically expect a near four time speed-up compared to a single core. This extends to a GPU so that a GPU with n cores can expect a speed-up in the order of n in ideal cases. An example of this is a Gaussian blur (or smoothing) filter.

2.3 Image compression

The image compression standard *JPEG2000* offers algorithms with parallelism but is very computationally and memory intensive. The standard aims to improve performance over JPEG, but also to add new features. The following sec-

tions are part of the JPEG2000 algorithm [6]:

1. Color Component transformation
2. Tiling
3. Wavelet transform
4. Quantization
5. Coding

The computation heavy parts can be identified as the Discrete Wavelet Transform (DWT) and the encoding engine uses Embedded Block Coding with Optimized Truncation (EBCOT) Tier-1.

One difference between the older format *JPEG* and the newer JPEG2000 is the use of DWT instead of Discrete Cosine Transform (DCT). In comparison to the DFT, the DCT operates solely on real values. DWTs, on the other hand, uses another representation that allows for a time complexity of $\mathcal{O}(N)$.

2.4 Linear algebra

Linear algebra is central to both pure and applied mathematics. In scientific computing it is a highly relevant problem to solve dense linear systems efficiently. In the initial uses of GPUs in scientific computing, the graphics pipeline was successfully used for linear algebra through programmable vertex and pixel shaders [20]. Methods and systems used later on for utilizing GPUs have been shown efficient also in hybrid systems (multi-core CPUs + GPUs) [27]. Linear algebra is highly suitable for GPUs and with careful calibration it is possible to reach 80%-90% of the theoretical peak speed of large matrices [28].

Common operations are vector addition, scalar multiplication, dot products, linear combinations, and matrix multiplication. Matrix multiplications have a high time complexity, $\mathcal{O}(N^3)$, which makes it a bottleneck in many algorithms. Matrix decomposition like LU, QR, and Cholesky decomposition are used very often and are subject for benchmark applications targeting GPUs [28].

2.5 Sorting

The sort operation is an important part of computer science and is a classic problem to work on. There exists several sorting techniques, and depending on problem and requirements a suitable algorithm is found by examining the attributes.

Sorting algorithms can be organized into two categories, data-driven and data-independent. The quicksort algorithm is a well known example of a data-driven sorting algorithm. It performs with time complexity $\mathcal{O}(n \log n)$ on average, but have a time complexity of $\mathcal{O}(n^2)$ in the worst case. Another data-driven algorithm that does not have this problem is the heap sort algorithm, but it suffers from

difficult data access patterns instead. Data-driven algorithms are not the easiest to parallelize since their behaviour is unknown and may cause bad load balancing among computational units.

The data independent algorithms is the algorithms that always perform the same process no matter what the data. This behaviour makes them suitable for implementation on multiple processors, and fixed sequences of instructions, where the moment in which data is synchronized and communication must occur are known in advance.

2.5.1 Efficient sorting

Bitonic sort have been used early on in the utilization of GPUs for sorting. Even though it has the time complexity of $\mathcal{O}(n \log n^2)$ it has been an easy way of doing a reasonably efficient sort on GPUs. Other high-performance sorting on GPUs are often combinations of algorithms. Some examples of combined sort methods on GPUs are the bitonic merge sort, and a bucket sort that split the sequence into smaller sequences before each being sorted with a merge sort.

A popular algorithm for GPUs have been variants of radix sort which is a non-comparative integer sorting algorithm. Radix sorts can be described as being easy to implement and still as efficient as more sophisticated algorithms. Radix sort works by grouping the integer keys by the individual digit value in the same significant position and value.

2.6 Criteria for Algorithm Selection

A benchmarking application is sought that have the necessary complexity and relevance for both practical uses and the scientific community. The algorithm with enough complexity and challenges is the FFT. Compared to the other presented algorithms the FFT is more complex than the matrix operations and the regular sorting algorithms. The FFT does not demand as much domain knowledge as the image compression algorithms, but it is still a very important algorithm for many applications.

The difficulties of working with multi-core systems are applied to GPUs. What GPUs are missing compared to multi-core CPUs, is the power of working in sequential. Instead, GPUs are excellent at fast context switching and hiding memory latencies. Most effort of working with GPUs extends to supply tasks with enough parallelism, avoiding branching, and optimize memory access patterns. One important issue is also the host to device memory transfer-time. If the algorithm is much faster on the GPU, a CPU could still be faster if the host to device and back transfer is a large part of the total time.

By selecting an algorithm that have much scientific interest and history relevant comparisons can be made. It is sufficient to say that one can demand a reasonable performance by utilizing information sources showing implementations on GPUs.

3
Theory

This chapter will give an introduction to the FFT algorithm and a brief introduction of the GPU.

3.1 Graphics Processing Unit

A GPU is traditionally specialized hardware for efficient manipulation of computer graphics and image processing [24]. The inherent parallel structure of images and graphics makes them very efficient at some general problems where parallelism can be exploited. The concept of GPGPU is solving a problem on the GPU platform instead of a multi-core CPU system.

3.1.1 GPGPU

In the early days of GPGPU one had to know a lot about computer graphics to compute general data. The available APIs were created for graphics processing. The dominant APIs were OpenGL and DirectX. High-Level Shading Language (HLSL) and OpenGL Shading Language (GLSL) made the step easier, but it still generated code into the APIs.

A big change was when NVIDIA released CUDA, which together with new hardware made it possible to use standard C-code to program the GPU (with a few extensions). Parallel software and hardware was a small market at the time, and the simplified use of the GPU for parallel tasks opened up to many new customers. However, the main business is still graphics and the manufacturers can not make cards too expensive, especially at the cost of graphics performance (as would increase of more double-precision floating-point capacity). This can be exemplified with a comparison between NVIDIA's Maxwell micro architecture, and the prede-

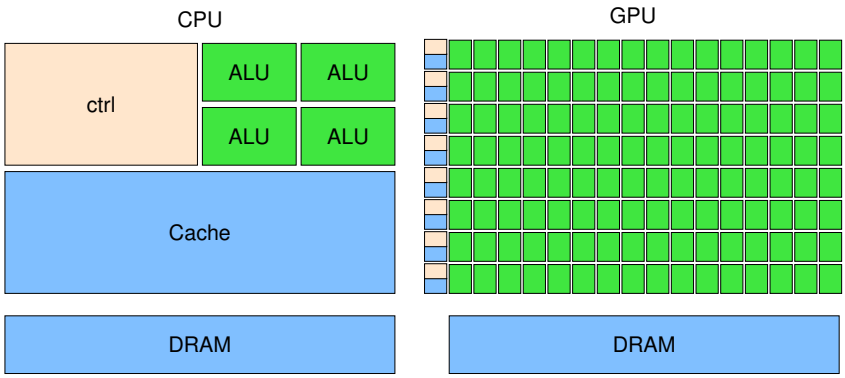


Figure 3.1: *The GPU uses more transistors for data processing*

cessor Kepler: Both are similar, but with Maxwell some of the double-precision floating-point capacity was removed in favour of single-precision floating-point value capacity (preferred in graphics).

Using GPUs in the context of data centers and High-Performance Computing (HPC), studies show that GPU acceleration can reduce power [19] and it is relevant to know the behaviour of the GPUs in the context of power and HPC [16] for the best utilization.

3.1.2 GPU vs CPU

The GPU is built on a principle of more execution units instead of higher clock-frequency to improve performance. Comparing the CPU with the GPU, the GPU performs a much higher theoretical Floating-point Operations Per Second (FLOPS) at a better cost and energy efficiency [23]. The GPU relies on using high memory bandwidth and fast context switching (execute the next warp of threads) to compensate for lower frequency and hide memory latencies. The CPU is excellent at sequential tasks with features like branch prediction.

The GPU thread is lightweight and its creation has little overhead, whereas on the CPU the thread can be seen as an abstraction of the processor, and switching a thread is considered expensive since the context has to be loaded each time. On the other hand, a GPU is very inefficient if not enough threads are ready to perform work. Memory latencies are supposed to be hidden by switching in a new set of working threads as fast as possible.

A CPU thread have its own registers whereas the GPU thread work in groups where threads share registers and memory. One can not give individual instructions to each thread, all of them will execute the same instruction. The figure 3.1 demonstrates this by showing that by sharing control-structure and cache, the GPU puts more resources on processing than the CPU where more resources goes into control structures and memory cache.

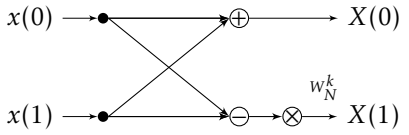


Figure 3.2: *Radix-2 butterfly operations*

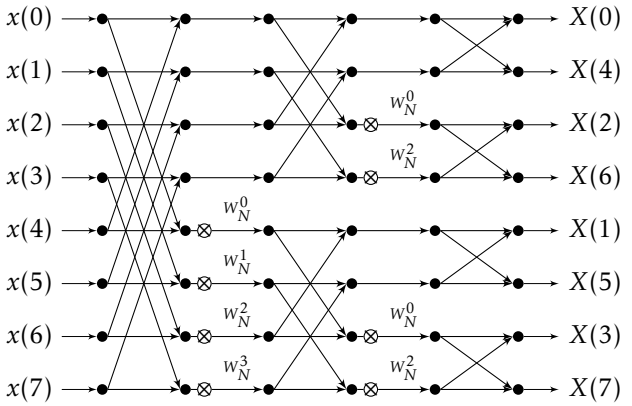


Figure 3.3: *8-point radix-2 FFT using Cooley-Tukey algorithm*

3.2 Fast Fourier Transform

This section extends the information from section 2.1.1 in the *Benchmark application* chapter.

3.2.1 Cooley-Tukey

The Fast Fourier Transform is by far mostly associated with the Cooley-Tukey algorithm [7]. The Cooley-Tukey algorithm is a devide-and-conquer algorithm that recursively breaks down a DFT of any composite size of $N = N_1 \cdot N_2$. The algorithm decomposes the DFT into $s = \log_r N$ stages. The N-point DFT is composed of r-point small DFTs in s stages. In this context the r-point DFT is called radix-r butterfly.

Butterfly and radix-2

The implementation of an N-point radix-2 FFT algorithm have $\log_2 N$ stages with $N/2$ butterfly operations per stage. A butterfly operation is an addition, and a subtraction, followed by a multiplication by a twiddle factor, see figure 3.2.

Figure 3.3 shows an 8-point radix-2 Decimation-In-Frequency (DIF) FFT. The input data are in natural order whereas the output data are in bit-reversed order.

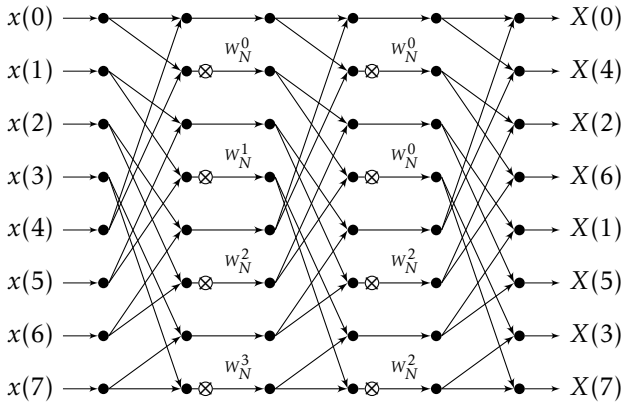


Figure 3.4: *Flow graph of an radix-2 FFT using the constant geometry algorithm.*

3.2.2 Constant geometry

Constant geometry is similar to Cooley-Tukey, but with another data access pattern that uses the same indexing in all stages. Constant geometry removes the overhead of calculating the data input index at each stage, as seen in figure 3.3 where the top butterfly in the first stage require input $x[0], x[4]$, and in the second stage $x[0], x[2]$, whilst the constant geometry algorithm in figure 3.4 uses $x[0], x[4]$ as input for all stages.

3.2.3 Parallelism in FFT

By examining the FFT algorithm, parallelism can be exploited in several ways. Naturally, when decomposing the DFT into radix-2 operations, parallelism can be achieved by mapping one thread per data input. That would, however, lead to an unbalanced load as every second input is multiplied by the complex twiddle factor, whereas the other half has no such step. The solution is to select one thread per radix-2 butterfly operation, each thread will then share the same workload.

3.2.4 GPU algorithm

The complete FFT application can be implemented in two different kernels: One kernel executing over a single stage, and another kernel executing the last stages that could fit within one block. The single-stage kernel, called the *global kernel*, would execute each stage of the algorithm in sequential order. Each execution would require in total as many threads as there are butterfly-operations. The host would supply the kernel with arguments depending on stage number and problem size. (See table 3.1 for full parameter list.) The global kernel algorithm is shown in algorithm 1. The global kernel would only be called for the number of stages not fitted in a single block (this depends on the number of selected threads per block). The global kernel implements Cooley-Tukey algorithm.

Parameter	Argument
data	Input/Output data buffer
stage	$[0, \log_2(N) - \log_2(N_{block})]$
bitmask	LEFTSHIFT($\texttt{FFFFFFFF}_{16}, 32 - stage$)
angle	$(2 \cdot \pi)/N$
dist	RIGHTSHIFT($N, steps$)

Table 3.1: *Global kernel parameter list with argument depending on size of input N and $stage$.*

Algorithm 1 Pseudo-code for the global kernel with input from the host.

```
1:  procedure GLOBALKERNEL(data, stage, bitmask, angle, dist)
2:      tid ← GLOBALTHREADID
3:      low ← tid + (tid & bitmask)
4:      high ← low + dist
5:      twMask ← SHIFTLEFT(dist − 1, stage)
6:      twStage ← POWEROFTWO(stage) · tid
7:      a ← angle · (twStage & twMask)
8:      IMAG(twiddleFactor) ← SIN(a)
9:      REAL(twiddleFactor) ← COS(a)
10:     temp ← COMPLEXSUB(data_low, data_high)
11:     data_low ← COMPLEXADD(data_low, data_high)
12:     data_high ← COMPLEXMUL(temp, twiddleFactor)
13: end procedure
```

Parameter	Argument
in	Input data buffer
out	Output data buffer
angle	$(2 \cdot \pi)/N$
stages	$[\log_2(N) - \log_2(N_{block}), \log_2(N)]$
leadingBits	$32 - \log_2(N)$
c	Forward: -1, Inverse: $1/N$

Table 3.2: *Local kernel parameter list with argument depending on size of input N and number of stages left to complete.*

Shared/Local memory

The *local kernel* is always called, and encapsulates all remaining stages and the bit-reverse order output procedure. It is devised as to utilize shared memory completely for all stages. This reduces the primary memory accesses to a single read and write per data point. The kernel implements the constant geometry algorithm to increase performance in the inner loop: the input and output index is only calculated once. See algorithm 2.

Register width

The targeted GPUs work on 32 bit registers and all fast integer arithmetic is based on that. Procedures using bitwise operations are constructed with this architectural specific information, as seen in the *bitmask* parameter in table 3.1 and the *leadingBits* parameter in table 3.2. The bitmask parameter is used to get the offset for each stage using the Cooley-Tukey algorithm. The leadingBits parameter is used in the bit-reverse operation to remove the leading zeroes that comes as a result of the use of a 32 bit register.

Bit-reverse example: If the total size is 1024 elements, the last $\log_2(1024) = 10$ bits are used. When encountering $1008 = 1111110000_2$ for bit-reversal in this context (with a problem size of 1024 points) the result is 63. However, using a 32 bit register:

$$1008 = 00000000000000000000001111110000_2 \tag{3.1}$$

bits reversed:

$$264241152 = 00001111110000000000000000000000_2 \tag{3.2}$$

The leading zeroes becomes trailing zeroes that needs to be removed. A logic right shift operation by the length of leadingBits $= 32 - \log_2(1024) = 22$ solves this.

3.3 Related research

Scientific interest have mainly been targeting CUDA and OpenCL for comparisons. Benchmarking between the two have established that there is difference

Algorithm 2 Pseudo-code for the local kernel with input from the host.

```
1:  procedure LocalKernel(in, out, angle, stages, leadingBits, c)
2:      let shared be a shared/local memory buffer
3:      low ← ThreadId
4:      high ← low + BlockDim
5:      offset ← BlockId · BlockDim · 2
6:      shared_low ← in_{low+offset}
7:      shared_high ← in_{high+offset}
8:      ConstantGeometry(shared, low, high, angle, stages)
9:      revLow ← BitReverse(low + offset, leadingBits)
10:     revHigh ← BitReverse(high + offset, leadingBits)
11:     out_{revLow} ← ComplexMul(c, shared_low)
12:     out_{revHigh} ← ComplexMul(c, shared_high)
13: end procedure

14: procedure ConstantGeometry(shared, low, high, angle, stages)
15:     out_i ← low · 2
16:     out_ii ← outI + 1
17:     for stage ← 0, stages − 1 do
18:         bitmask ← ShiftLeft(0xFFFFFFFF, stage)
19:         a ← angle · (low & bitmask)
20:         Imag(twiddleFactor) ← Sin(a)
21:         Real(twiddleFactor) ← Cos(a)
22:         temp ← ComplexSub(shared_low, shared_high)
23:         shared_{out_i} ← ComplexAdd(shared_low, shared_high)
24:         shared_{out_ii} ← ComplexMul(twiddleFactor, temp)
25:     end for
26: end procedure
```

in favour of CUDA, however it can be due to unfair comparisons [10], and with the correct tuning OpenCL can be just as fast. The same paper stated that the biggest difference came from running the forward FFT algorithm. Examination showed that large differences could be found in the Parallel Thread Execution (PTX) instructions (intermediary GPU code).

Porting from CUDA to OpenCL without loosing performance have been explored in [9], where the goal was to achieve a performance-portable solution. Some of the main differences between the technologies are described in that paper.

4

Technologies

Five different multi-core technologies are used in this study. One is a proprietary parallel computing platform and API, called CUDA. *Compute Shaders* in OpenGL and DirectCompute are parts of graphic programming languages but have a fairly general structure and allows for general computing. OpenCL have a stated goal to target any heterogeneous multi-core system but is used in this study solely on the GPU. To compare with the CPU, OpenMP is included as an effective way to parallelize sequential C/C++-code.

4.1 CUDA

CUDA is developed by NVIDIA and was released in 2006. CUDA is an extension of the C/C++ language and have its own compiler. CUDA supports the functionality to execute kernels, and modify the graphic card RAM memory and the use of several optimized function libraries such as *cuBLAS* (CUDA implementation of Basic Linear Algebra Subprograms (BLAS)) and *cuFFT* (CUDA implementation of FFT).

A program launched on the GPU is called a kernel. The GPU is referred to as the *device* and the the CPU is called the *host*. To run a CUDA kernel, all that is needed is to declare the program with the function type specifier `__global__` and call it from the host with launch arguments, for other specifiers see table 4.1. The kernel execution call includes specifying the thread organization. Threads are organized in blocks, that in turn are specified within a *grid*. Both the block and grid can be used as One-dimensional (1D), Two-dimensional (2D) or Three-dimensional (3D) to help the addressing in a program. These can be accessed within a kernel by the structures `blockDim` and `gridDim`. Thread and block

Function type	Executed on	Callable from
`__device__`	Device	Device
`__global__`	Device	Host
`__host__`	Host	Host

Table 4.1: *Table of function types in CUDA.*

```
__global__ void cu_global(
    cpx *in,
    unsigned int mask,
    float angle,
    int steps,
    int dist)
{
    cpx w;
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    // Input offset
    in += tid + (tid & mask);
    cpx *h = in + dist;

    // Twiddle factor
    angle *= ((tid << steps) & ((dist - 1) << steps));
    __sincosf(angle, &w.y, &w.x);

    // Butterfly
    float x = in->x - h->x;
    float y = in->y - h->y;
    *in = { in->x + h->x, in->y + h->y };
    *h  = {(w.x * x)-(w.y * y), (w.y * x)+(w.x * y)};
}
```

Figure 4.1: *CUDA global kernel*

identification is done with `threadIdx` and `blockIdx`.

All limitations can be polled from the device and all devices have a minimum feature support called *Compute capability*. The compute capability aimed at in this thesis is 3.0 and includes the NVIDIA GPU models starting with *GK* or later models (*Tegra* and *GM*).

CUDA exposes intrinsic integer functions on the device and a variety of fast math functions, optimized single-precision operations, denoted with the suffix *-f*. In the CUDA example in figure 4.1 the trigonometric function `__sincosf` is used to calculate both $\sin\alpha$ and $\cos\alpha$ in a single call.

4.2 OpenCL

OpenCL is a framework and an open standard for writing programs that executes on multi-core platforms such as the CPU, GPU and Field-Programmable Gate Array (FPGA) among other processors and hardware accelerators. OpenCL uses a

```
__kernel void ocl_global (
    __global cpx *in,
    float angle,
    unsigned int mask,
    int steps,
    int dist )
{
    cpx w;
    int tid = get_global_id(0);

    // Input offset
    in += tid + (tid & mask);
    cpx *high = in + dist;

    // Twiddle factor
    angle *= ((tid << steps) & ((dist - 1) << steps));
    w.y = sincos(angle, &w.x);

    // Butterfly
    float x = in->x - high->x;
    float y = in->y - high->y;
    in->x   = in->x + high->x;
    in->y   = in->y + high->y;
    high->x = (w.x * x) - (w.y * y);
    high->y = (w.y * x) + (w.x * y);
}
```

Figure 4.2: *OpenCL global kernel*

similar structure as CUDA: The language is based on *C99* when programming a device. The standard is supplied by the *The Khronos Groups* and the implementation is supplied by the manufacturing company or device vendor such as AMD, INTEL, or NVIDIA.

OpenCL views the system from a perspective where computing resources (CPU or other accelerators) are a number of *compute devices* attached to a host (a CPU). The programs executed on a compute device is called a kernel. Programs in the OpenCL language are intended to be compiled at run-time to preserve portability between implementations from various host devices.

The OpenCL kernels are compiled by the host and then enqueued on a compute device. The kernel function accessible by the host to enqueue is specified with `__kernel`. Data residing in global memory is specified in the parameter list by `__global` and local memory have the specifier `__local`. The CUDA threads are in OpenCL terminology called *Work-items* and they are organized in *Work-groups*.

Similarly to CUDA the host application can poll the device for its capabilities and use some fast math function. The equivalent CUDA kernel in figure 4.1 is implemented in OpenCL in figure 4.2 and displays small differences. The OpenCL math function `sincos` is the equivalent of `__sincosf`.

```
[numthreads(GROUP_SIZE_X, 1, 1)]
void dx_global(
    uint3 threadIDInGroup : SV_GroupThreadID,
    uint3 groupID : SV_GroupID,
    uint groupIndex : SV_GroupIndex,
    uint3 dispatchThreadID : SV_DispatchThreadID)
{
    cpx w;
    int tid = groupID.x * GROUP_SIZE_X + threadIDInGroup.x;

    // Input offset
    int in_low = tid + (tid & mask);
    int in_high = in_low + dist;

    // Twiddle factor
    float a = angle * ((tid<<steps)&((dist - 1)<<steps));
    sincos(a, w.y, w.x);

    // Butterfly
    float x = input[in_low].x - input[in_high].x;
    float y = input[in_low].y - input[in_high].y;
    rw_buf[in_low].x  = input[in_low].x + input[in_high].x;
    rw_buf[in_low].y  = input[in_low].y + input[in_high].y;
    rw_buf[in_high].x = (w.x * x) - (w.y * y);
    rw_buf[in_high].y = (w.y * x) + (w.x * y);
}
```

Figure 4.3: *DirectCompute global kernel*

4.3 DirectCompute

Microsoft DirectCompute is an API that supports GPGPU on Microsoft's Windows Operating System (OS) (Vista, 7, 8, 10). DirectCompute is part of the *DirectX* collection of APIs. DirectX was created to support computer games development for the *Windows 95* OS. The initial release of DirectCompute was with DirectX 11 API, and have similarities with both CUDA and OpenCL. DirectCompute is designed and implemented with HLSL. The program (and kernel equivalent) is called a *compute shader*. The compute shader is not like the other types of shaders that are used in the graphic processing pipeline (like vertex or pixel shaders).

A difference from CUDA and OpenCL in implementing a compute shader compared to a kernel is the lack of C-like parameters: A *constant buffer* is used instead, where each value is stored in a read-only data structure. The setup share similarities with OpenCL and the program is compiled at run-time. The thread dimensions is built in as a constant value in the compute shader, and the block dimensions are specified at shader dispatch/execution.

As the code example demonstrated in figure 4.3 the shader body is similar to that of CUDA and OpenCL.

```
void main()
{
    cpx w;
    uint tid = gl_GlobalInvocationID.x;

    // Input offset
    uint in_low = tid + (tid & mask);
    uint in_high = in_low + dist;

    // Twiddle factor
    float a = angle * ((tid<<steps)&((dist - 1U)<<steps));
    w.x = cos(a);
    w.y = sin(a);

    // Butterfly
    cpx low = data[in_low];
    cpx high = data[in_high];
    float x = low.x - high.x;
    float y = low.y - high.y;
    data[in_low]  = cpx(low.x + high.x, low.y + high.y);
    data[in_high] = cpx((w.x*x)-(w.y*y),(w.y*x)+(w.x*y));
}
```

Figure 4.4: *OpenGL global kernel*

4.4 OpenGL

OpenGL share much of the same graphics inheritance as DirectCompute but also provides a compute shader that breaks out of the graphics pipeline. The OpenGL is managed by the Khronos Group and was released in 1992. Analogous to HLSL, OpenGL programs are implemented with GLSL. The differences between the two are subtle, but include how arguments are passed and the use of specifiers.

Figure 4.4 show the OpenGL version of the global kernel.

4.5 OpenMP

Open Multi-Processing (OpenMP) is an API for multi-platform shared memory multiprocessing programming. It uses a set of compiler directives and library routines to implement multithreading. OpenMP uses a master thread that *forks* slave threads where work is divided among them. The threads runs concurrently and are allocated to different processors by the runtime environment. The parallel section of the code is marked with preprocessor directives (`#pragma`) and when the threads are running the code they can access their respective id with the `omp_get_thread_num()` call. When the section is processed the threads *join* back into the master thread (with id 0).

Figure 4.5 shows how the *for-loop* section is parallelized by scheduling the workload evenly with the `static` keyword. An important difference from the GPU-implementations is that the twiddle factors are computed in advance and stored in memory. Another difference is the number of threads, which is a fixed num-

```
void add_sub_mul(cpx *l, cpx *u, cpx *out, cpx *w)
{
    float x = l->x - u->x;
    float y = l->y - u->y;
    *out = {l->x + u->x, l->y + u->y};
    *(++out)={(w->x*x)-(w->y*y),(w->y*x)+(w->x*y)};
}
void fft_stage(cpx *i, cpx *o, cpx *w, uint m, int r)
{
#pragma omp parallel for schedule(static)
    for (int l = 0; l < r; ++l)
        add_sub_mul(i+l,i+r+l,o+(l<<1),w+(l & m));
}
```

Figure 4.5: *OpenMP procedure completing one stage*

ber where each thread will work on a consecutive span of the iterated butterfly operations.

4.6 External libraries

External libraries were selected for reference values. FFTW and cuFFT were selected because they are frequently used in other papers. clFFT was selected by the assumption that it is the AMD equivalent of cuFFT for AMDs graphic cards.

FFTW

Fastest Fourier Transform in the West (FFTW) is a C subroutine library for computing the DFT. FFTW is a free software[11] that have been available since 1997, and several papers have been published about the FFTW [12, 13, 14]. FFTW supports a variety of algorithms and by estimating performance it builds a plan to execute the transform. The estimation can be done by either performance test of an extensive set of algorithms, or by a few known fast algorithms.

cuFFT

The library cuFFT (NVIDIA CUDA Fast Fourier Transform product) [21] is designed to provide high-performance on NVIDIA GPUs. cuFFT uses algorithms based on the Cooley-Tukey and the Bluestein algorithm [4].

clFFT

The library clFFT, found in [3] is part of the open source AMD Compute Libraries (ACL)[2]. According to an AMD blog post[1] the library performs at a similar level of cuFFT[1].

[1]The performance tests was done using *NVIDIA Tesla K40* for cuFFT and *AMD Firepro W9100* for clFFT.

5 Implementation

The FFT application has been implemented in C/C++, CUDA, OpenCL, DirectCompute, and OpenGL. The application was tested on a GeForce GTX 670 and a Radeon R7 R260X graphics card and on an Intel Core i7 3770K 3.5GHz CPU.

5.1 Benchmark application GPU

5.1.1 FFT

Setup

The implementation of the FFT algorithm on a GPU can be broken down into steps, see figure 5.1 for a simplified overview. The application setup differs among the tested technologies, however some steps can be generalized; get platform and device information, allocate device buffers and upload data to device.

The next step is to calculate the specific FFT arguments for a N-point sequence for each kernel. The most important differences between devices and platforms are local memory capacity and thread and block configuration. Threads per block was selected for the best performance. See table 5.1 for details.

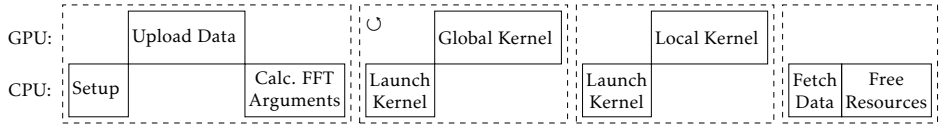


Figure 5.1: *Overview of the events in the algorithm.*

Device	Technology	Threads / Block	Max Threads	Shared memory
GeForce GTX 670	CUDA	1024	1024	49152
	OpenCL	512		49152
	OpenGL	1024		32768
	DirectX	1024		32768
Radeon R7 260X	OpenCL	256	256	32768
	OpenGL	256		
	DirectX	256		

Table 5.1: *Shared memory size in bytes, threads and block configuration per device.*

Thread and block scheme

The threading scheme was one butterfly per thread, so that a sequence of sixteen points require eight threads. Each platform was configured to a number of threads per block (see table 5.1): any sequences requiring more butterfly operations than the threads per block configuration needed the computations to be split over several blocks. In the case of a sequence exceeding one block, the sequence is mapped over the `blockIdx.y` dimension with size `gridDim.y`. The block dimensions are limited to 2^{31}, 2^{16}, 2^{16} respectively for `x`, `y`, `z`. Example: if the threads per block limit is two, then four blocks would be needed for a sixteen point sequence.

Synchronization

Thread synchronization is only available of threads within a block. When the sequence or partial sequence fitted within a block, that part was transferred to local memory before computing the last stages. If the sequence was larger and required more than one block, the synchronization was handled by launching several kernels in the same stream to be executed in sequence. The kernel launched for block wide synchronization is called the *global kernel* and the kernel for thread synchronization within a block is called the *local kernel*. The global kernel had an implementation of the Cooley-Tukey FFT algorithm, and the local kernel had constant geometry (same indexing for every stage). The last stage outputs data from the shared memory in bit-reversed order to the global memory. See figure 5.2, where the sequence length is 16 and the threads per block is set to two.

Calculation

The indexing for the global kernel was calculated from the thread id and block id (`threadIdx.x` and `blockIdx.x` in CUDA) as seen in figure 5.3. Input and output is located by the same index.

Index calculation for the local kernel is done once for all stages, see figure 5.4. These indexes are separate from the indexing in the global memory. The global memory offset depends on threads per block (`blockDim.x` in CUDA) and block id.

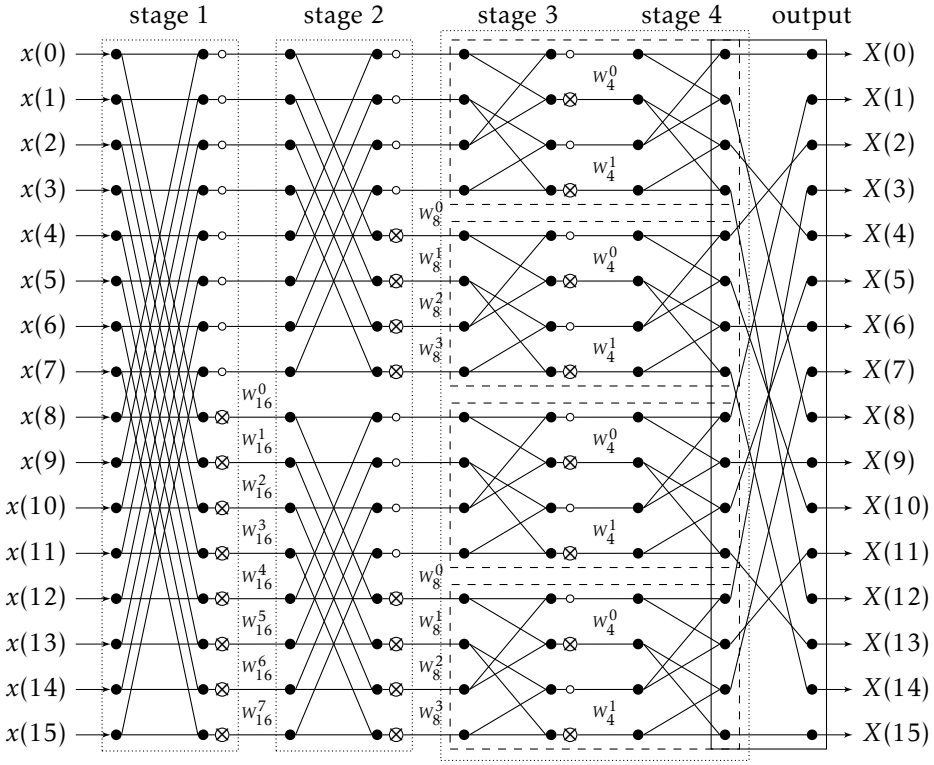


Figure 5.2: *Flow graph of a 16-point FFT using (stage 1 and 2) Cooley-Tukey algorithm and (stage 3 and 4) constant geometry algorithm. The solid box is the bit-reverse order output. Dotted boxes are separate kernel launches, dashed boxes are data transfered to local memory before computing the remaining stages.*

```
int tid     = blockIdx.x * blockDim.x + threadIdx.x,
    io_low  = tid + (tid & (0xFFFFFFFF << stages_left)),
    io_high = io_low + (N >> 1);
```

Figure 5.3: *CUDA example code showing index calculation for each stage in the global kernel, N is the total number of points.* `io_low` *is the index of the first input in the butterfly operation and* `io_high` *the index of the second.*

```
int n_per_block = N / gridDim.x.
    in_low      = threadId.x.
    in_high     = threadId.x + (n_per_block >> 1).
    out_low     = threadId.x << 1.
    out_high    = out_low + 1;
```

Figure 5.4: *CUDA code for index calculation of points in shared memory.*

Technology	Language	Signature
CUDA	C extension	`__brev(unsigned int);`
OpenGL	GLSL	`bitfieldReverse(unsigned int);`
DirectCompute	HLSL	`reversebits(unsigned int);`

Table 5.2: *Integer intrinsic bit-reverse function for different technologies.*

The last operation after the last stage is to perform the bit-reverse indexing operation, this is done when writing from shared to global memory. The implementation of bit-reverse is available as an intrinsic integer instruction (see table 5.2). If the bit-reverse instruction is not available, figure 5.5 shows the code used instead. The bit-reversed value had to be right shifted the number of zeroes leading the number in a 32-bit integer type value. Figure 5.2 show the complete bit-reverse operations of a 16-point sequence in the output step after the last stage.

5.1.2 FFT 2D

The FFT algorithm for 2D data, such as images, is first transformed row-wise (each row as a separate sequence) and then an equal transform of each column. The application performs a row-wise transformation followed by a transpose of the image to reuse the row-wise transform procedure for the columns. This method gives better memory locality when transforming the columns. A transformed image is shown in figure 5.6.

```
x = (((x & 0xaaaaaaaa) >> 1) | ((x & 0x55555555) << 1));
x = (((x & 0xcccccccc) >> 2) | ((x & 0x33333333) << 2));
x = (((x & 0xf0f0f0f0) >> 4) | ((x & 0x0f0f0f0f) << 4));
x = (((x & 0xff00ff00) >> 8) | ((x & 0x00ff00ff) << 8));
return((x >> 16) | (x << 16));
```

Figure 5.5: *Code returning a bit-reversed unsigned integer where x is the input. Only 32-bit integer input and output.*

(a) Original image *(b) Magnitude representation*

Figure 5.6: *Original image in 5.6a transformed and represented with a quadrant shifted magnitude visualization (scale skewed for improved illustration) in 5.6b.*

The difference between the FFT kernel for 1D and 2D is the indexing scheme. 2D rows are indexed with `blockIdx.x`, and columns with `threadIdx.x` added with an offset of `blockIdx.y` · `blockDim.x`.

Transpose

The transpose kernel uses a different index mapping of the 2D-data and threads per blocks than the FFT kernel. The data is tiled in a grid pattern where each tile represents one block, indexed by `blockIdx.x` and `blockIdx.y`. The tile size is a multiple of 32 for both dimensions and limited to the size of the shared memory buffer, see table 5.1 for specific size per technology. To avoid the banking issues, the last dimension is increased with one but not used. However, resolving the banking issue have little effect on total running-time so when shared memory is limited to 32768, the extra column is not used. The tile rows and columns are divided over the `threadIdx.x` and `threadIdx.y` index respectively. See figure 5.7 for a code example of the transpose kernel.

Shared memory example: The CUDA shared memory can allocate 49152 bytes and a single data point require `sizeof(float)` · 2 = 8 bytes. That leaves room for a tile size of $64 \cdot (64 + 1) \cdot 8 = 33280$ bytes. Where the integer 64 is the highest power of two that fits.

The transpose kernel uses the shared memory and tiling of the image to avoid large strides through global memory. Each block represents a tile in the image. The first step is to write the complete tile to shared memory and synchronize the threads before writing to the output buffer. Both reading from the input memory

```
__global__ void transpose(cpx *in, cpx *out, int n)
{
  uint tx = threadIdx.x;
  uint ty = threadIdx.y;
  __shared__ cpx tile[CU_TILE_DIM][CU_TILE_DIM + 1];
  // Write to shared (tile) from global memory (in)
  int x = blockIdx.x * CU_TILE_DIM + tx;
  int y = blockIdx.y * CU_TILE_DIM + ty;
  for (int j = 0; j < CU_TILE_DIM; j += CU_BLOCK_DIM)
    for (int i = 0; i < CU_TILE_DIM; i += CU_BLOCK_DIM)
      tile[ty+j][tx+i]=in[(y+j)*n+(x+i)];
  __syncthreads();
  // Write to global (out) from shared memory (tile)
  x = blockIdx.y * CU_TILE_DIM + tx;
  y = blockIdx.x * CU_TILE_DIM + ty;
  for (int j = 0; j < CU_TILE_DIM; j += CU_BLOCK_DIM)
    for (int i = 0; i < CU_TILE_DIM; i += CU_BLOCK_DIM)
      out[(y+j)*n+(x+i)]=tile[tx+i][ty+j];
}
```

Figure 5.7: *CUDA device code for the transpose kernel.*

and writing to the output memory is performed in close stride. Figure 5.8 shows how the transpose is performed in memory.

5.1.3 Differences

Setup

The majority of differences in the implementations were related to the setup phase. The CUDA implementation is the most straightforward, calling the procedure `cudaMalloc()` to allocate a buffer and `cudaMemcpy()` to populate it. With CUDA you can write the device code in the same file as the host code and share functions. OpenCL and OpenGL require a `char *` buffer as the kernel source and is most practical if written in a separate source file and read as a file stream to a `char` buffer. DirectCompute shaders are most easily compiled from file. Figure 5.9 gives a simplified overview of how to setup the kernels in the different technologies.

Kernel execution

Kernel launch in CUDA is like a procedure call in C/C++ with the difference that the kernel launch syntax require <<< and >>> to set the block and thread dimension. The special syntax is used in-between the name and the function operators `()`, see example: `kernel_name<<<threadDim, blockDim>>>()`.

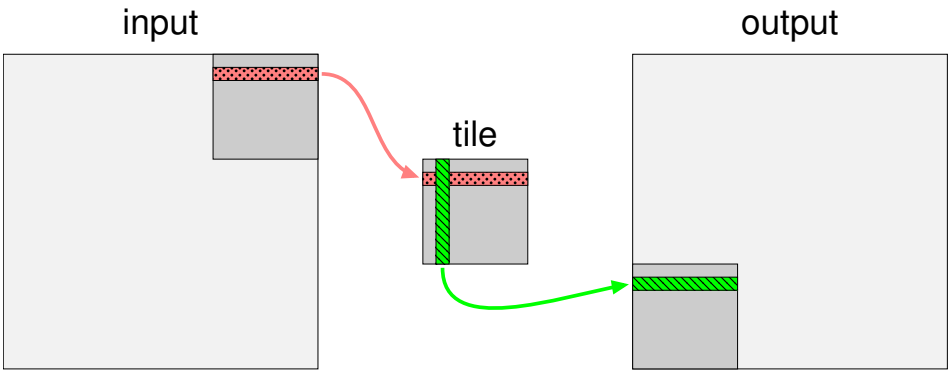


Figure 5.8: *Illustration of how shared memory is used in transposing an image. Input data is tiled and each tile is written to shared memory and transposed before written to the output memory.*

```
// Allocate memory buffer on device
cudaMalloc
// Upload data
cudaMemcpy
```

(a) *CUDA setup*

```
// Select platform from available
clGetPlatformIDs
clGetDeviceIDs
clCreateContext
clCreateCommandQueue
clCreateBuffer
// Upload data
clEnqueueWriteBuffer
// Create kernel
clCreateProgramWithSource
clBuildProgram
clCreateKernel
```

(b) *OpenCL setup*

```
// Select adapter from
// enumeration not shown
D3D11CreateDevice
device->CreateBuffer
// Data upload
context->UpdateSubresource
// Creation of views for access.
device->CreateShaderResourceView
device->CreateUnorderedAccessView
D3DCompileFromFile
device->CreateComputeShader
context->CSSetConstantBuffers
```

(c) *DirectCompute setup*

```
// OpenGL needs a valid rendering
// context, supplied by the OS.
glutInit(&argc, argv);
glutCreateWindow("GLContext");
glewInit();
// Create compute shader
glCreateProgram();
glCreateShader(GL_COMPUTE_SHADER);
glShaderSource
glCompileShader
glAttachShader
glLinkProgram
// Create buffer and upload
glGenBuffers
glBindBufferBase
glBufferData
```

(d) *OpenGL setup*

Figure 5.9: *An overview of the setup phase for the GPU technologies.*

The other technologies require some more setup; OpenCL and OpenGL need one function call per parameter. OpenCL maps with index, whereas OpenGL maps with a string to the parameter. In DirectCompute a constant buffer is suitable for the parameters. The constant buffer is read-only and accessible globally over the kernel. DirectCompute and OpenGL share a similar launch style where the compute shader is set as the current program to the device context, and a dispatch call is made with the group (block) configuration. See table 5.3 for a list of how the kernels are launched.

Technology	Code to set parameters and execute kernel
CUDA	`cuda_kernel<<<blocks, threads>>>(in, out,...);`

OpenCL	`clSetKernelArg(kernel, 0, sizeof(cl_mem), &in);`
	`clSetKernelArg(kernel, 0, sizeof(cl_mem), &out);`
	`/* Set rest of the arguments. */`
	`clEnqueueNDRangeKernel(cmd_queue, kernel, dim, 0, work_sz, ...);`

DirectCompute	`context->CSSetUnorderedAccessViews(0, 1, output_uav, NULL);`
	`context->CSSetShaderResources(0, 1, &input_srv);`
	`context->CSSetShader(compute_shader, nullptr, 0);`
	`arguments = {/* Struct holding all arguments */ ...}`
	`dx_map_args<dx_cs_args>(context, constant_buffer, &arguments);`
	`context->Dispatch(groups.x, groups.y, groups.z);`

OpenGL	`glUseProgram(program);`
	`glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, buffer_in);`
	`glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 1, buffer_out);`
	`glUniform1f(glGetUniformLocation(program, "angle"), angle);`
	`/* Set rest of the arguments. */`
	`glDispatchCompute(groups.x, groups.y, groups.z)`

Table 5.3: *Table illustrating how to set parameters and launch a kernel.*

Kernel code

The kernel code for each technology had a few differences. CUDA have the strongest support for a C/C++ -like language and only adds a function-type specifier. The kernel program is accessible from the host via the `__global__` specifier. OpenCL share much of this but is restricted to a C99-style in the current version (2.0). A difference is how global and local buffers can referenced, these must be declared with the specifier `__global` or `__local`.

DirectCompute and OpenGL Compute Shader is coded in HLSL and GLSL respectively. These languages are similar and share the same level of restrictions compared to CUDA C/C++ and OpenCL C-code. Device functions can not use pointers or recursion. However, these are of little importance for the performance since all code is in-lined in the building and compilation of the kernel program.

Synchronization

The synchronization on thread and block-level is different. In CUDA can threads within a block can be synchronized directly on the GPU. On block-level synchronization, the host is required as in building the stream or queue of commands. The equivalent exists in all technologies. Device and host synchronization is similar to block-level synchronization, however, in DirectCompute this is not done

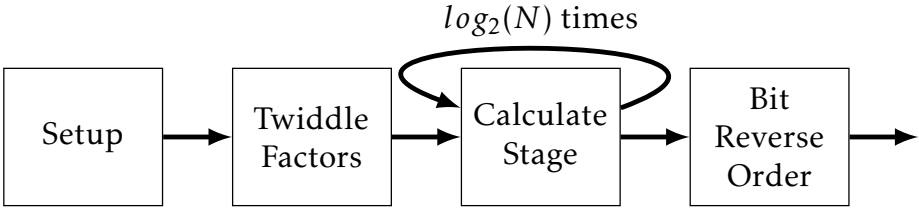


Figure 5.10: *OpenMP implementation overview transforming sequence of size N.*

trivially as with a blocking function call from the host. CUDA, OpenCL and DirectCompute uses the same kernel stream to execute kernels sequentially, while the sequential execution in OpenGL is accomplished by the use of

```
glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT)
```

in between launches. See table 5.4 for the synchronization functions used.

Technology	Threads in blocks	Host and Device
CUDA	`__syncthreads();`	`cudaDeviceSynchronize();`
OpenCL	`barrier(CLK_LOCAL_MEM_FENCE);`	`clFinish(cmd_queue);`
OpenGL	`barrier();`	`glFinish();`
DirectCompute	`GroupMemoryBarrierWithGroupSync();`	-

Table 5.4: *Synchronization in GPU technologies.*

5.2 Benchmark application CPU

5.2.1 FFT with OpenMP

The OpenMP implementation benefits in performance from calculating the twiddle factors in advance. The calculated values are stored in a buffer accessible from all threads. The next step is to calculate each stage of the FFT algorithm. Last is the output index calculation where elements are reordered. See figure 5.10 for an overview.

Twiddle factors

The twiddle factor are stored for each butterfly operation. To save time, only the real part is calculated and the imaginary part is retrieved from the real parts due to the fact that $\sin(x) = \cos(\pi/2 + x)$ and $\sin(\pi/2 + x) = -\cos(x)$. See figure 5.5 for an example. The calculations will be split among the threads by static scheduling in two steps: first calculate the real values, then copy from real to imaginary.

Butterfly

The same butterfly operation uses the constant geometry index scheme. The indexes are not stored from one stage to the next but it makes the output come in

Twiddle factor table W

i	$\Re\mathfrak{e}(W)$	$\Im\mathfrak{m}(W)$
0	$\cos(\alpha \cdot 0)$	$\Re\mathfrak{e}(W[4])$
1	$\cos(\alpha \cdot 1)$	$\Re\mathfrak{e}(W[5])$
2	$\cos(\alpha \cdot 2)$	$\Re\mathfrak{e}(W[6])$
3	$\cos(\alpha \cdot 3)$	$\Re\mathfrak{e}(W[7])$
4	$\cos(\alpha \cdot 4)$	$-\Re\mathfrak{e}(W[0])$
5	$\cos(\alpha \cdot 5)$	$-\Re\mathfrak{e}(W[1])$
6	$\cos(\alpha \cdot 6)$	$-\Re\mathfrak{e}(W[2])$
7	$\cos(\alpha \cdot 7)$	$-\Re\mathfrak{e}(W[3])$

Table 5.5: *Twiddle factors for a 16-point sequence where α equals $(2 \cdot \pi)/16$. Each row i corresponds to the ith butterfly operation.*

```
void omp_bit_reverse(cpx *x, int leading_bits, int N)
{
#pragma omp parallel for schedule(static)
    for (int i = 0; i <= N; ++i) {
        int p = bit_reverse(i, leading_bits);
        if (i < p)
            swap(&(x[i]), &(x[p]));
    }
}
```

Figure 5.11: *C/C++ code performing the bit-reverse ordering of a N-point sequence.*

continuous order. The butterfly operations are split among the threads by static scheduling.

Bit-Reversed Order

See figure 5.11 for code showing the bit-reverse ordering operation in C/C++ code.

5.2.2 FFT 2D with OpenMP

The implementation of 2D FFT with OpenMP runs the transformations row-wise and transposes the image and repeat. The twiddle factors are calculated once and stays the same.

5.2.3 Differences with GPU

The OpenMP implementation is different from the GPU-implementations in two ways: twiddle factors are pre-computed, and all stages uses the constant geom-

Platform	Model	Library name	Version
NVIDIA GPU	GeForce GTX 670	cuFFT	7.5
AMD GPU	Radeon R7 260X	clFFT	2.8.0
Intel CPU	Core i7 3770K 3.5GHz	FFTW	3.3.4

Table 5.6: *Libraries included to compare with the implementation.*

etry algorithm. The number of parallel threads are on the Intel Core i7 3770K 3.5GHz CPU four, whereas the number is on the GPU up to 192 (seven warps, one per Streaming Multiprocessor (SM), with 32 threads each).

5.3 Benchmark configurations

5.3.1 Limitations

All implementations are limited to handle sequences of 2^n length or $2^m \times 2^m$ where n and m are integers with maximal value of $n = m + m = 26$. The selected GPUs have a maximum of 2GB global memory available. The limitation is required since the implementation uses a total of $2^{26} \cdot \texttt{sizeof(float2)} \cdot 2 = 1073741824$ bytes. However on the Radeon R7 R260X card, problems with DirectCompute and OpenGL set the limit lower. DirectCompute handled sizes of $n <= 2^{24}$ and OpenGL $n <= 2^{24}$ and $m <= 2^9$.

5.3.2 Testing

All tests executed on the GPU utilize some implementation of event time stamps. The time stamp event retrieve the actual start of the kernel if the current stream is busy. The CPU implementations used Windows *QueryPerformanceCounter* function, which is a high resolution ($< 1\mu s$) time stamp.

5.3.3 Reference libraries

One reference library per platform was included to compare how well the FFT implementation performed. The *FFTW* library for the CPU, runs a planning scheme to create an execution plan for each environment and data input size. Similar strategy is used in the cuFFT and clFFT libraries used for the GeForce GTX 670 and Radeon R7 R260X respectively. Table 5.6 sums up information about the external libraries.

6 Evaluation

6.1 Results

The results will be shown for the two graphics cards GeForce GTX 670 and Radeon R7 R260X, where the technologies were applicable. The tested technologies are shown in table 6.1. The basics of each technology or library is explained in chapter 4.

The performance measure is total execution time for a single forward transform using two buffers: one input and one output buffer. The implementation input size range is limited by the hardware (graphics card primary memory). However there are some unsolved issues near the upper limit on some technologies on the Radeon R7 R260X.

CUDA is the primary technology and the GeForce GTX 670 graphics card is the primary platform. All other implementations are ported from CUDA implementation. To compare the implementation, external libraries are included and can be found in italics in the table 6.1. Note that the clFFT library failed to be measured in the same manner as the other GPU implementations: the times are measured at host, and short sequences suffer from large overhead.

The experiments are tuned on two parameters, the number of threads per block and how large the tile dimensions are in the transpose kernel, see chapter 5 and table 5.1.

[1] Free software, available at [11].
[2] Available through the *CUDAToolkit* at [22].
[3] OpenCL FFT library available at [3].

Platform	Tested technology
Intel Core i7 3770K 3.5GHz CPU	C/C++ OpenMP *FFTW*[1]
GeForce GTX 670	CUDA OpenCL DirectCompute OpenGL *cuFFT*[2]
Radeon R7 R260X	OpenCL DirectCompute OpenGL *clFFT*[3]

Table 6.1: *Technologies included in the experimental setup.*

6.1.1 Forward FFT

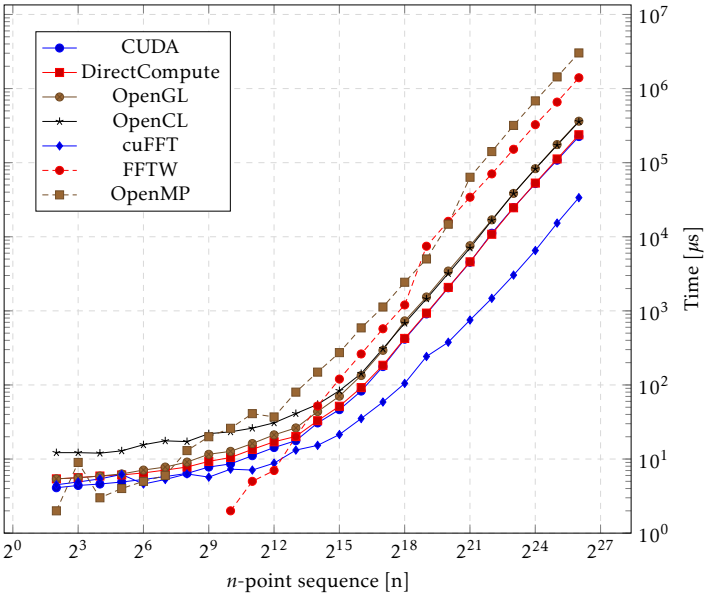
The results for a single transform over a 2^n-point sequence are shown in figure 6.1 for the GeForce GTX 670, Radeon R7 R260X and Intel Core i7 3770K 3.5GHz CPU.

The CUDA implementation was the fastest on the GeForce GTX 670 over most sequences. The OpenCL implementation was the only technology that could run the whole test range on the Radeon R7 R260X. DirectCompute was limited to 2^{24} points and OpenGL to 2^{23} points. A normalized comparison using CUDA and OpenCL is shown in figure 6.2.

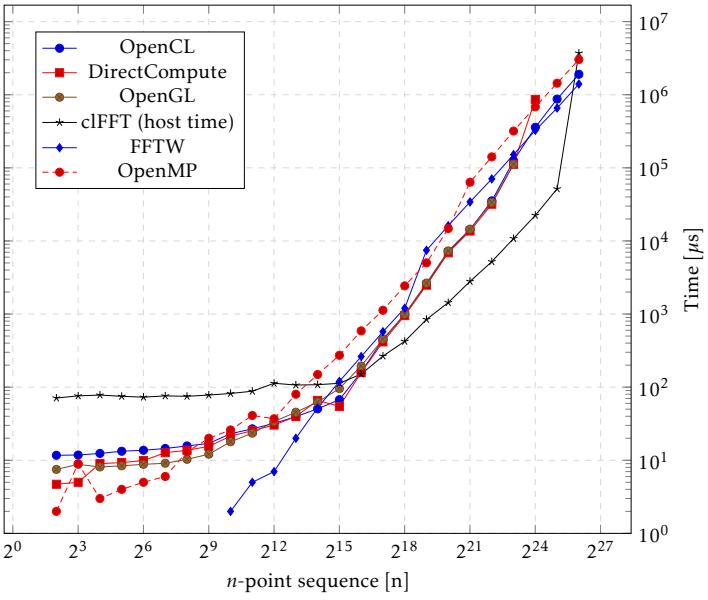
The experimental setup for the CPU involved low overhead and the short sequences could not be measured accurately. This is shown as $0\mu s$ in the figures. Results from comparing the sequential C/C++ and multi-core OpenMP implementation with CUDA are shown in figure 6.3. FFTW was included and demonstrated how an optimized (per n-point length) sequential CPU implementation perform.

Results from comparing the implementations on the different graphics cards are shown in figure 6.4. The results are normalized on the result of the tests on the GeForce GTX 670.

DirectCompute, OpenGL, and OpenCL was supported on both graphics cards, the results of normalizing the resulting times with the time of the OpenCL implementation is shown in figure 6.5.

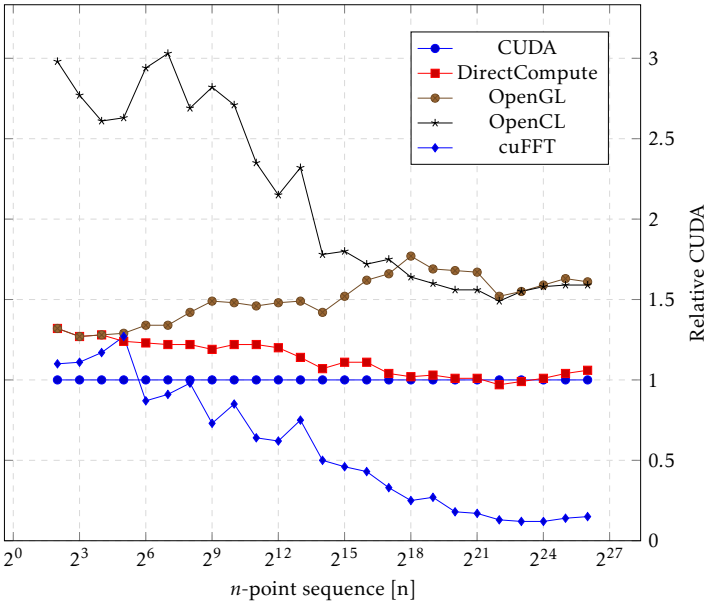


(a) GeForce GTX 670

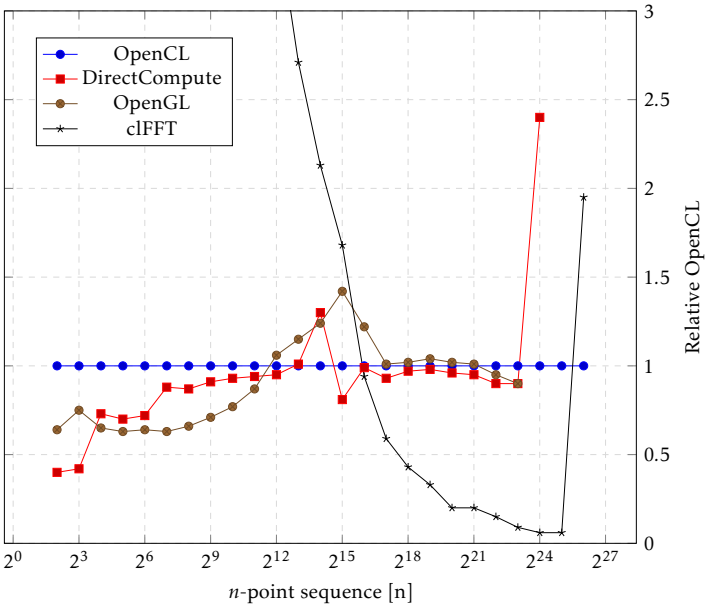


(b) Radeon R7 R260X

Figure 6.1: *Overview of the results of a single forward transform. The clFFT was timed by host synchronization resulting in an overhead in the range of* 60µs*. Lower is faster.*



(a) *GeForce GTX 670*



(b) *Radeon R7 R260X*

Figure 6.2: *Performance relative CUDA implementation in 6.2a and OpenCL in 6.2b. Lower is better.*

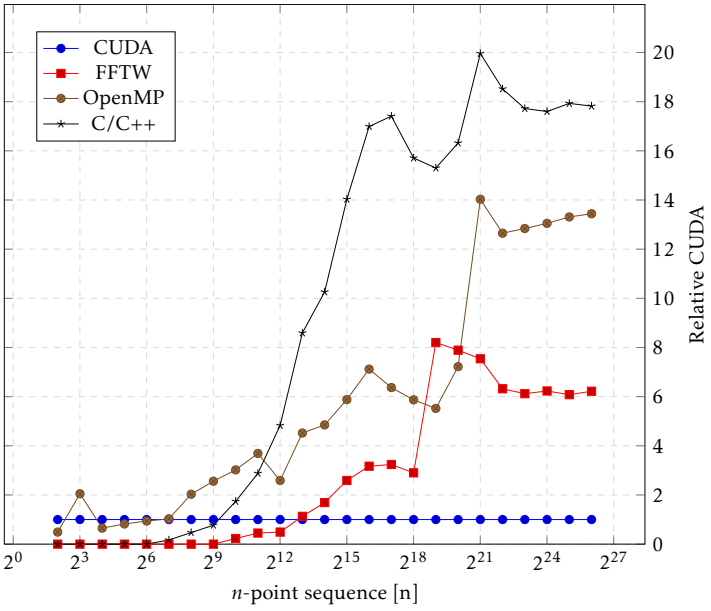


Figure 6.3: *Performance relative CUDA implementation on GeForce GTX 670 and Intel Core i7 3770K 3.5GHz CPU.*

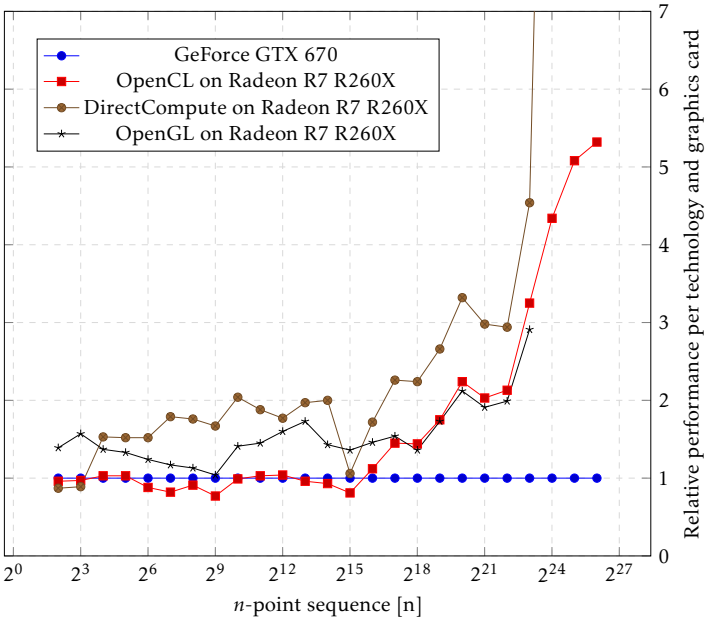


Figure 6.4: *Comparison between Radeon R7 R260X and GeForce GTX 670.*

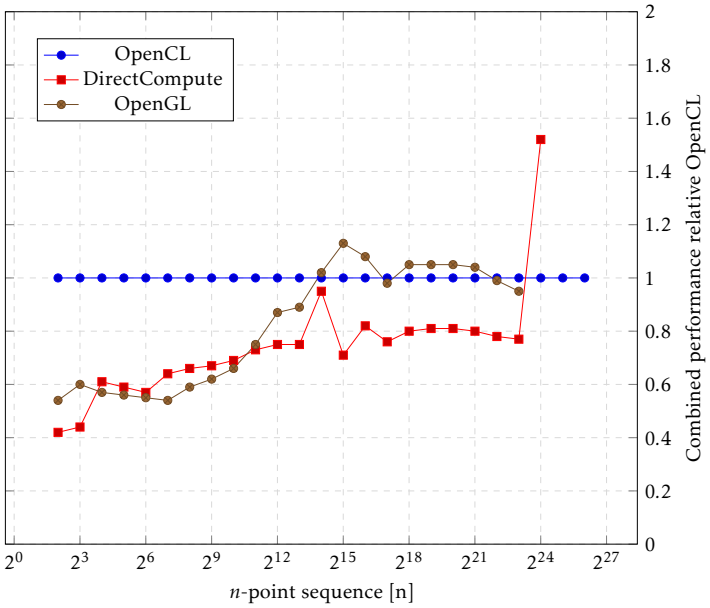


Figure 6.5: *Performance relative OpenCL accumulated from both cards.*

6.1.2 FFT 2D

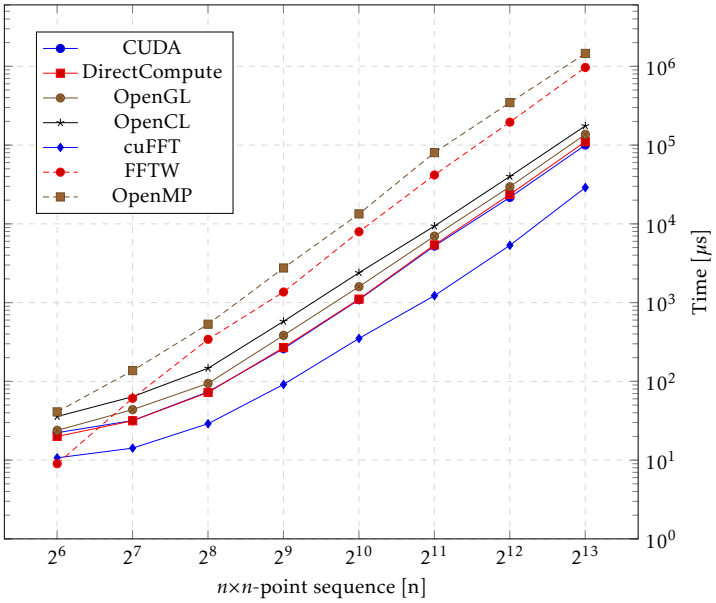
The equivalent test was done for 2D-data represented by an image of $m \times m$ size. The image contained three channels (red, green, and blue) and the transformation was performed over one channel. Figure 6.6 shows an overview of the results of image sizes ranging from $2^6 \times 2^6$ to $2^{13} \times 2^{13}$.

All implementations compared to CUDA and OpenCL on the GeForce GTX 670 and Radeon R7 R260Xrespectively are shown in 6.7. The OpenGL implementation failed at images larger then $2^{11} \times 2^{11}$ points.

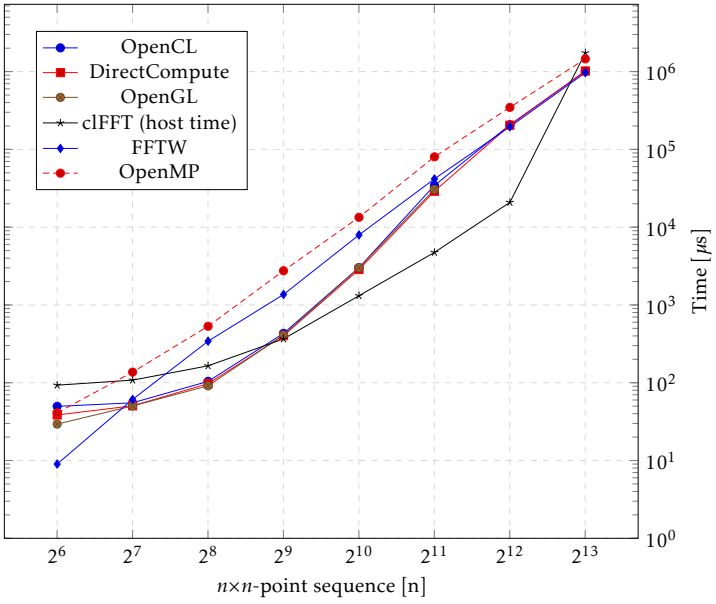
The results of comparing the GPU and CPU handling of a 2D forward transform is shown in figure 6.8.

Comparison of the two cards are shown in figure 6.9.

DirectCompute, OpenGL and OpenCL was supported on both graphics cards, the results of normalizing the resulting times with the time of the OpenCL implementation is shown in figure 6.10.

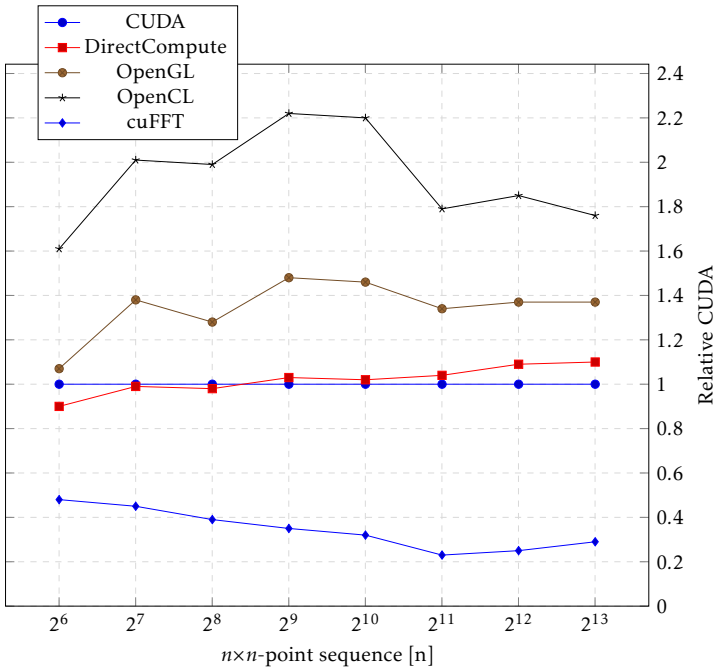


(a) GeForce GTX 670

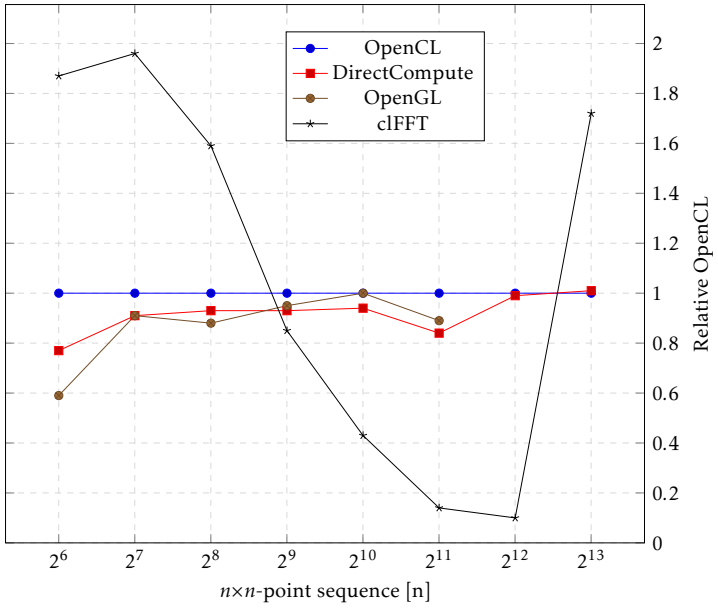


(b) Radeon R7 R260X

Figure 6.6: *Overview of the results of measuring the time of a single 2D forward transform.*



(a) *GeForce GTX 670*



(b) *Radeon R7 R260X*

Figure 6.7: *Time of 2D transform relative CUDA in 6.7a and OpenCL in 6.7b.*

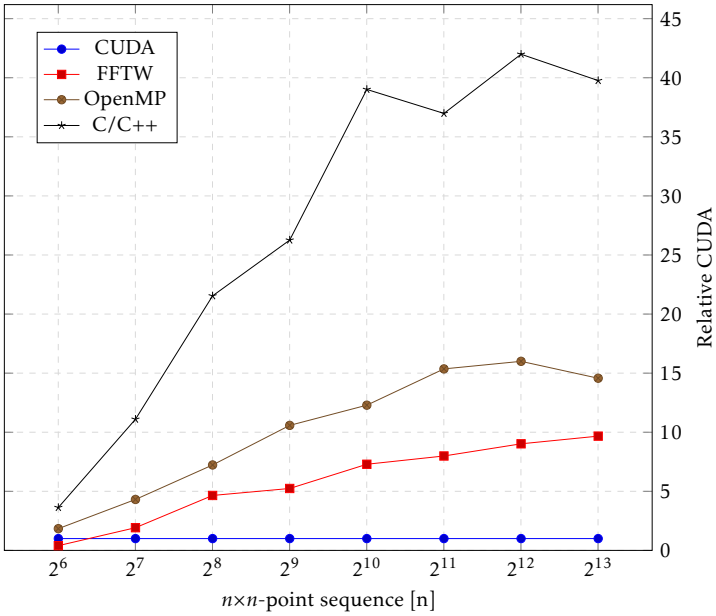


Figure 6.8: *Performance relative CUDA implementation on GeForce GTX 670 and Intel Core i7 3770K 3.5GHz CPU.*

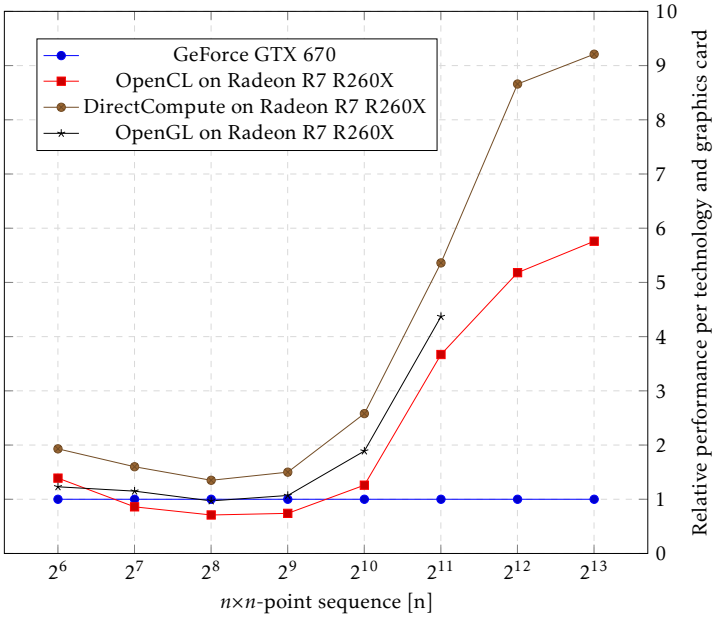


Figure 6.9: *Comparison between Radeon R7 R260X and GeForce GTX 670 running 2D transform.*

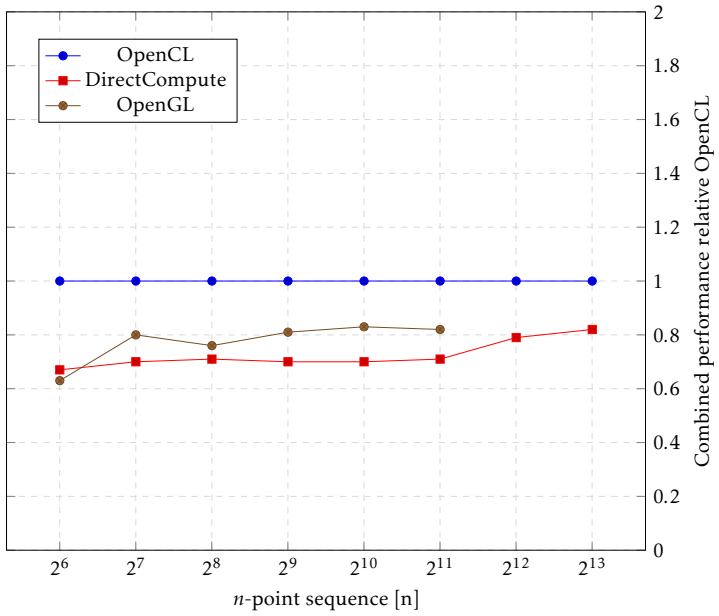


Figure 6.10: *Performance relative OpenCL accumulated from both cards.*

6.2 Discussion

The foremost known technologies for GPGPU, based on other resarch-interests, are CUDA and OpenCL. The comparisons from earlier work have focused primarily on the two [10, 25, 26]. Bringing DirectCompute (or Direct3D Compute Shader) and OpenGL Compute Shader to the table makes for an interesting mix since the result from the experiment is that both are strong alternatives in terms of raw performance.

The most accurate and fair comparison with a GPU is when number of data is scaled up, the least amount of elements should be in the order of 2^{12}. By not fully saturating the GPUs streaming multiprocessors, there is less gain from moving from the CPU. One idea is to make sure that even if the sequences are short, they should be calculated in batches. The results from running the benchmark application on small sets of data are more or less discarded in the evaluation.

The CPU vs GPU

The implementation aimed at sequences of two-dimensional data was however successful at proving the strength of the GPU versus the CPU. The difference in execution time of the CPU and GPU is a factor of 40 times slower when running a 2D FFT over large data. Compared to the multi-core OpenMP solution, the difference is still a factor of 15 times slower. Even the optimized FFTW solution is a factor of 10 times slower. As a side note, the cuFFT is 36 times faster than

FFTW on large enough sequences, they do use the same strategy (build an execution plan based on current hardware and data size) and likely use different hard-coded unrolled FFTs for smaller sizes.

The GPU

The unsurprising result from the experiments is that CUDA is the fastest technology on GeForce GTX 670, but only with small a margin. What might come as surprise, is the strength of the DirectCompute implementation. Going head-to-head with CUDA (only slightly slower) on the GeForce GTX 670, and performing equally (or slightly faster) than OpenCL.

OpenGL is performing on par with DirectCompute on the Radeon R7 R260X. The exception is the long sequences that fails with the OpenGL-solution on the Radeon R7 R260X, sequences otherwise working on the GeForce GTX 670. The performance of the OpenGL tests are equal or better then OpenCL in 1D, but outperforming OpenCL in 2D.

The biggest surprise is actually the OpenCL implementation. Falling behind by a relatively big margin on both graphics cards. This large margin was not anticipated based on other papers in comparisons. Effort has been made to assure that the code does in fact run fairly compared to the other technologies. The ratio for OpenCL versus CUDA on long sequences are about 1.6 and 1.8 times slower for 1D and 2D respectively on the GeForce GTX 670. The figure 6.5 and 6.10 shows that DirectCompute is faster by a factor of about 0.8 of the execution-time of OpenCL. The same comparisons of OpenCL and OpenGL shows similar results. The one thing that goes in favor of OpenCL is that the implementation did scale without problem: All sequences were computed as expected. The figures 6.2b and 6.7b shows that something happened with the other implementations, even clFFT had problem with the last sequence. OpenGL and DirectCompute could not execute all sequences.

External libraries

Both FFTW on the CPU and cuFFT on the GeForce GTX 670 proved to be very mature and optimized solutions, far faster then any of my implementations on respective platform. Not included in the benchmark implementation is a C/C++ implementation that partially used the concept of the FFTW (a decomposition with hard-coded unrolled FFTs for short sequences) and was fairly fast at short sequences compared to FFTW. Scalability proved to be poor and provided very little gain in time compared to much simpler implementations such as the constant geometry algorithm.

cuFFT proved stable and much faster than any other implementation on the GPU. The GPU proved stronger than the CPU at data sizes of 2^{12} points or larger, this does not include memory transfer times. Comparing cuFFT with clFFT was possible on the GeForce GTX 670, but that proved only that clFFT was not at all written for that architecture and was much slower at all data sizes. A big problem when including the clFFT library was that measuring by events on the device

failed, and measuring at the host included an overhead. Short to medium-long sequences suffered much from the overhead, a quick inspection suggests of close to $60\mu s$ (comparing to total runtime of the OpenCL at around $12\mu s$ for short sequences). Not until sequences reached 2^{16} elements or greater could the library beat the implementations in the application. The results are not that good either, a possible explanation is that the clFFT is not at all efficient at executing transforms of small batches, the blog post at [1] suggest a completely different result when running in batch and on GPUs designed for computations. The conclusions in this work are based on cards targeting the gaming consumer market and variable length sequences.

6.2.1 Qualitative assessment

When working with programming, raw performance is seldom the only requirement. This subsection will provide qualitative based assessments of the technologies used.

Scalability of problems

The different technologies are restricted in different ways. CUDA and OpenCL are device limited and suggest polling the device for capabilities and limitations. DirectCompute and OpenGL are standardized with each version supported. An example of this is the shared memory size limit: CUDA allowed for full access, whereas DirectCompute was limited to a API-specific size and not bound by the specific device. The advantage of this is the ease of programming with DirectCompute and OpenGL when knowing that a minimum support is expected at certain feature support versions.

Both DirectCompute and OpenGL had trouble when data sizes grew, no such indications when using CUDA and OpenCL.

Portability

OpenCL have a key feature of being portable and open for many architecture enabling computations. However, as stated in [10, 9], performance is not portable over platforms but can be addressed with auto-tuning at the targeted platform. There were no problems running the code on different graphic cards on either OpenCL or DirectCompute. OpenGL proved to be more problematic with two cards connected to the same host. The platform-specific solution using either OS tweaking or specific device OpenGL expansions made OpenGL less convenient as a GPGPU platform. CUDA is a proprietary technology and only usable with NVIDIAs own hardware.

Moving from the GPU, the only technology is OpenCL and here is where it excels among the others. This was not in the scope of the thesis however it is worth noting that it would be applicable with minor changes in the application.

Programmability

The experience of this work was that CUDA was by far the least complicated to implement. The fewest lines of code needed to get started and compared to

C/C++ there were fewer limitations. The CUDA community and online documentation is full of useful information, finding solutions to problems was relatively easy. The documentation[4] provided guidance for most applications.

OpenCL implementation was not as straight forward as CUDA. The biggest difference is the setup. Some differences in the setup are:

- Device selection is not needed actively in CUDA
- Command queue or stream is created by default in CUDA
- CUDA creates and builds the kernel run-time instead of compile-time.

Both DirectCompute and OpenGL follows this pattern, although they inherently suffer from graphic specific abstractions. The experience was that creating and handle memory-buffers was more prone to mistakes. Extra steps was introduced to create and use the memory in a compute shader compared to a CUDA and OpenCL-kernel.

The biggest issue with OpenGL is the way the device is selected, it is handled by the OS. Firstly, in the case of running *Windows 10*, the card had to be connected to a screen. Secondly, that screen needed to be selected as the primary screen. This issue is also a problem when using services based on Remote Desktop Protocol (RDP). RDP enables the user to log in to a computer remotely. This works for the other technologies but not for OpenGL. Not all techniques for remote access have this issue, it is convenient if the native tool in the OS support GPGPU features such as selecting the device, especially when running a benchmarking application.

6.2.2 Method

The first issue that have to be highlighted is the fact that 1D FFTs were measured by single sequence execution instead of executing sequences in batch. The 2D FFT implementation did inherently run several 1D sequences in batch and provided relevant results. One solution would have been to modify the 2D transformation to accept a batch of sequences organized as 2D data. The second part of performing column-wise transformations would then be skipped.

The two graphics cards used are not each others counterparts, the releases differs 17 months (GeForce GTX 670 released May 10, 2012 compared to Radeon R7 R260X in October 8, 2013). The recommended release price hints the targeted audience and capacity of the cards, the GeForce GTX 670 was priced at $400 compared to the Radeon R7 R260X at $139. Looking at the OpenCL performance on both cards as seen in figure 6.9, revealed that medium to short sequences are about the same, but longer sequences goes in favour of the GeForce GTX 670.

Algorithm

The FFT implementation was rather straight forward and without heavy optimization. The implementation lacked in performance compared to the NVIDIA

[4] *https://docs.nvidia.com/cuda/cuda-c-programming-guide/*

developed cuFFT library. Figure 6.7a shows cuFFT to perform three to four times faster then the benchmark application. Obviously there is a lot to improve. This is not a big issue when benchmarking, but it removes some of the credibility of the algorithm as something practically useful.

By examining the code with NVIDIA Nsight[5], the bottleneck was the memory access pattern when outputting data after bit-reversing the index. There were no coalesced accesses and bad data locality. There are algorithms that solves this by combining the index transpose operations with the FFT computation as in [17].

Some optimizations were attempted during the implementation phase and later abandoned. The reason why attempts was abandoned was the lack of time or no measurable improvement (or even worse performance). The use of shared memory provide fewer global memory accesses, the shared memory can be used in optimized ways such as avoiding banking conflicts[6]. This was successfully tested on the CUDA technology with no banking conflicts at all, but gave no measurable gain in performance. The relative time gained compared to global memory access was likely to small or was negated by the fact that an overhead was introduced and more shared memory had to be allocated per block.

The use of shared memory in the global kernel combined with removing most of the host-synchronization is a potentially good optimization. The time distribution during this thesis did not allow further optimizations. The attempts to implement this in short time were never completed successfully. Intuitively guessed, the use of shared memory in the global kernel would decrease global memory accesses and reduce the total number of kernel launches to $\lceil \log_2(\frac{N}{N_{block}}) \rceil$ compared to $\log_2(N) - \log_2(N_{block}) + 1$.

Wider context

Since GPU acceleration can be put to great use in large computing environments, the fastest execution time and power usage is important. If the same hardware performs faster with another technology the selection or migration is motivated by reduced power costs. Data centers and HPC is becoming a major energy consumer globally, and future development must consider all energy saving options.

6.3 Conclusions

6.3.1 Benchmark application

The FFT algorithm was successfully implemented as benchmark application in all technologies. The application provided parallelism and enough computational complexity to take advantage of the GPU. This is supported by the increase in speed compared to the CPU. The benchmark algorithm executed up to a factor

[5] A tool for debugging and profiling CUDA applications.

[6] Good access pattern allows for all threads in a warp to read in parallel, one per memory bank at a total of 32 banks on GeForce GTX 670, a banking conflict is two threads in a warp attempting to read from the same bank and becomes serialized reads.

Implementation	1D	2D
CUDA	1	1
OpenMP	×13	×15
C/C++	×18	×40

Table 6.2: *Table comparing CUDA to CPU implementations.*

of 40 times faster on the GPU as seen in table 6.2 where the CUDA implementation is compared to the CPU implementations.

6.3.2 Benchmark performance

Benchmarking on the GeForce GTX 670 graphics card performing a 2D forward-transformation resulted in the following rank:

1. CUDA
2. DirectCompute
3. OpenGL
4. OpenCL

Benchmarking on the Radeon R7 R260X graphics card performing a 2D forward-transformation resulted in the following rank:

1. DirectCompute
2. OpenGL[7]
3. OpenCL[8]

The ranking reflects the results of a combined performance relative OpenCL, where DirectCompute average at a factor of 0.8 the speed of OpenCL.

6.3.3 Implementation

The CUDA implementation had a relative advantage in terms of code size and complexity. The necessary steps to setup a kernel before executing was but a fraction of the other technologies. OpenCL needs runtime compiling of the kernel and thus require many more steps and produces a lot more setup code. Portability is in favour of OpenCL, but was not examined further (as in running the application on the CPU). The DirectCompute setup was similar to OpenCL with the addition of some more specifications required. OpenGL followed this pattern but did lack the support to select device if several was available.

The kernel code was fairly straight forward and was relatively easy to port from CUDA to any other technology. Most issues could be traced back to the memory

[7] Failed to compute sequences longer then 2^{23} elements.

[8] Performance very close to or equal to DirectCompute when sequence reached 2^{24} elements or more.

buffer handling and related back to the setup phase or how buffers needed to be declared in-code. CUDA offered the most in terms of support to a C/C++ like coding with fewer limitations than the rest.

6.4 Future work

This thesis work leave room for expanding with more test applications and improve the already implemented algorithm.

6.4.1 Application

The FFT algorithm is implemented in many practical applications, the performance tests might give different results with other algorithms. The FFT is very easy parallelized and put great demand on the memory by making large strides. It would be of interest to expand with other algorithms that puts more strain on the use of arithmetic operations.

FFT algorithms

The benchmark application is much slower than the external libraries for the GPU, the room for improvements ought to be rather large. One can not alone expect to beat a mature and optimized library such as cuFFT, but one could at least expect a smaller difference in performance in some cases. Improved or further use of shared memory and explore a precomputed twiddle factor table would be a topic to expand upon. Most important would probably be to examine how to improve the access pattern towards the global memory.

For the basic algorithm there are several options to remove some of the overhead when including the bit-reversal as a separate step by selecting an algorithm with different geometry.

Based on cuFFT that uses Cooley-Tukey and Bluestein's algorithm, a suggested extension would be to expand to other then 2^k sizes and implement to compare any size of sequence length.

6.4.2 Hardware

More technologies

The graphic cards used in this thesis are at least one generation old compared to the latest graphic cards as of late 2015. It would be interesting to see if the cards have the same differences in later generations and to see how much have been improved over the generations. It is likely that the software drivers are differently optimized towards the newer graphic cards.

The DirectX 12 API was released in the fourth quarter of 2015 but this thesis only utilized the DirectX 11 API drivers. The release of *Vulkan*, comes with the premise much like DirectX 12 of high-performance and more low-level interaction. In a similar way AMDs *Mantle* is an alternative to Direct3D with the aim of

reducing overhead. Most likely, the (new) hardware will support the newer APIs in a more optimized way during the coming years.

Graphics cards

The GeForce GTX 670 have the *Kepler* micro architecture. The model have been succeeded by booth the 700 and 900 GeForce series and the micro architecture have been followed by *Maxwell* (2014). Both Kepler and Maxwell uses 28nm design. The next micro architecture is *Pascal* and is due in 2016. Pascal will include 3D memory (High Bandwidth Memory (HBM2)) that will move onto the same package as the GPU and greatly improve memory bandwidth and size. Pascal will use a 16nm transistor design that will grant higher speed and energy efficiency.

The Radeon R7 R260X have the Graphics Core Next (GCN) 1.1 micro architecture and have been succeeded by the Radeon Rx 300 Series and GCN 1.2. The latest graphic cards in the Rx 300 series include cards with High Bandwidth Memory (HBM) and will likely be succeeded by HBM2. The Radeon R7 R260X is not target towards the high-end consumer so it would be interesting to see the performance with a high-end AMD GPU.

Intel Core i7 3770K 3.5GHz CPU

The used Intel Core i7 3770K 3.5GHz CPU have four real cores but can utilize up to eight threads in hardware. Currently the trend is to utilize more cores per die when designing new CPUs. The release of Intel Core i7-6950X and i7-6900K targeting the high-end consumer market will have 10 and 8 cores. The i7-6950X is expected some time in the second quarter in 2016.

Powerful multi-core CPUs will definitely challenge GPUs in terms of potential raw computing capability. It would make for an interesting comparison by using high-end consumer products of the newest multi-core CPUs and GPUs. This work was made with processing units from the same generation (released in 2012-2013) and the development in parallel programming have progressed and matured since.

Bibliography

[1] AMD. Accelerating Performance: The ACL clFFT Library - AMD. http://developer.amd.com/community/blog/2015/08/24/accelerating-performance-acl-clfft-library/, 2015. URL http://developer.amd.com/community/blog/2015/08/24/accelerating-performance-acl-clfft-library/. Cited on pages 20 and 44.

[2] AMD. ACL - AMD Compute Libraries - AMD. http://developer.amd.com/tools-and-sdks/opencl-zone/acl-amd-compute-libraries/, 2015. URL http://developer.amd.com/tools-and-sdks/opencl-zone/acl-amd-compute-libraries/. Cited on page 20.

[3] AMD. clFFT: OpenCL Fast Fourier Transforms (FFTs). http://clmathlibraries.github.io/clFFT/, 2015. URL http://clmathlibraries.github.io/clFFT/. Cited on pages 20 and 33.

[4] Leo I Bluestein. A linear filtering approach to the computation of discrete fourier transform. *Audio and Electroacoustics, IEEE Transactions on*, 18(4): 451–455, 1970. Cited on page 20.

[5] E Oran Brigham and RE Morrow. The fast fourier transform. *Spectrum, IEEE*, 4(12):63–70, 1967. Cited on page 4.

[6] Charilaos Christopoulos, Athanassios Skodras, and Touradj Ebrahimi. The jpeg2000 still image coding system: an overview. *Consumer Electronics, IEEE Transactions on*, 46(4):1103–1127, 2000. Cited on page 5.

[7] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965. Cited on pages 4 and 9.

[8] James W Cooley, Peter AW Lewis, and Peter D Welch. The fast fourier transform and its applications. *Education, IEEE Transactions on*, 12(1):27–34, 1969. Cited on page 4.

[9] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming. *Parallel Computing*, 38(8): 391–407, 2012. Cited on pages 14 and 44.

[10] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A comprehensive performance comparison of cuda and opencl. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216–225. IEEE, 2011. Cited on pages 14, 42, and 44.

[11] Fftw.org. FFTW Home Page. http://fftw.org/, 2015. URL `http://fftw.org/`. Cited on pages 20 and 33.

[12] Matteo Frigo. A fast fourier transform compiler. In *Acm sigplan notices*, volume 34, pages 169–180. ACM, 1999. Cited on page 20.

[13] Matteo Frigo and Steven G Johnson. Fftw: An adaptive software architecture for the fft. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, volume 3, pages 1381–1384. IEEE, 1998. Cited on page 20.

[14] Matteo Frigo and Steven G Johnson. The design and implementation of fftw3. *Proceedings of the IEEE*, 93:216–231, 2005. Cited on page 20.

[15] W Morven Gentleman and Gordon Sande. Fast fourier transforms: for fun and profit. In *Proceedings of the November 7-10, 1966, fall joint computer conference*, pages 563–578. ACM, 1966. Cited on page 3.

[16] Sayan Ghosh, Sunita Chandrasekaran, and Barbara Chapman. Energy analysis of parallel scientific kernels on multiple gpus. In *Application Accelerators in High Performance Computing (SAAHPC), 2012 Symposium on*, pages 54–63. IEEE, 2012. Cited on page 8.

[17] Naga K Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High performance discrete fourier transforms on graphics processors. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 2. IEEE Press, 2008. Cited on page 46.

[18] Michael T Heideman, Don H Johnson, and C Sidney Burrus. Gauss and the history of the fast fourier transform. *ASSP Magazine, IEEE*, 1(4):14–21, 1984. Cited on page 4.

[19] Song Huang, Shucai Xiao, and Wu-chun Feng. On the energy efficiency of graphics processing units for scientific computing. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009. Cited on page 8.

[20] Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 908–916. ACM, 2003. Cited on page 5.

[21] Nvidia. Cufft Library User ' S Guide, 2013. Cited on page 20.

[22] NVIDIA. CUDA Toolkit Documentation: cuFFT. http://docs.nvidia.com/cuda/cufft/#axzz3ufRZ1S8v, 2015. URL `http://docs.nvidia.com/cuda/cufft/`. Cited on page 33.

[23] John D Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E Lefohn, and Timothy J Purcell. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library, 2007. Cited on page 8.

[24] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008. Cited on page 7.

[25] In Kyu Park, Nitin Singhal, Man Hee Lee, Sungdae Cho, and Chris W Kim. Design and performance evaluation of image processing algorithms on gpus. *Parallel and Distributed Systems, IEEE Transactions on*, 22(1):91–104, 2011. Cited on pages 4 and 42.

[26] Ching-Lung Su, Po-Yu Chen, Chun-Chieh Lan, Long-Sheng Huang, and Kuo-Hsuan Wu. Overview and comparison of opencl and cuda technology for gpgpu. In *Circuits and Systems (APCCAS), 2012 IEEE Asia Pacific Conference on*, pages 448–451. IEEE, 2012. Cited on page 42.

[27] Stanimire Tomov, Rajib Nath, Hatem Ltaief, and Jack Dongarra. Dense linear algebra solvers for multicore with gpu accelerators. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8. IEEE, 2010. Cited on page 5.

[28] Vasily Volkov and James W Demmel. Benchmarking gpus to tune dense linear algebra. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–11. IEEE, 2008. Cited on page 5.