Institutionen för systemteknik
Department of Electrical Engineering

Examensarbete

Comparison of technologies for General-purpose computing on graphics processing units

Examensarbete utfört i Mindroad AB
vid Tekniska högskolan vid Linköpings universitet
av

Torbjörn Sörman

LiTH-ISY-EX--YY/NNNN--SE

Linköping 2015

Department of Electrical Engineering
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings tekniska högskola
Linköpings universitet
581 83 Linköping

Comparison of technologies for General-purpose computing on graphics processing units

Examensarbete utfört i Mindroad AB
vid Tekniska högskolan vid Linköpings universitet
av

Torbjörn Sörman

LiTH-ISY-EX--YY/NNNN--SE

Handledare: **Robert Forchheimer**
ISY, Linköpings universitet
Åsa Detterfelt
Mindroad AB

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

Linköping, 24 december 2015

Avdelning, Institution
Division, Department

Organisatorisk avdelning
Department of Electrical Engineering
SE-581 83 Linköping

Datum
Date

2015-12-24

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English
☐ _____

Rapporttyp
Report category

☐ Licentiatavhandling
☒ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport
☐ _____

ISBN
__

ISRN
LiTH-ISY-EX--YY/NNNN--SE

Serietitel och serienummer
Title of series, numbering

ISSN
__

URL för elektronisk version
http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-XXXXX

Titel
Title

Svensk titel
Comparison of technologies for General-purpose computing on graphics processing units

Författare
Author

Torbjörn Sörman

Sammanfattning
Abstract

If your thesis is written in English, the primary abstract would go here while the Swedish abstract would be optional.

Nyckelord
Keywords

gpgpu, gpu, benchmarking

Abstract

If your thesis is written in English, the primary abstract would go here while the Swedish abstract would be optional.

Acknowledgments

I would like to thank Åsa for the opportunity to make this Thesis work at MindRoad AB. I would also like to thank Ingemar and Robert at ISY.

Linköping, December 2015
Torbjörn Sörman

Contents

Notation **ix**

1 Introduction **1**
1.1 Background . . . 1
1.1.1 Problem statement . . . 2
1.1.2 Purpose and goal of the thesis work . . . 2
1.2 Algorithm . . . 2
1.2.1 Discrete Fourier Transform . . . 2
1.2.2 Fast Fourier Transform . . . 3
1.2.3 Image processing . . . 3
1.2.4 Image compression . . . 4
1.2.5 Linear algebra . . . 4
1.2.6 Sorting . . . 5
1.2.7 Criteria for Algorithm Selection . . . 5

2 Technologies **7**
2.1 CUDA . . . 7
2.2 OpenCL . . . 8
2.3 DirectCompute . . . 8
2.4 OpenGL Compute Shader . . . 8
2.5 OpenMP . . . 8

3 Theory **9**

4 Implementation **11**
4.1 Benchmark application GPU . . . 11
4.1.1 FFT implementation . . . 11
4.1.2 FFT 2D implementation . . . 15
4.2 Benchmark application CPU . . . 17
4.2.1 FFT in C/C++ with OpenMP . . . 17
4.2.2 2D FFT in C/C++ with OpenMP . . . 19
4.3 Benchmark configurations . . . 19
4.3.1 Limitations . . . 19

4.3.2 Testing 19

5 Avslutande kommentarer **21**

Bibliography **25**

Notation

Glossary

Term	Meaning
Kernel	A GPU program
Thread	A way for a program to split itself to many simultaneously running tasks
Block	A way to organize threads, a block is executed by a multiprocessing unit

Abbreviations

Abbreviation	Meaning
API	Application Programming Interface
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture, a technology developed by NVIDIA used to program their GPUs
DFT	Discrete Fourier Transform
DIT	Decimation In Time
DWT	Discrete Wavelet Transform
FFT	Fast Fourier Transform
GPGPU	General-Purpose computing on Graphics Processing Units
GPU	Graphics Processing Unit
OpenCL	Open Computing Language, initially developed by Apple, today the Khronos Group
SIMD	Single Instruction, Multiple Data
SIMT	Single Instruction, Multiple Threads
SM	Streaming multiprocessor, NVIDIA's GPUs contains many streaming multiprocessors and each consists of eight scalar processors

1 Introduction

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

1.1 Background

The computationally demanding problems have during a long period of time been solved faster by technical improvements in hardware. However, some limitations have been reached the last decades. Operating frequency of the CPU is no longer significantly improved. 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 its hard to further exploit simultanious CPU instructions, techniques like instruction pipelining, superscalar execution and VLIW exists but complexity and latency of hardware reduces the benefits. Related to the first is second wall, the gap between CPU speed and memory access time, that may cost several hundreds of CPU cykles if accessing primary memory. The third wall is 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 reconstruct the problem or the problem itself is already inherently parallelizable. This trend

manifests in development towards use and construction of multi-core microprocessors. The graphical processing unit (GPU) is one such device, originally exploited the inherent parallelism within visual rendering but now is available as a tool for massively parallelizable problems.

1.1.1 Problem statement

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

Depending on one's preferences, needs and future goals; selecting one technology over the other might be derived from portability or hardware requirements, programmability, how well it integrates with other frameworks or APIs or how well it's supported by the provider or developer community. Within the range of this thesis, the covered technologies are CUDA (Compute Unified Device Architecture), OpenCL (Open Computing Language), DirectCompute (API within DirectX) and OpenGL Compute Shaders.

1.1.2 Purpose and goal of the thesis work

One goal is to evaluate, select and implement an application suitable for GPGPU (General-purpose computing on graphics processing units).

Implement the same application in important technologies for GPGPU:

- CUDA
- OpenCL
- DirectCompute
- OpenGL Compute Shaders

The purpose is to compare the different technologies by means of benchmarking performance and make qualitative assessments.

1.2 Algorithm

This part cover some choices of application for a GPGPU study. The basic theory and motivation why they are suitable for benchmarking GPGPU technologies is presented. For this thesis, the Fast Fourier Transform is selected as the benchmarking application and will be more detailed in another part of the report.

1.2.1 Discrete Fourier Transform

The Fourier transform is of use when analyzing the spectrum of a continuous analog 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 analog 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's a conversion to the frequency domain. It is defined as

$$X_k = \sum_{n=0}^{N-1} x(n) W_N^{kn}, k \in [0, N-1] \tag{1.1}$$

where $W_N^{kn} = e^{-2iknN}$, commonly named the twiddle factor[4].

The DFT is used in many practical applications to perform Fourier analysis. In digital signal processing, such as discrete samples of sound waves, radio signal or any continuous signal over a finite time interval. In image processing the sampled sequence can be pixels along a row or column. The DFT takes input in complex numbers and outputs in complex coefficients. In practical applications the input is usually real numbers.

1.2.2 Fast Fourier Transform

The problem with the DFT is that it has by definition a time complexity of $\mathcal{O}(n^n)$ that makes it too slow for some applications. The Fast Fourier Transform (FFT) is one of the most common algorithm used to compute the DFT of a sequence. An FFT computes the transformation by factorizing the transformation matrix of the DFT into a product of mostly zero factors. This reduces the time complexity to $\mathcal{O}(n \log n)$. The FFT was made popular in 1965 by J.W Cooley and J.W. Tukey and it found its way into practical use at the same time and meant a serious breakthrough in digital signal processing [3, 1]. 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[5].

The algorithm is a divide and conquer algorithm that relies on recursively dividing the input into sub-blocks and eventually the problem is small enough to be solved and the sub-blocks are combined into the final result.

1.2.3 Image processing

Image processing consists of a wide range of domains. Earlier academic work with performance evaluation on the GPU[7] tested four major domains (3D shape reconstruction, feature extraction, image compression and computational photography) and compared with the CPU. Generally image processing is by nature parallel and one can expect good results on a GPU.

Most of image processing algorithms apply the same computation on a number of pixels and that is a typically 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 neighboring 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 speedup near linear to the number of computational cores used. That is a CPU with four cores can theoretically expect a near four time speedup compared to a single core. This extends to a GPU so a GPU with n cores can in ideal cases expect a speedup in the order of n. An example of this is a Gaussian blur (or smoothing) filter.

1.2.4 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 adding new features. The following sections are part of the JPEG2000 algorithm[2].

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 using Embedded Block Coding with Optimized Truncation (EBCOT) Tier-1.

The important difference between the older format JPEG compared to JPEG2000 is the use of DWT instead of Discrete Cosine Transform (DCT). In comparison to the DFT, the DCT operates solely on real values but at the same time complexity. DWT's on the other hand uses another representation that allows for a time complexity of $\mathcal{O}(N)$.

1.2.5 Linear algebra

Linear algebra is central to both pure and applied mathematics. In scientific computing it's a highly relevant problem to solve dense linear systems efficiently. From the initial uses of GPUs in scientific computing the graphics pipeline was successfully used for linear algebra through programmable vertex and pixel shaders[6]. Later on methods and systems for utilizing GPUs have been shown efficient also in hybrid system (multi-core CPUs + GPUs)[8]. 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[9].

Common operations are vector addition, scalar multiplication, dot products, linear combinations, and matrix multiplication. Matrix multiplications are of much interest since the high time complexity $\mathcal{O}(N^3)$ makes it a bottleneck in many algorithms. Matrix decomposition like LU, QR and Cholesky decomposition are used very often and are subject for benchmarking GPUs to linear algebra[9].

1.2.6 Sorting

The sort operation is an important part in computer science and have been a classic problem to work on. There exists several techniques and mostly it comes down to what problem you have and choose the best suited algorithm.

Sorting algorithms can be organized into two categories, data-driven and data-independent. The classic quicksort algorithm is probably the best known example of a data-driven sorting algorithm. It performs with time complexity On log(n) on average but have a time complexity of ON2 in the worst case. Another data-driven algorithm that does not have this problem is heap sort but instead it suffers from difficult data access patterns. Data-driven algorithms are not the easiest to parallelize since the behaviour is unknown and may cause bad load balancing.

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

Efficient sorting algorithms

Bitonic sort have been used early on in the utilization of GPUs for sorting, even though it has the time complexity of On log(n2) it's been an easy way of doing reasonably efficient sort on GPUs. Other high performance sorting on GPUs are often combinations of algorithms. Examples of fast sorting algorithms on GPUs have used bucket sort or quicksort that first splits the list into sublist and then sort in parallel with merge sort or by using bitonic sort followed by 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 digits value in the same significant position and value.

1.2.7 Criteria for Algorithm Selection

For this thesis, a benchmarking application is sought after that have the necessary complexity and relevance to 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 are 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 its still a very potent algorithm for many specific applications.

The major difficulties working with multi-core systems are applied to GPUs. What GPUs are missing compared to multi-core CPUs are the power of working in sequential, instead GPUs are excellent at fast context switching and hiding memory latencies. Most effort of working with GPUs must be to put into supply with enough parallelism, avoiding branching and refine 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 and it is sufficient to say that one can demand a reasonable performance by utilizing information sources concerning other implementations on GPUs.

2

Technologies

Five different multi-core technologies are used in this study. One is specialized in GPGPU, namely CUDA. OpenGL and DirectCompute are parts of graphic programming languages but breaks away from the graphics abstraction with *Compute Shaders*. OpenCL aims at any heterogeneous multi-core system and is used in this study to use on the GPU. To compare with the CPU, OpenMP is included as a fast and easy way to parallelize C/C++ -code.

2.1 CUDA

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

A program running 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 a function type specifier, see table 2.1, and call it from the host with launch parameters.

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

Table 2.1: *Table of function types in CUDA.*

```
// Device program (GPU)
__global__ void myKernel(float val)
{
  int tid = threadIdx.x + blockDim.x * blockIdx.x;
}

// Host program (CPU)
__host__ void cuda_fft(transform_direction dir, cpx *in, cpx *out, int n)
{
    fft_args args;
    dim3 blocks;
    int threads;
    set_block_and_threads(&blocks, &threads, CU_BLOCK_SIZE, (n >> 1));
    set_fft_arguments(&args, dir, blocks.y, CU_BLOCK_SIZE, n);
    cudaDeviceSetCacheConfig(cudaFuncCachePreferL1);
    if (blocks.y > 1) {
            while (--args.steps_left > args.steps_gpu) {
                cuda_kernel_global KERNEL_ARGS2(blocks, threads)(in, args.global_angle, 0xFFFFFFFF << args.steps_left, args.steps++, args.dist >>= 1);
            }
            ++args.steps_left;
    }
    cuda_kernel_local KERNEL_ARGS3(blocks, threads, sizeof(cpx) * args.n_per_block) (in, out, args.local_angle, args.steps_left, args.leading_bits, args.scalar);
}
```

Figure 2.1: *CUDA host code.*

2.2 OpenCL

2.3 DirectCompute

2.4 OpenGL Compute Shader

2.5 OpenMP

3
Theory

Det här är kapitlet där teorin presenteras.

4

Implementation

The FFT application has been implemented in C/C++, CUDA, OpenCL, DirectCompute and OpenGL on a GeForce GTX 670 and Radeon R7 260X graphics card and a Core i7 3770K 3.5GHz CPU.

4.1 Benchmark application GPU

4.1.1 FFT implementation

Setup

The implementation of the FFT algorithm on a GPU can be broken down into a few steps, see figure 4.1 for an 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 difference between devices and platforms are local memory capacity and thread and block configuration. Threads per block was selected for the best performance. See table 4.1 for details.

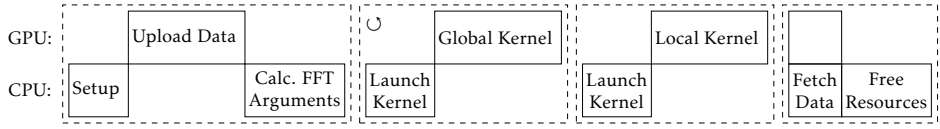

Figure 4.1: *Overview of the events in the algorithm.*

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 4.1: *Shared memory size, threads and block configuration per device.*

Butterfly

The implementation of a N-point radix-2 FFT algorithm have $\log_2 N$ stages with $N/2$ butterfly operations per stage. A butterfly operation is an addition, a subtraction, followed by a multiplication by a twiddle factor, showed in figure 4.2.

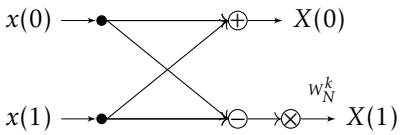

Figure 4.2: *Butterfly operations*

Thread and block scheme

The thread and block 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 4.1) and any sequences twice as long as the threads per block configuration needed the algorithm to be split over several blocks. If the sequence exceed one block then the sequence is mapped over multiple blocks using the `blockIdx.y` dimension. The block dimension `blockIdx.x` is used to calculate sequence id in when running a large batch, 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 within a block. When the sequence or partial sequence fitted within a block all data was transferred to local memory before completing the last stages. If the sequence was larger and required more then one block the synchronization was handled by launching several kernels 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

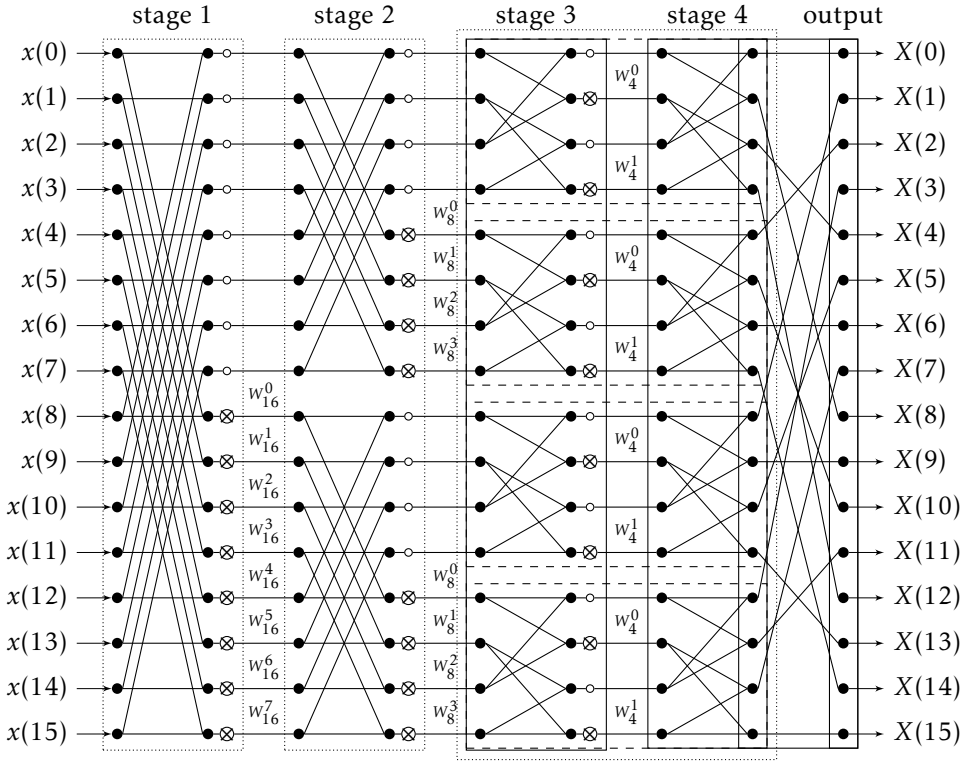

Figure 4.3: *Example flow graph of a sixteen-point FFT using (stage 1 and 2) Cooley-Tukey algorithm and (stage 3 and 4) constant geometry algorithm. The solid box is the bit-reverse order output. Dotted boxes are separate kernel launches, dashed boxes are data transfered to local memory before computing the remaining stages.*

```
int tid      = blockIdx.x * blockDim.x + threadIdx.x,
    io_low   = tid + (tid & (0xFFFFFFFF << stages_left)),
    io_high  = index1 + (N >> 1);
```

Figure 4.4: *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    = out1 + 1;
```

Figure 4.5: *CUDA example code showing index calculation for points in shared memory for the CUDA local kernel.*

for every stage). The last stage outputs from shared memory to the bit reversed index of the complete sequence. See figure 4.3 where the sequence length is sixteen and the thread 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 4.4. Input and output is done on the same index.

Index calculation for the local kernel is done once for all stages, see figure 4.5. 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.

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 reversing is available as a intrinsic integer instruction, see table 4.2. If instruction is not available figure 4.6 shows the code used. The bit reversed value had to be right shifted the number of zeroes leading the number in a 32-bit int. Example of the bit-reverse index operation: index 8 of a 16 point sequence is bit-reversed to 1, in binary its 1000 reversed to 0001. Index 8 of a 32 point sequence is bit-reversed to 2, corresponds to 01000 to 00010. Figure 4.3 show the complete bit-reverse operations of a 16-point sequence in the output step after the last stage.

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

Table 4.2: *Integer intrinsic bit-reverse function for different technologies.*

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

4.1.2 FFT 2D implementation

The FFT algorithm for two dimensional data, such as images, is first transformed row wise (each row as a separate sequence) and then a transform of each column. The implementation performs a row wise transformation and then transposes the whole image twice, see figure 4.7. A transformed image is shown in figure 4.8.

```
set_transpose_dimensions(&blocks, &threads, CU_TILE_DIM, CU_BLOCK_DIM, n);
cuda_fft_2d_helper(dir, dev_in, dev_out, n);
cuda_transpose_kernel <<<blocks, threads>>> (dev_out, dev_in, n);
cuda_fft_2d_helper(dir, dev_in, dev_out, n);
cuda_transpose_kernel <<<blocks, threads>>> (dev_out, dev_in, n);
swap_buffer(&dev_in, &dev_out);
```

Figure 4.7: *CUDA host code for the 2D FFT algorithm.*

The difference between the FFT kernel for 1D and 2D are the indexing scheme. 2D are indexed as rows at `blockIdx.x` and columns at `threadIdx.x+blockIdx.y·bloc` For 2D `blockIdx.z` is used as the sequence id in a batch.

Transpose

The transpose kernel uses a different index mapping of the 2D-data and blocks/threads then 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 4.1 for specific size per technology. To avoid banking issues, the last dimension is increased with one but not used. However, resolving

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

Figure 4.8: *Original image 4.8a transformed and represented with a quadrant shifted magnitude visualization 4.8b.*

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 tiles rows and columns are diveded over the `threadIdx.x` and `threadIdx.y` index respectively. See figure 4.9 for code example. Example: The CUDA shared memory can allocate 49152 bytes and a single data point require `sizeof(float)` $\cdot 2 = 8$ bytes. That leaves room for a tile size of $64 \cdot (64 + 1) \cdot 8 = 33280$ bytes.

```
__global__ void cuda_transpose_kernel(cpx *in, cpx *out, int n)
{
    __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 + threadIdx.x;
    int y = blockIdx.y * CU_TILE_DIM + threadIdx.y;
    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[threadIdx.y + j][threadIdx.x + i] = in[(y + j) * n + (x + i)];

    __syncthreads();
    // Write to global (out) from shared memory (tile)
    x = blockIdx.y * CU_TILE_DIM + threadIdx.x;
    y = blockIdx.x * CU_TILE_DIM + threadIdx.y;
    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[threadIdx.x + i][threadIdx.y + j];
}
```

Figure 4.9: *CUDA device code for the transpose kernel.*

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 and writing to the output memory is performed in close stride. Figure 4.10 shows how the transpose is performed in memory.

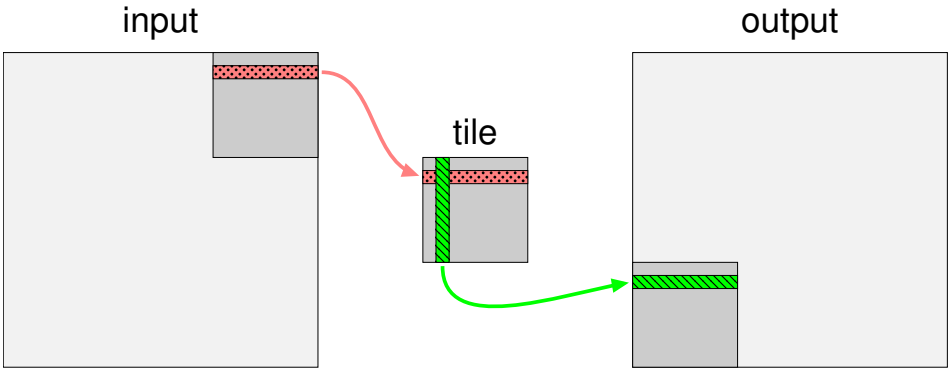

Figure 4.10: *Illustration of how shared memory is used in transposing an image. Input data is tiled and each tile is written to shared memory and transposed before written to the output memory.*

4.2 Benchmark application CPU

4.2.1 FFT in C/C++ 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 4.11 for an overview.

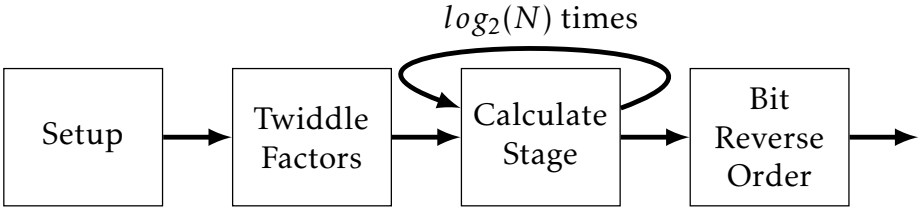

Figure 4.11: *OpenMP implementation overview transforming sequence of size N.*

Twiddle factors

The twiddle factor are stored for each butterfly operation. To save time, only the real part are calculated and the imaginary part is retrieved from the real parts due to the fact that $\sin(x) = \cos(\pi/2 + x)$ and $\sin(\pi/2 + x) = -\cos(x)$ to store. See figure 4.3 for an example. The calculations will be split among the threads by static scheduling in two steps, first calculate the real values, secondly copy from real to imaginary.

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

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

Bit Reversed Order

See figure 4.12 for code showing the bit reverse ordering operation in C/C++ code.

```
static void __inline 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 4.12: *C/C++ code performing the bit reverse ordering of a N-point sequence.*

4.2.2 2D FFT in C/C++ with OpenMP

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

4.3 Benchmark configurations

4.3.1 Limitations

All implementations are limited to handle sequences of 2^n length or $2^m \times 2^m$ where n and m are integers. The GPUs have a maximum of 2GB global memory available and the upper limit of 2D transformations are $m = 8192$ since the implementation uses two buffers and require $8192 \cdot 8192 \cdot$ `sizeof(float2)` $\cdot 2 = 1073741824$ bytes, however the Radeon R7 260X card does not handle that size well and is limited to $m = 4096$ (and OpenGL is even limited to $m = 2048$).

4.3.2 Testing

All tests executed on the GPU utilize some form of event timestamps supplied by the technology used. See table 4.4 for specific method per technology.

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

Table 4.4: *Method to extract timestamps for kernel events for the different technologies.*

5

Avslutande kommentarer

Sätt av ett kort kapitel sist i rapporten till att avrunda och föreslå rikningar för framtida utveckling av arbetet.

Appendix

Bibliography

[1] E. O. Brigham and R. E. Morrow. The fast Fourier transform. *Spectrum, IEEE*, 4(12):63 –70, 1967. ISSN 0018-9235. doi: 10.1109/MSPEC.1967.5217220. URL http://ieeexplore.ieee.org/ielx5/6/5217195/05217220.pdf?tp={&}arnumber=5217220{&}isnumber=5217195$\delimiter"026E30F$nhttp://ieeexplore.ieee.org/stamp/stamp.jsp?tp={&}arnumber=5217220. Cited on page 3.

[2] Charilaos Christopoulos, Athanassios Skodras, Touradj Ebrahimi, and Corporate Unit. The {JPEG2000} Still Image Coding Systems: An Overview. *IEEE Trans. Consumer Electronics*, 46(4):1103–1127, 2000. Cited on page 4.

[3] James W. Cooley, Peter a. W. Lewis, and Peter D. Welch. The Fast Fourier Transform and Its Applications. *IEEE Transactions on Education*, 12(1), 1969. ISSN 0018-9359. doi: 10.1109/TE.1969.4320436. Cited on page 3.

[4] W M Gentleman and G Sande. Fast Fourier Transforms: for fun and profit. *Proceedings of the November 7-10, 1966, fall joint computer conference*, pages 563–578, 1966. URL papers2://publication/uuid/7065C1C0-089B-4DA8-8524-D5B62CB2B37A. Cited on page 3.

[5] Michael T. Heideman, Don H. Johnson, and C. Sidney Burrus. Gauss and the history of the fast Fourier transform. *Archive for History of Exact Sciences*, 34(3):265–277, 1985. ISSN 00039519. doi: 10.1007/BF00348431. Cited on page 3.

[6] Jens Krüger and Rüdiger Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics*, 22(3): 908, 2003. ISSN 07300301. doi: 10.1145/882262.882363. Cited on page 4.

[7] Ik Park, Nitin Singhal, Mh Lee, Sangdae Cho, and Cw Kim. Design and performance evaluation of image processing algorithms on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):91–104, 2011. Cited on page 3.

[8] Stanimire Tomov, Rajib Nath, Hatem Ltaief, and Jack Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. *Proceedings of the*

2010 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum, IPDPSW 2010, 2010. ISSN 1878-3449. doi: 10.1109/IPDPSW.2010.5470941. Cited on page 4.

[9] Vasily Volkov, James Demmel, and U C Berkeley. Benchmarking g GPUs to Tune Dense Linear Algebra. (November), 2008. Cited on page 4.