Internal Report 2013-05 May 2013

Universiteit Leiden
Opleiding Informatica

On GPU Fourier Transformations

for

a Thesis

Giso Dal

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

ON GPU FOURIER TRANSFORMATIONS

by

GISO DAL, BSc.

THESIS

Presented to the Faculty of the Graduate School of

Leiden University

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Leiden Institute of Advanced Computer Science

LEIDEN UNIVERSITY

March 28, 2013

Abstract

The Fourier Transform is one of the most influential mathematical equations of our time. The Discrete Fourier Transform (DFT) (which is equal to the Fourier Transform for signals with equally spaced samples) has been improved by a more efficient algorithm called the Fast Fourier Transform contributed by Cooley-Tukey[8] and Gentlemen-Sande[11]. Improvements since then have primarily been focused on hardware correspondence.

The terms memory wall and power wall restrict CPUs from increasing their performance the way they have in the past. CPUs can only increase the number of cores, following the example of graphics cards and increasing parallelization. Developments like CUDA have made it possible to program GPUs the way one would a CPU. Publications on optimization of FFTs on GPUs have primarily been focused on overlapping data transfer and computation, and multi-dimensional FFTs. As a multi-dimensional FFT consists for 1-dimensional FFTs along each dimension, we focus on optimizing the 1-dimensional case and propose a method to run the FFT algorithm on multiple GPUs having minimized communication.

Table of Contents

Chapter 1

Introduction

The Discrete Fourier Transform (DFT) in the field of signal processing is the equivalent of the continuous Fourier Transform for signals of which samples are separated by equal intervals. The Fast Fourier Transform (FFT) proposed by Cooley-Tukey[8] is a method to calculate the DFT reducing the computational complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$. Their method would later become known as *decimation-in-time* and the FFT proposed by Gentlemen-Sande[11] would become known as *decimation-in-frequency*. Increased performance since then would primarily rely on the exploitation of hardware architectures.

Developments like Compute Unified Device Architecture (CUDA) by NVIDIA has brought graphics cards in reach of developers and researchers. It is a parallel computation platform and programming model. Using this model, algorithms formally developed for the CPU can be redesigned for GPUs. The FFT is not a textbook applicant for GPU optimization due to its data dependencies throughout the algorithm, revealing it to be less parallel in nature than desirable. The GPU can however accommodate the needs of the FFT with its unique memory architecture outperforming CPUs with the use of many threads and high bandwidth.

Work has been done on the overlapping of data transfer and computation for 3-dimensional FFTs[21, 27, 13], we show how this can be done for multi-GPU FFTs with the aid of the proposed *distributed butterflies*. A more straightforward approach motivated by NVIDIA's best practice guide[2] are the memory usage optimizations proposed by [12]. Yet these contributions should be standard by now. Multi-dimensional FFTs seem to be a hard topic

because the memory architecture of the GPU does not perform well when accessing data along the rows of different dimensions. [26, 17, 16, 7, 12] tackle this issue by optimizing memory locality through bank-conflict reduction and transposing the data as the calculations progress along each dimension. Publications for multi-GPU implementations of the FFT are scarce, but [7] proposes a method to calculate a 3-dimensional FFT with a complex web of CUDA, PThread, MPI en Infiniband IB/verbs.

We contribute to this field of research the following:

- We discuss a CUDA FFT implementation from scratch. In order to develop an efficient FFT implementation we must first dive into the architecture and rely on GPU characteristic information as revealed by [24, 15] and NVIDIA[2].
- We optimize the FFT by using a higher radix, requiring less global memory accesses.
- Most work has been done on multi-dimensional FFTs. We propose a method of calculating a large n-dimensional FFT on multiple GPUs without synchronization nor data sharing during the algorithm.
- Computation is moved to faster memories as much as possible, including the L2 cache which to the best of our knowledge has not been done yet.

We start with the mathematical background of the Discrete Fourier Transform and the Fast Fourier Transform in Chapter 2. Chapter 3 consists of two major sections. Section 3.1 will discuss the hardware architecture *Fermi*[2, 3, 4] and how the CUDA framework maps to this architecture. Section 3.2 discusses a FFT implementation in great detail and how it is influenced by the Fermi architecture. We finish with results and comparisons in Chapter 4.

Chapter 2

The Fourier Transform

The Fourier transform has many applications. In general, it tries to separate noise from repeating cyclic patterns in some given data x. We speak of *Fourier frequency* f, if x contains a cyclic pattern which repeats f times. In other words, it transforms a function of time to a function of frequency. As the Fourier transform is reversible, we can always return to the original function.

We start by describing the Discrete Fourier Transform in Section 2.1, in which we also give some required mathematical background on complex numbers and trigonometry to understand the inner workings of Fourier transform algorithms. The Fast Fourier Transform will be described in Section 2.2, including some known variants and a mathematical explanation through an example. Finally, Section 2.3 will show how to perform and multi-dimensional Fourier Transform.

2.1 Discrete Fourier Transform

The Discrete Fourier Transform (DFT) is an invertible, linear transformation of time complexity $\mathcal{O}(n^2)$[25].

$$\mathcal{F} : \mathbb{C}^n \longrightarrow \mathbb{C}^n,$$

where $\mathbb{C}$ is the set of complex numbers.

Given data set $x = \{x_0, x_1, \ldots, x_{N-1}\}$, with $N = |x|$. With $X = \{X_0, X_1, \ldots, X_r, \ldots, X_{N-1}\}$

the DFT of x is denoted $X = \mathrm{DFT}(x)$. Equation 2.1 shows the mathematical representation of the DFT and Equation 2.2 its inverse.

$$X_r = \sum_{l=0}^{N-1} x_\ell \cdot \omega_N^{r\ell}, \tag{2.1}$$

$$x_n = \frac{1}{N} \sum_{r=0}^{N-1} X_r \cdot \omega_N^{r\ell}, \tag{2.2}$$

with $r = \{0, \ldots, N-1\}$, $\omega_N^{r\ell} = e^{-i2\pi \frac{r\ell}{N}}$, e the base of natural logarithm and $i = \sqrt{-1}$. Complex numbers and trigonometry play an important part in these equations. The following sections discuss the mathematical background to give an understanding of how to implement the DFT algorithm.

2.1.1 Complex Numbers

Let $\mathbb{R}$ be the set of real numbers and $\mathbb{C}$ be the set of complex numbers. Complex numbers were introduced to solve polynomial equations having no solution in $\mathbb{R}$. $\mathbb{C}$ contains numbers of the form $a + ib$, where $a, b \in \mathbb{R}$ and $i = \sqrt{-1}$. Operators have to be defined to be able to do calculations with these numbers. For any $\phi = a + ib \in \mathbb{C}$, let ϕ_r refer to its real part a and let ϕ_i refer to its imaginary part b. Let $\phi, \psi \in \mathbb{C}$, equations 2.3 to 2.8 define complex inverse, multiplicative inverse, addition, subtraction, division and multiplication, respectively.

$$\overline{\phi} = -\phi_r + -i\phi_i. \tag{2.3}$$

$$\phi^{-1} = \frac{\phi_r - i\phi_i}{\phi_r^2 + \phi_i^2}. \tag{2.4}$$

$$\phi + \psi = (\phi_r + \psi_r) + i(\phi_i + \psi_i). \tag{2.5}$$

$$\phi - \psi = (\phi_r - \psi_i) + i(\phi_i - \psi_i). \tag{2.6}$$

$$\frac{\phi}{\psi} = \frac{(\phi_r \cdot \psi_r + \phi_i \cdot \psi_i) + (\phi_i \cdot \psi_r - \phi_r \cdot \psi_i)}{\psi_r^2 + \psi_i^2}. \tag{2.7}$$

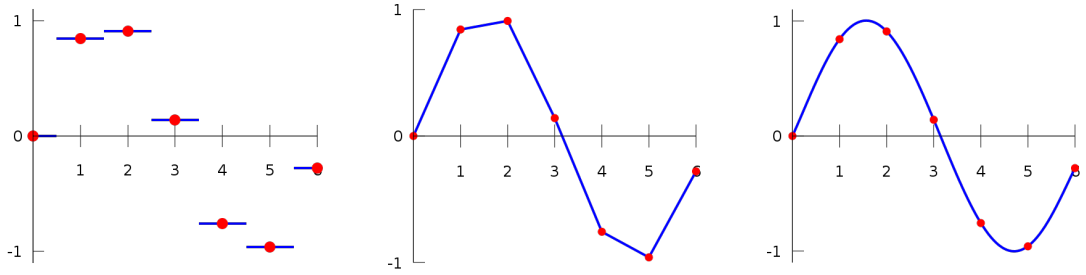
$$\phi \cdot \psi = (\phi_r \cdot \psi_i - \phi_i \cdot \psi_r) + i(\phi_r \cdot \psi_i + \phi_i \cdot \psi_r). \tag{2.8}$$

It will become clear that Fourier transform algorithms only use complex multiplication, addition and subtraction.

2.1.2 Trigonometric Interpolation

Interpolation is a technique used where data points are mapped to points within the range of the given data. Take *constant interpolation* for example, where data points within the given data and a certain range get the same value. Figure 2.1 shows examples for *constant*, *linear* and *polynomial interpolation*.

Figure 2.1 Interpolation examples



(a) Constant Interpolation **(b)** Linear Interpolation **(c)** Polynomial Interpolation

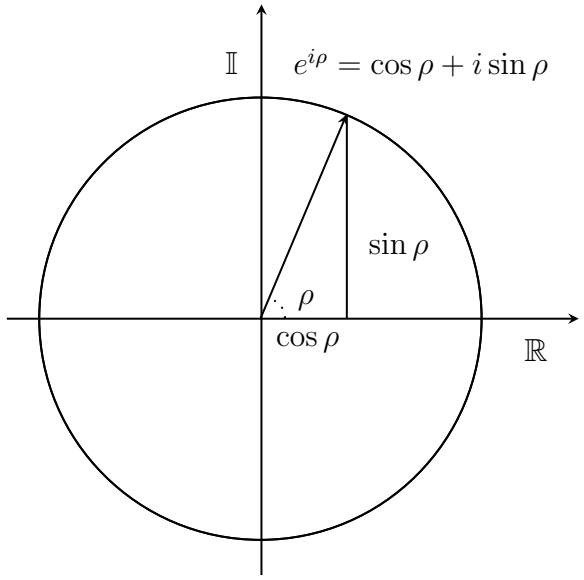
It should now be clear what the purpose of interpolation is. *Trigonometric interpolation*[20] refers to polynomial interpolation of functions consisting of the sum of cosines and sines for a given period. It is suited for highly periodic functions. An example of a trigonometric interpolation method is the DFT.

The Fourier transform algorithms map to an Imaginary domain. To show the relation between the imaginary points or *complex numbers* and to visualize them, we need to use *Euler's formula*[18], which establishes the relationship between the trigonometric functions and the complex exponential function. Euler's formula states that, for any real number ρ:

$$e^{i\rho} = \cos\rho + i\sin\rho, \tag{2.9}$$

where e is the base of the natural logarithm, $i = \sqrt{-1}$, and cos and sin are the trigonometric functions cosine and sine respectively, with the argument ρ given in radians. Figure 2.2 shows an example of how to plot a complex number. It consists of real axis $\mathbb{R}$ and an imaginary axis $\mathbb{I}$.

Figure 2.2 Visualizing complex numbers



Let ϕ be a complex number. The relationship between trigonometric functions and complex numbers can be shown as follows:

$$e^{\phi} = e^{\phi_r + i\phi_i} = e^{\phi_r} e^{i\phi_i}, \text{ and } e^{i\phi_i} = \cos\phi_i + i\sin\phi_i \tag{2.10}$$

Thus, $\mathbb{R}(e^{\phi}) = e^{\phi_r} + \cos\phi_i$ and $\mathbb{I}(e^{\phi}) = e^{\phi_r}\sin\phi_i$. This will be used to partially solve Equation 2.1 in the following section.

2.1.3 Twiddle Factors

The ω values in Equation 2.1 and 2.2 are also known as *twiddle factors*[11]. The term was proposed by Gentleman-Sande in 1966 to describe the trigonometric constant coefficients used in Fourier transform algorithms. It became widely accepted and has been used in

many papers since then.

The twiddle factors can be rewritten into a trigonometric form with the aid of Euler's formula (Equation 2.9):

$$\omega_N^{r\ell} = e^{-i2\pi\frac{r\ell}{N}} = \cos -2\pi\frac{r\ell}{N} + i\sin -2\pi\frac{r\ell}{N}. \tag{2.11}$$

In Section 2.1.4 we use Equation 2.11 to create a function calculating the twiddle factors. Trigonometric function calls are expensive and can be reduced due to the periodic nature the twiddle factors. Twiddle factors are symmetric and are equally spaced when visualizing them like in Figure 2.2. This shows through trigonometric identities that if $a+ib$ is a twiddle factor, then $\pm a \pm ib$ and $\pm b \pm ia$ are also twiddle factors. In addition, the following rules hold for any N:

$$\omega_N^0 = 1 + i0, \tag{2.12}$$

$$\omega_N^{r\ell} = \omega_N^{r\ell \pm \alpha N} = \omega_N^{r\ell \bmod N}, \tag{2.13}$$

where $\alpha \in \mathbb{N}$ and $\alpha \geq 0$. This information was utilized in an algorithm proposed by Singleton[22] for certain N. Obviously, the outcome of a Fourier transform also has symmetry. Thus we can reduce the number of operations given this information.

2.1.4 The Procedure

Algorithm 2.1 shows the translation from the mathematical representation of the DFT (Equation 2.1) to pseudo code.

Algorithm 2.1 Discrete Fourier Transform

```
▷ x is the input array

function TWIDDLE(rℓ,N)
    c ← −2π rℓ/N
    return cos(c) + i sin(c)
end function

procedure DFT(x)
    N ← |x|
    for r ← 0 to N do
        X_r ← 0
        for ℓ ← 0 to N do
            X_r ← X_r + x_ℓ ∗ TWIDDLE(rℓ, N)
            ℓ ← ℓ + 1
        end for
        r ← r + 1
    end for
    return X
end procedure
```

2.2 Fast Fourier Transform

The Fast Fourier Transform (FFT) takes advantage of the symmetric property of twiddle factors. Thereby, the number of calculation required are reduced and this results in a computational complexity of $\mathcal{O}(n \log n)$[14].

The FFT variant proposed by Cooley and Tukey[8] is most commonly used. It employs a divide and conquer strategy, which partitions a DFT into smaller and smaller DFTs. This algorithm will be used to explain the FFT. The term *radix* is used to indicate in how many partitions the DFT is split at each decimation step. Take a radix-2 algorithm for example. Given data set of size $N = 8$, it will be split in 2, creating 2 DFTs of size N / 2 = 4. We decimate again and get 4 DFTs of size 2. This process continues until all DFTs are of size 1. It implies that a radix-2 Cooley-Tukey can only work with 2^n data sizes, and radix-r can only work with r^n data sizes. There are two main approaches to calculate the FFT, called *decimation-in-time* (DIT) and *decimation-in-frequency* (DIF). We discuss these in Section 2.2.1 and 2.2.2 including mathematical derivations. We then show how to handle the output produced by the FFT in Section 2.2.3 and show some known variants of the FFT in Section 2.2.5.

2.2.1 Decimation-in-time

Given data set $x = \{x_0, x_1, \ldots, x_{N-1}\}$, with $N = |x|$. $X = \{X_0, X_1, \ldots, X_r, \ldots, X_{N-1}\}$ gives the Fast Fourier Transform of x denoted by $X = FFT(x)$. As an example we will use a radix-P algorithm with an input data set of size $N = 12$ and calculate the FFT by P sub-DFTs (X^0, X^1, X^2) of size $\frac{N}{P}$, with $P = 3$. Note that the size of the input to a traditional radix-3 algorithm must be 3^n.

Decimation-in-time and and decimation-in-frequency are mirrored versions of each other. The difference between DIT and DIF can be seen in the derivation from the general DFT

formula (Equation 2.1). Remember that we want three partitions:

$$\begin{aligned}
X_r &= \sum_{\ell=0}^{N-1} x_\ell \omega_N^{r\ell} \\
&= \sum_{k=0}^{\frac{N}{3}-1} x_{3k+0}\omega_N^{r(3k+0)} + \sum_{k=0}^{\frac{N}{3}-1} x_{3k+1}\omega_N^{r(3k+1)} + \sum_{k=0}^{\frac{N}{3}-1} x_{3k+2}\omega_N^{r(3k+2)} \\
&= \omega_N^{0r}\sum_{k=0}^{\frac{N}{3}-1} x_{3k+0}\omega_N^{r(3k)} + \omega_N^{1r}\sum_{k=0}^{\frac{N}{3}-1} x_{3k+1}\omega_N^{r(3k)} + \omega_N^{2r}\sum_{k=0}^{\frac{N}{3}-1} x_{3k+2}\omega_N^{r(3k)} \\
&= \omega_N^{0r}\sum_{k=0}^{\frac{N}{3}-1} x_{3k+0}\omega_{\frac{N}{3}}^{rk} + \omega_N^{1r}\sum_{k=0}^{\frac{N}{3}-1} x_{3k+1}\omega_{\frac{N}{3}}^{rk} + \omega_N^{2r}\sum_{k=0}^{\frac{N}{3}-1} x_{3k+2}\omega_{\frac{N}{3}}^{rk},
\end{aligned}$$

with $r = \{0, 1, \ldots, N-1\}$. We can now isolate our three sub-DFTs.

$$\begin{aligned}
X_r^0 &= \sum_{k=0}^{\frac{N}{3}-1} x_{3k+0}\omega_{\frac{N}{3}}^{rk} \\
X_r^1 &= \sum_{k=0}^{\frac{N}{3}-1} x_{3k+1}\omega_{\frac{N}{3}}^{rk} \\
X_r^2 &= \sum_{k=0}^{\frac{N}{3}-1} x_{3k+2}\omega_{\frac{N}{3}}^{rk},
\end{aligned}$$

with $r = \{0, 1, \ldots, \frac{N}{3}-1\}$. Note that some twiddle factors are missing. They will participate in the calculation at a later stage we refer to as the *combination step*. We can first calculate the sub-DFTs and *combine* them afterward with the missing twiddle factors. From the previous derivation we can conclude that the input to each sub-DFT is determined by:

$$\begin{aligned}
X_r^0 &\leftarrow x_{3r+0}, \\
X_r^1 &\leftarrow x_{3r+1}, \\
X_r^2 &\leftarrow x_{3r+2},
\end{aligned}$$

with $r = \{0, 1, \ldots, N/3-1\}$. We can now choose to solve the sub-DFTs with the DFT algorithm described in Section 2.1 or follow the Cooley-Tukey algorithm, and continue

decimating. The size of each sub-DFT is now $\frac{N}{3} = 4$, which means we cannot continue radix-3, because 4 is not dividable by 3. We can however continue radix-2, taking two more steps, or radix-4, taking on more step. This technique is called *mixed-radix*, on the count of using different radixes consecutively to decimate the DFT. More FFT variants will be discussed in Section 2.2.5. Also note that if input size is N, then the FFT radix-N would be the original DFT. Equation 2.14 is a generalization for the isolation into sub-DFTs for any P.

$$X_r^m = \sum_{k=0}^{\frac{N}{P}-1} x_{Pk+m}\omega_{\frac{N}{P}}^{rk}, \tag{2.14}$$

with $m = \{0, 1, \ldots, P-1\}$.

Lets look at the combination step. Due to the periodic phase of each sub-DFT, we can conclude that $X_r^m \equiv X_{r+\frac{N}{3}}^m \equiv X_{r+\frac{2N}{3}}^m$. This means that the output is:

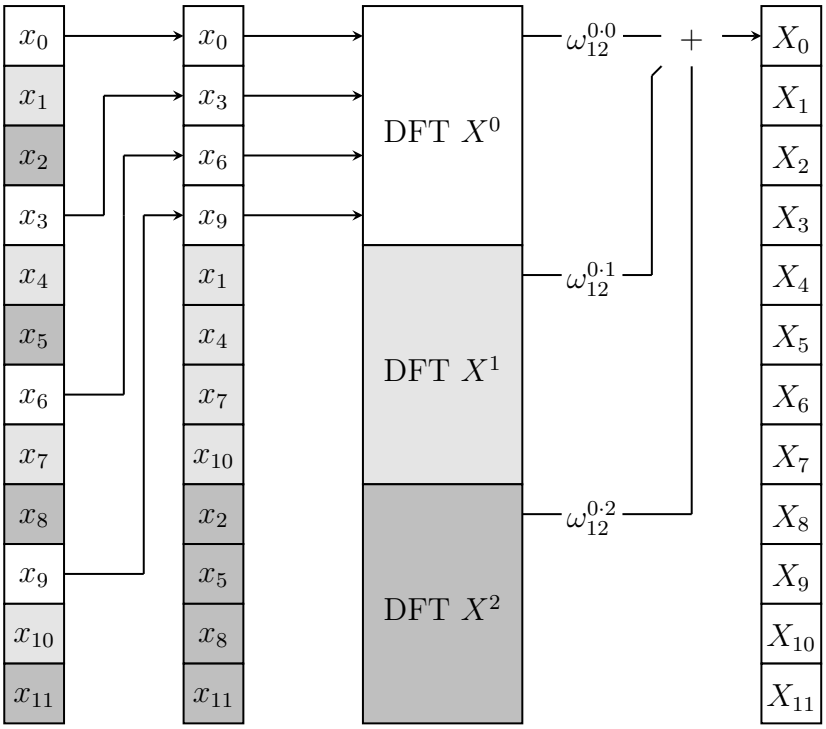
$$\begin{aligned}
X_{r+\frac{0N}{3}} &= \omega_N^{0(r+\frac{0N}{3})} X_r^0 + \omega_N^{1(r+\frac{0N}{3})} X_r^1 + \omega_N^{2(r+\frac{0N}{3})} X_r^2, \\
X_{r+\frac{1N}{3}} &= \omega_N^{0(r+\frac{1N}{3})} X_r^0 + \omega_N^{1(r+\frac{1N}{3})} X_r^1 + \omega_N^{2(r+\frac{1N}{3})} X_r^2, \\
X_{r+\frac{2N}{3}} &= \omega_N^{0(r+\frac{2N}{3})} X_r^0 + \omega_N^{1(r+\frac{2N}{3})} X_r^1 + \omega_N^{2(r+\frac{2N}{3})} X_r^2,
\end{aligned}$$

with $r = \{0, 1, \ldots, \frac{N}{3} - 1\}$. A generalized version of these formulas for P partitions is:

$$X_{r+\frac{mN}{P}} = \sum_{l=0}^{P-1} \left(\sum_{k=0}^{\frac{N}{P}-1} x_{Pk+l}\omega_{\frac{N}{P}}^{rk} \right) \omega_N^{rl}, \tag{2.15}$$

with $m = \{0, 1, \ldots, P-1\}$. Figure 2.3 shows the DIT approach partially for a radix-3 algorithm, how the input data is moved to the respective sub-DFT including the combination step. The structure is recursive in nature, meaning that inside the DFT node the same structure arises. Whenever twiddle factors are shown, it means that the data value must be multiplied with it.

Figure 2.3 DIT Fourier Transform



2.2.2 Decimation-in-frequency

As mentioned before, decimation-in-frequency is the mirrored version of decimation-in-time. It does the combination step at the beginning. Again we start from Equation 2.1 and use $P = 3$ to derive the first step with the DIF approach:

$$\begin{aligned}
X_r &= \sum_{\ell=0}^{N-1} x_l \omega_N^{r\ell} \\
&= \sum_{k=\frac{0N}{3}}^{\frac{1N}{3}-1} x_k \omega_N^{rk} + \sum_{k=\frac{1N}{3}}^{\frac{2N}{3}-1} x_k \omega_N^{rk} + \sum_{k=\frac{2N}{3}}^{\frac{3N}{3}-1} x_k \omega_N^{rk} \\
&= \sum_{k=0}^{\frac{N}{3}-1} x_{k+\frac{0N}{3}} \omega_N^{r(k+\frac{0N}{3})} + \sum_{k=0}^{\frac{N}{3}-1} x_{k+\frac{1N}{3}} \omega_N^{r(k+\frac{1N}{3})} + \sum_{k=0}^{\frac{N}{3}-1} x_{k+\frac{2N}{3}} \omega_N^{r(k+\frac{2N}{3})} \\
&= \omega_N^{\frac{0N}{3}r} \sum_{k=0}^{\frac{N}{3}-1} x_{k+\frac{0N}{3}} \omega_N^{rk} + \omega_N^{\frac{1N}{3}r} \sum_{k=0}^{\frac{N}{3}-1} x_{k+\frac{1N}{3}} \omega_N^{rk} + \omega_N^{\frac{1N}{3}r} \sum_{k=0}^{\frac{N}{3}-1} x_{k+\frac{2N}{3}} \omega_N^{rk} \\
&= \omega_3^{0r} \sum_{k=0}^{\frac{N}{3}-1} x_{k+\frac{0N}{3}} \omega_N^{rk} + \omega_3^{1r} \sum_{k=0}^{\frac{N}{3}-1} x_{k+\frac{1N}{3}} \omega_N^{rk} + \omega_3^{2r} \sum_{k=0}^{\frac{N}{3}-1} x_{k+\frac{2N}{3}} \omega_N^{rk} \\
&= \sum_{k=0}^{\frac{N}{3}-1} \left(x_{k+\frac{0N}{3}} \omega_3^{0r} + x_{k+\frac{1N}{3}} \omega_3^{1r} + x_{k+\frac{2N}{3}} \omega_3^{2r} \right) \omega_N^{rk},
\end{aligned}$$

with $r = \{0, 1, \ldots, N-1\}$. Now instead of interleaved input, we have consecutive input, but the output is interleaved. The input is determined by:

$$X_r^m \leftarrow \{x_{\frac{0N}{3}+r}, x_{\frac{1N}{3}+r}, x_{\frac{2N}{3}+r}\},$$

with $m = \{0, 1, \ldots, 3\}$ and $r = \{0, 1, \ldots, \frac{N}{3} - 1\}$. The output thus is:

$$\begin{aligned}
X_{3r+0} &= \sum_{k=0}^{\frac{N}{3}-1} \left(x_{k+\frac{0N}{3}} \omega_3^{0(3r+0)} + x_{k+\frac{1N}{3}} \omega_3^{1(3r+0)} + x_{k+\frac{2N}{3}} \omega_3^{2(3r+0)} \right) \omega_N^{(3r+0)k} \\
&= \sum_{k=0}^{\frac{N}{3}-1} \left(x_{k+\frac{0N}{3}} \omega_3^{0(3r+0)} + x_{k+\frac{1N}{3}} \omega_3^{1(3r+0)} + x_{k+\frac{2N}{3}} \omega_3^{2(3r+0)} \right) \omega_N^{0k} \omega_N^{3rk} \\
&= \sum_{k=0}^{\frac{N}{3}-1} \left(x_{k+\frac{0N}{3}} \omega_3^{0(3r+0)} + x_{k+\frac{1N}{3}} \omega_3^{1(3r+0)} + x_{k+\frac{2N}{3}} \omega_3^{2(3r+0)} \right) \omega_N^{0k} \omega_3^{rk} \\
X_{3r+1} &= \sum_{k=0}^{\frac{N}{3}-1} \left(x_{k+\frac{0N}{3}} \omega_3^{0(3r+1)} + x_{k+\frac{1N}{3}} \omega_3^{1(3r+1)} + x_{k+\frac{2N}{3}} \omega_3^{2(3r+1)} \right) \omega_N^{(3r+1)k} \\
&= \sum_{k=0}^{\frac{N}{3}-1} \left(x_{k+\frac{0N}{3}} \omega_3^{0(3r+1)} + x_{k+\frac{1N}{3}} \omega_3^{1(3r+1)} + x_{k+\frac{2N}{3}} \omega_3^{2(3r+1)} \right) \omega_N^{1k} \omega_N^{3rk} \\
&= \sum_{k=0}^{\frac{N}{3}-1} \left(x_{k+\frac{0N}{3}} \omega_3^{0(3r+1)} + x_{k+\frac{1N}{3}} \omega_3^{1(3r+1)} + x_{k+\frac{2N}{3}} \omega_3^{2(3r+1)} \right) \omega_N^{1k} \omega_3^{rk} \\
X_{3r+2} &= \sum_{k=0}^{\frac{N}{3}-1} \left(x_{k+\frac{0N}{3}} \omega_3^{0(3r+2)} + x_{k+\frac{1N}{3}} \omega_3^{1(3r+2)} + x_{k+\frac{2N}{3}} \omega_3^{2(3r+2)} \right) \omega_N^{(3r+2)k} \\
&= \sum_{k=0}^{\frac{N}{3}-1} \left(x_{k+\frac{0N}{3}} \omega_3^{0(3r+2)} + x_{k+\frac{1N}{3}} \omega_3^{1(3r+2)} + x_{k+\frac{2N}{3}} \omega_3^{2(3r+2)} \right) \omega_N^{2k} \omega_N^{3rk} \\
&= \sum_{k=0}^{\frac{N}{3}-1} \left(x_{k+\frac{0N}{3}} \omega_3^{0(3r+2)} + x_{k+\frac{1N}{3}} \omega_3^{1(3r+2)} + x_{k+\frac{2N}{3}} \omega_3^{2(3r+2)} \right) \omega_N^{2k} \omega_3^{rk}
\end{aligned}$$

with $r = \{0, 1, \ldots, \frac{N}{3} - 1\}$. The outcome of one step of this equation is the input to each sub-DFT. A generalization for the input of P sub-DFTs is:

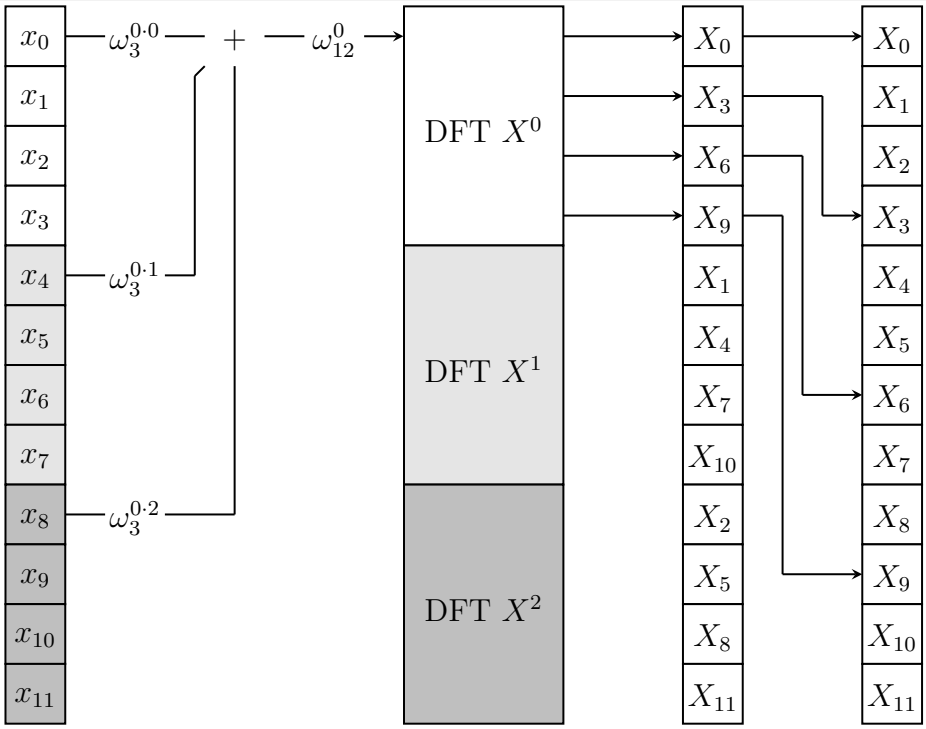
$$X_r^m \leftarrow \left(\sum_l^{P-1} x_{r+\frac{lN}{P}} \omega_P^{l(Pr+m)} \right) \omega_N^{ml}, \tag{2.16}$$

with $m = \{0, 1, \ldots, P-1\}$ and $r = \{0, 1, \ldots, \frac{N}{P} - 1\}$. When looking at the example of three partitions and at the data needed by X^0 we see:

$$
\begin{aligned}
X_0^0 &\leftarrow \{x_{\frac{0N}{3}+0}, x_{\frac{1N}{3}+0}, x_{\frac{2N}{3}+0}\} \\
X_1^0 &\leftarrow \{x_{\frac{0N}{3}+1}, x_{\frac{1N}{3}+1}, x_{\frac{2N}{3}+1}\} \\
X_{\dots}^0 &\leftarrow \dots \\
X_{P-1}^0 &\leftarrow \{x_{\frac{0N}{3}+P-1}, x_{\frac{1N}{3}+P-1}, x_{\frac{2N}{3}+P-1}\}
\end{aligned}
$$

Figure 2.5 shows the decimation-in-frequency approach partially. Here also, the data values need to be multiplied with the twiddle factors.

Figure 2.4 DIF Fourier Transform

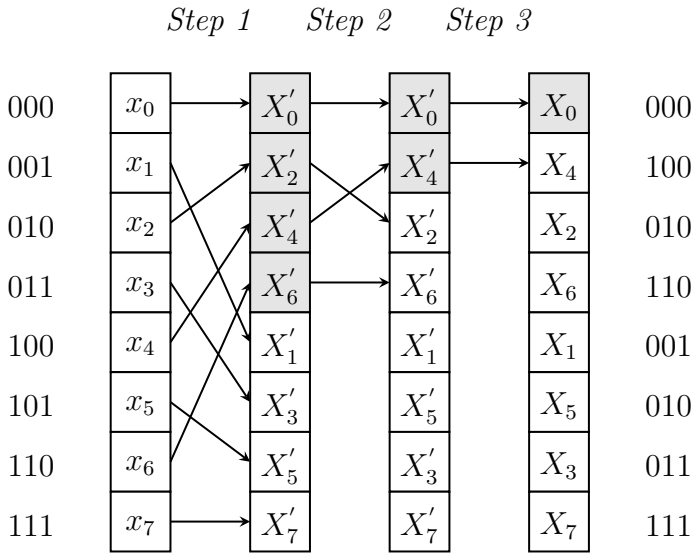


2.2.3 FFT Output Rearrangement

We have seen in the previous sections that the input or output of a FFT algorithm is not in natural order, depending on the approach we use. Furthermore, each decimation causes the output to become more out of order. Lets look at a DIF FFT radix-2 example. It can

only be used on 2^n sized data, we choose $N = 2^3$. We have in total $\log_2 8 = 3$ decimations, or *steps*. A unique relationship reveals itself with radix-2. When binary encoding the data positions, we can see that the output is bit-reversed, regardless of the approach used.

Figure 2.5 DIF data ordering



Step 1 will partition the data into two parts of size 4, *Step 2* will create 4 partitions of size 2, finishing off with *Step 3* creating 8 partitions of size 1. The input does not have to be in natural order. We could also feed the radix-2 algorithm bit-reversed input, and it will come out in natural order! To handle a more general case, Algorithm 2.2 will show the stepwise reordering of data (shown in Figure 2.5) for any radix-r algorithm. The reordering can also be done in just one step.

Algorithm 2.2 Reorder data stepwise

▷ X is the input array

▷ r is the radix

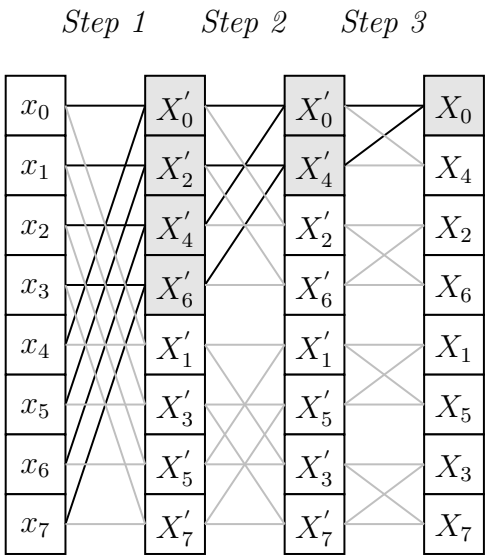
▷ $N \leftarrow |X|$

function REORDER(X, r, N)
  for $S \leftarrow N$ **to** 1 **do**
    for $offset \leftarrow 0$ **to** N **do**
      for $i \leftarrow 0$ **to** S **do**
        $T[offset + (i \mod r) \cdot \frac{S}{r} + \frac{i}{r}] \leftarrow X[offset + i]$
        $i \leftarrow i + 1$
      end for
      $offset \leftarrow offset + S$
    end for
    $X \leftarrow T$
    $S \leftarrow S/r$
  end for
end function

2.2.4 Butterflies

The term butterfly refers to the phase of the computation that combines sub-DFTs or decimates the DFT into sub-DFTs. The name comes from the shapes that emerge when looking at the data-flow diagram of a FFT. Figure 2.6 shows data accesses for a DIF radix-2 algorithm. Butterfly shapes are most recognizable at *step 3*.

Figure 2.6 DIF radix-2 data access



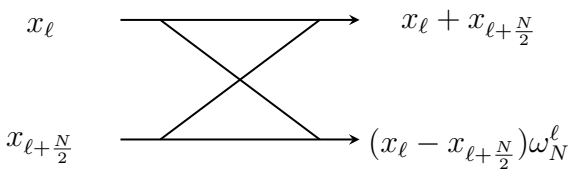
Gentleman-Sande [11] have made these butterflies computationally more efficient for decimation-in-frequency, by precalculating twiddle factors and incorporating them in the calculation. We will only cover the radix-2 case. To solve one butterfly we get the two equations:

$$X_\ell^0 = (x_\ell \cdot \omega_2^0 + x_{\ell+\frac{N}{2}} \cdot \omega_2^0)\omega_N^0, \tag{2.17}$$

$$X_\ell^1 = (x_\ell \cdot \omega_2^0 + x_{\ell+\frac{N}{2}} \cdot \omega_2^1)\omega_N^\ell, \tag{2.18}$$

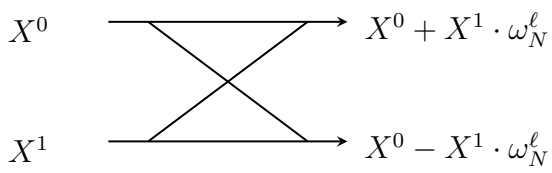
with $\ell = \{0, 1, \ldots, \frac{N}{2} - 1\}$. We can simplify this, because $\omega_2^0 = 1 + i0$ and $\omega_2^1 = -1 + i0$, thus we get the butterfly in Figure 2.7.

Figure 2.7 DIF radix-2 butterfly



Analogously, Cooley-Tukey [8] optimized the butterflies for decimation-in-time, resulting in Figure 2.8.

Figure 2.8 DIT radix-2 butterfly



We could do this trick for all radix-r algorithms, but the twiddle factors don't always evaluate that nicely. Radix-2 and radix-4 are optimal, as their twiddle factors do not contain values from $\mathbb{R}$, but from $\mathbb{N}$ (the set of natural numbers).

2.2.5 Variants

There are variants to the FFT, as Section 2.2.1 hinted towards. When talking about the algorithms, we refer to the DIF approach. The *radix-r* algorithm needs input of size r^n. At each step, the DFT is decimated into r sub-DFTs. The limited data sizes it can handle are very restricting, which brings us to *mixed-radix*. We do not have to always decimate into the same number of sub-DFTs. We can change r to whatever we want at each step, as long as the DFT size is dividable by r. Termination is the same as for radix-r and the allowed data sizes are less restricting, namely $2^n \cdot 3^m \cdot 4^o \cdot 5^p \cdot \ldots$. There is a different approach called *Split-radix*[9], which uses a composition of radixes at one step. A FFT of size $N = 8$ can be solved by a radix-2 algorithm, but it can also be solved by splitting it into sub problems of size 2, 2 and 4, solving it by respective radix-r algorithms.

2.3 Multi-Dimensional Fourier Transform

We have seen the 1-dimensional mathematical representation of the DFT in Section 2.1. The equation for the n-dimensional case is as follows:

$$X_r = \sum_{\ell=0}^{N-1} x_\ell \omega_N^{r\ell}, \tag{2.19}$$

where x is n-dimensional data of size $N_1 \times N_2 \times \cdots \times N_n$, $r = (r_1, r_2, \ldots, r_n)$, $N = (N_1, N_2, \ldots, N_n)$, $\ell = (\ell_1, \ell_2, \ldots, \ell_n)$, $\omega_N^{r\ell} = e^{-i2\pi \frac{r\ell}{N}}$, e the base of natural logarithm and $i = \sqrt{-1}$. In other words, 1-dimensional DFTs are performed along each dimension. Let us consider a 2-dimensional example:

$$X(r_1, r_2) = \sum_{\ell_2=0}^{N_2-1} \sum_{\ell_1=0}^{N_1-1} x(\ell_1, \ell_2) \omega_{N_1}^{\ell_1 r_1} \omega_{N_2}^{\ell_2 r_2}, \tag{2.20}$$

with x the 2-dimensional input data of size $N_1 \times N_2$. Obviously, this DFT can be solved by a FFT algorithm, given the appropriate derivation as shown in Section 2.2.

Chapter 3

GPU Optimizations

There are two ways to improve performance of any algorithm: improve its time complexity and/or make it architecture specific. The later takes the host architecture into account in (re-)designing the algorithm. We will use the GPU as the host architecture. Our approach to improve the Fast Fourier Transform is to take as much knowledge into account as possible. Popular FFT algorithms like FFTW[10] (for CPU) and NVIDIA's FFT[5] (for GPU) both have an optional preparation step. When invoked, the algorithm is optimized with the knowledge present at the time. This takes time and is therefore only advisable when running batches of the same size FFTs. We will tempt to do the same.

To make an algorithm architecture specific, we must first understand the architecture. A GPU architecture is described in Section 3.1. We will then continue to describing optimizations to the FFT specific to the presented architecture in Section 3.2.

3.1 GPU Architecture

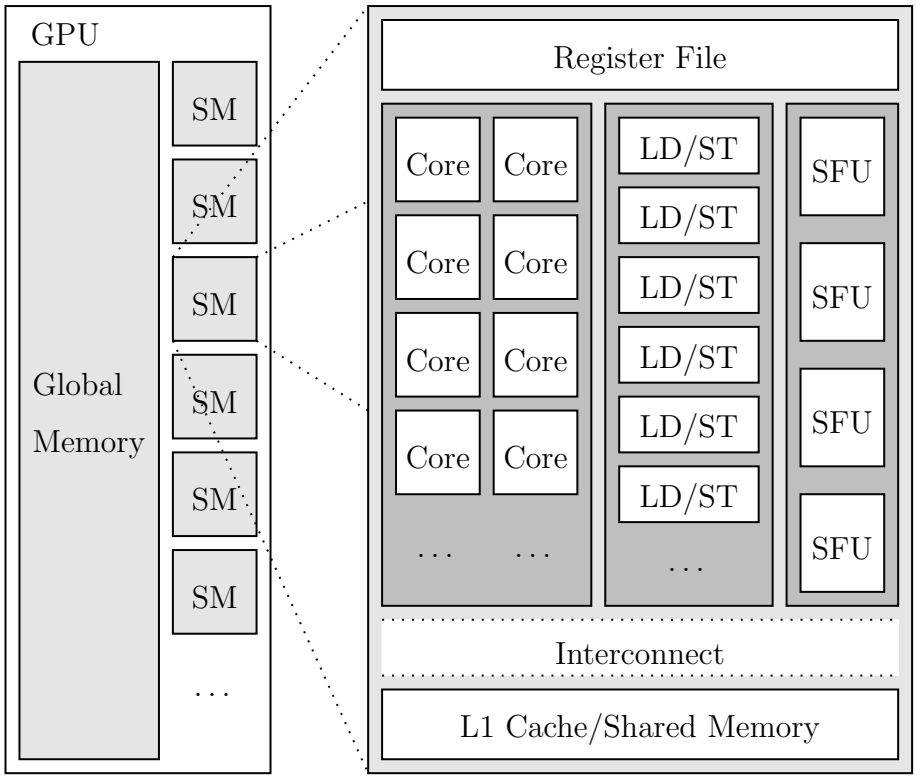
The architecture of choice is one called *Fermi*[2, 3, 4] from NVIDIA. They make a distinction between different Fermi architectures with the term *compute capability*. We will use compute capability 2.0 from now on. The manner in which we will run software is defined by NVIDIA's *Compute Unified Device Architecture* (CUDA), which is a parallel programming framework that extends the C programming language. Today, also C++ is supported.

We first show a global overview of the architecture in Section 3.1.1. Threads running on this architecture have certain restrictions which will be discussed in Section 3.1.2. Sections 3.1.3 through 3.1.5 describe the different types of memory and how to use them optimally.

3.1.1 Hardware Architecture

Figure 3.1 shows the Fermi architecture partially. Only the parts are shown that are of interest in this document. The GPU has components called *Streaming Multiprocessors* (SM), which contain a certain number of cores. SMs cannot access each others local storage but can all access global memory.

Figure 3.1 Fermi architecture



A SM has four Special Function Units (SFU) dedicated to perform instructions such as sin, cosine, square root, etc. One instruction per thread, per clock, is executed on each SFU. Threads are scheduled in batches of 32, thus it will take 8 clock cycles. A SM also has 16 load and store (LD/ST) units. Each unit calculates a source or destination address in one clock cycle, after which the data is either store or loaded by the same unit. There is also a 64KB configurable L1 cache. The user can choose between two configurations:

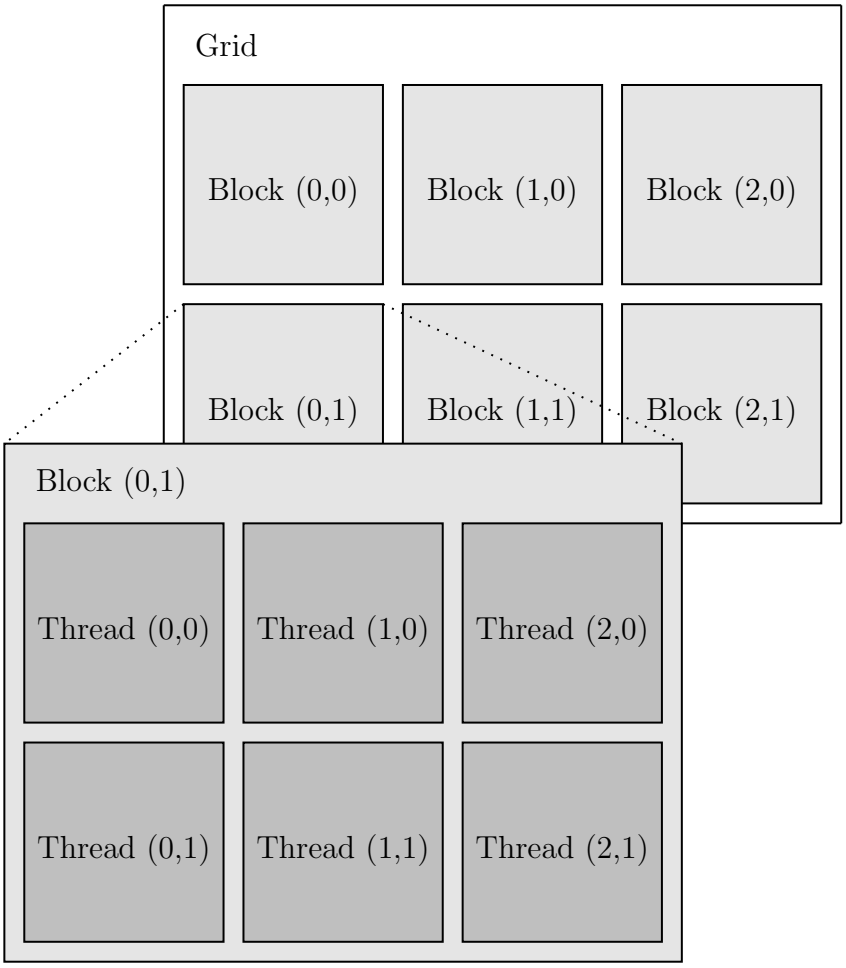
- 16KB Shared memory - 48KB L1 cache
- 48KB Shared memory - 16KB L1 cache

The L1 cache can only be accessed by the SM it is located on. A larger 768KB L2 cache is reachable from each SM. The dynamics of data-flow are addressed in Section 3.1.2, which shows the capabilities of threads.

3.1.2 Thread Execution

A CUDA *kernel* entails a function including all functions it depends on, much like C programming with its `main` function and depending functions. A kernel can therefore be seen as a standalone application. Each kernel is executed by a user specified number of threads with the following structure. A *grid* is an at most 3-dimensional structure consisting of *blocks*. In turn, blocks are an at most 3-dimensional structure consisting of *threads*. This structure is visualized in Figure 3.2, taking as an example a 2×3 grid and block structure.

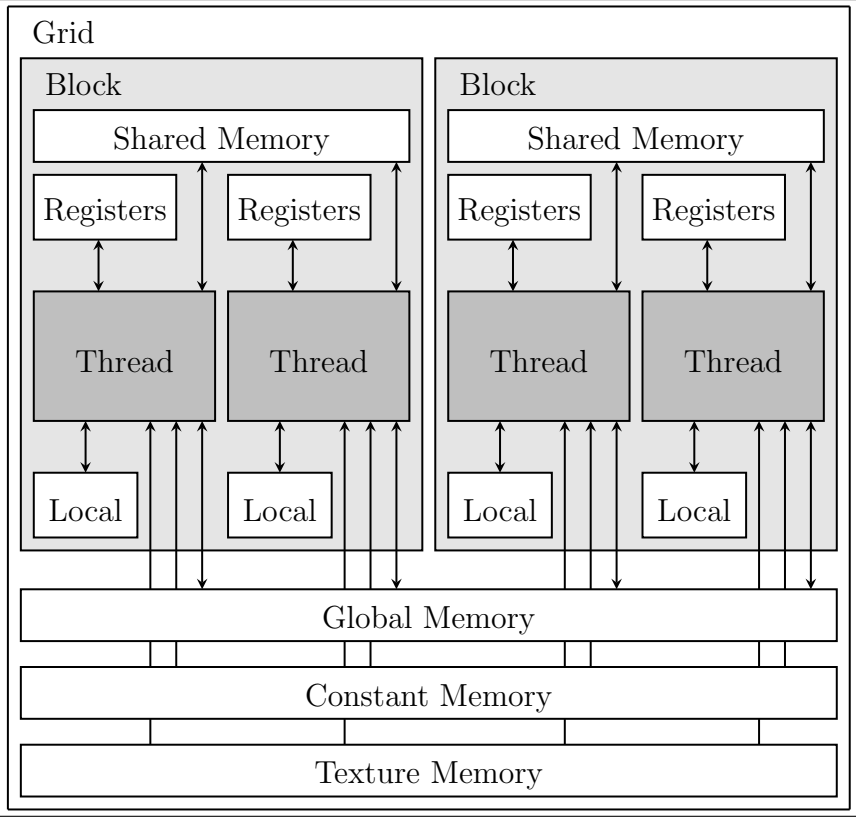
Figure 3.2 CUDA thread structure



Each level has a set of restrictions. Blocks are distributed to SMs and a block is executed as a whole on a SM. When multiple blocks are executed on the same SM, they have to

share the resources. Blocks can only exchange information through global memory, whereas threads can exchange information through shared memory if they reside within the same block. Threads within a block are executed in batches of 32, called *warps*. It is always advisable to therefore execute the function with a multiple of 32 threads. When talking about threads, an apparent issue is synchronization. CUDA provide such functionality, yet it is limited to threads within a block. Block synchronization can only be achieved at function termination or by implementing it through global memory. Figure 3.3 shows all the data transfer capabilities and restrictions.

Figure 3.3 CUDA memory model



There is a limit to how many threads can be executed concurrently, because resources on a SM are shared by all threads running on it. Threads can use a maximum of 64 registers each. Mind that as functions grow more complex, the number of registers required to run it also increases. Some parameters that restrict *thread occupancy* are shown in Table 3.1.

Table 3.1 thread restricting parameters

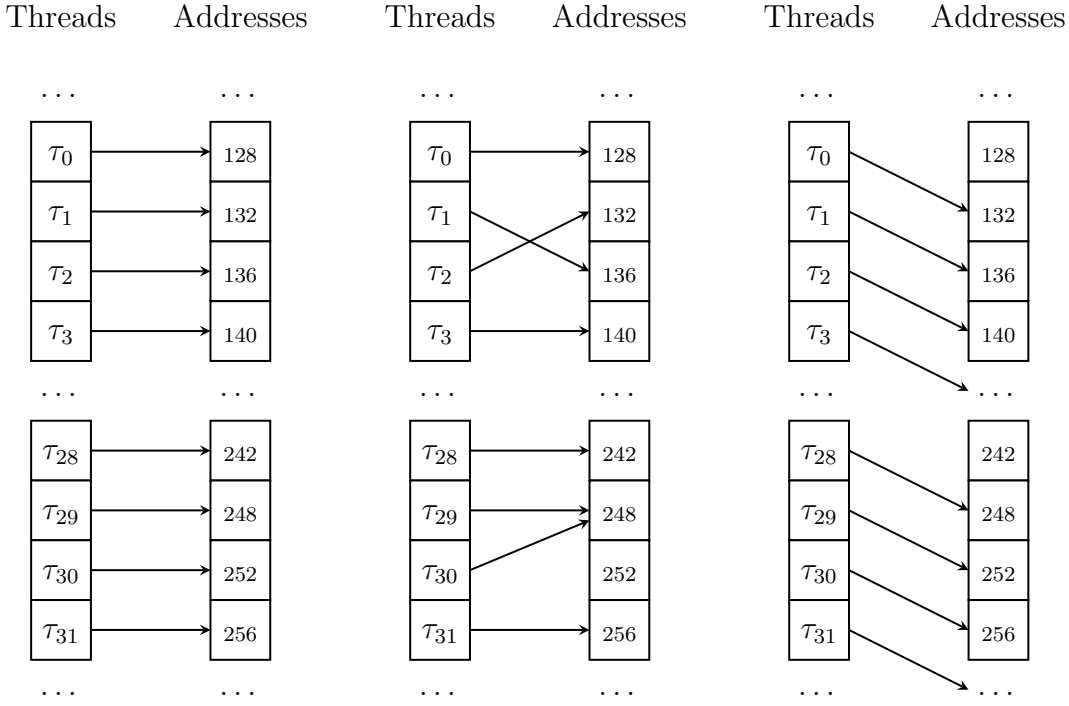
Max number of threads per block	1024
Max number of blocks per SM	8
Max number of threads per SM	1536
Max amount of shared memory per SM	48KB
Max amount of 32-bit registers per SM	32KB
Max resident threads per SM	1536
Max resident blocks per SM	8
Max resident warps per SM	48
Max amount of 32-bit registers per thread	63

3.1.3 Global Memory

Global memory is a virtual address space of which the values initially reside in off-chip DRAM. It is the biggest and slowest memory on the GPU. Accesses are optimized by a cache hierarchy much like CPUs have, consisting of a Level-2 (L2) and Level-1 (L1) caches. Each SM has its own L1 cache, which other SMs cannot access. The L2 cache is bigger, slower and each SM has access to it. Communication between SMs is fastest through this cache. When a thread requests a variable from global memory (when it is not already in a register), it will look for it in L1. If it is there we speak of a cache *hit*, otherwise it is a cache *miss* and we request the variable from the next memory in the hierarchy, the L2 cache. If it is not in the L2 cache we must go through DRAM. A variable in global memory can thus reside in DRAM, L2, L1 or a register. When a cache is full, values need to be *evicted* to make room for others. When evicting a value, it is written to the next memory in the hierarchy. We will be working with arrays of data. An array element will get evicted when (1) the cache is full and (2) it is has not been accessed the longest. Writes to global (or shared) memory are only guaranteed to be visible by other threads after performing memory fence functions or synchronization functions like `__threadfence_block()`, `__threadfence()`, or `__syncthreads()`. When the volatile keyword is used, the compiler assumes the value can

be accessed at any time. All accesses will thus compile to actual memory read or write instructions. A cache line is 128 bytes and maps to a 128 byte aligned segment in global memory, which can be fully utilized by a parallel access mechanism. Note that typical data types like single precision floating point values take up 4 bytes. A cache line can thus hold 32 values. Given data set $x = \{x_0, x_1, \ldots, x_{N-1}\}$, with $N = |x|$, global memory access is said to be *coalesced* when 32 threads access data, and thread τ accesses elements $x_{\tau+32\mu}$, where $\mu = \{0, 1, \ldots, \frac{N}{32} - 1\}$. This is more amply shown in Figure 3.9, where threads $\Gamma = \{\tau_0, \tau_1, \ldots, \tau_{31}\}$ access memory addresses.

Figure 3.4 Coalescing examples for global memory access



(a) Coalesced **(b)** Not coalesced **(c)** Not coalesced

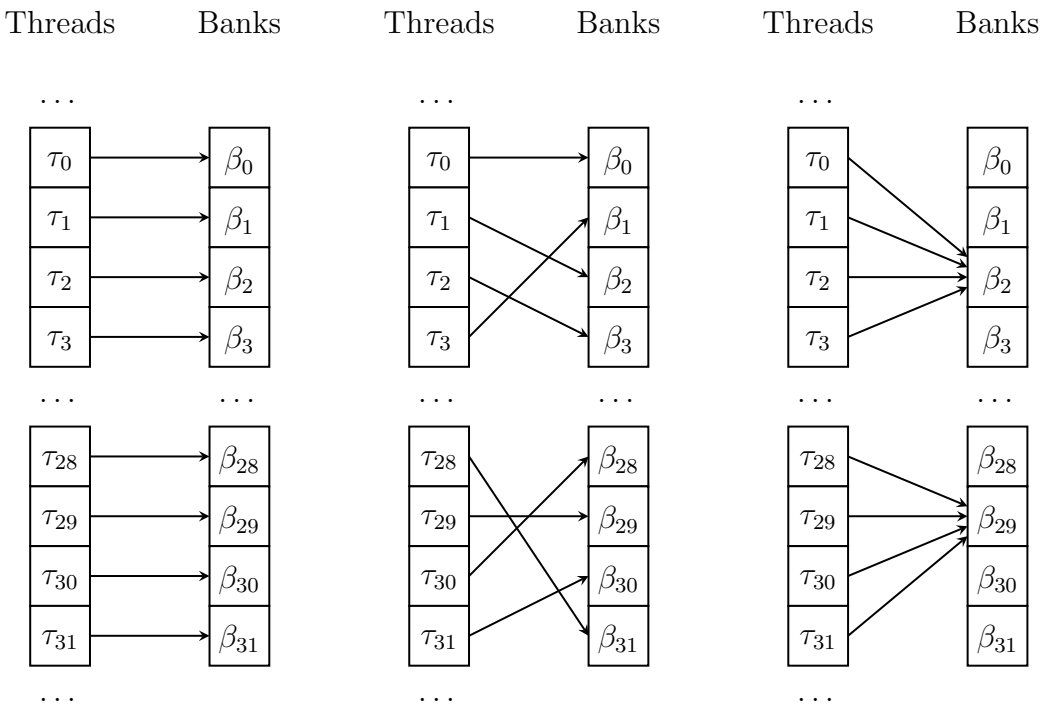
As you can see from Figure 3.9, the data segments must also be aligned to 128 bytes. When caching in both L1 and L2, the accesses are performed with 128-byte transactions. If accesses are limited only to L2, then they are performed with 32-byte transactions reducing over-fetching in case of scattered access patterns. The lifespan of data stored in global

memory lasts the execution of the entire program. On a side note, *local memory* is the same as global memory and is thus managed the same way. The only difference consists of the use of a different virtual address space.

3.1.4 Shared Memory

Shared memory resides on-chip and is used to shared data among threads within a block. It consists of 32 *banks*, which each have 32-bit bandwidth. When an array is stored, consecutive 4-byte elements are stored in consecutive banks. We speak of a *bank conflict* when multiple threads access a different element in the same bank. Figure 3.5 shows when conflicts occur.

Figure 3.5 Bank conflicts in shared memory



(a) Conflict free **(b)** Conflict free **(c)** (Not) conflict free

Figure 3.5a obviously has no conflicts. Figure 3.5b has a scattered access pattern, but also

has no conflicts. Figure 3.5c however brings an additional mechanism to light. Shared memory has the ability to broadcast when the same data element within a bank is accessed by multiple threads, thus not resulting in a bank conflict. Yet we do have a conflict if multiple threads access different elements in the same bank. Reducing bank conflict will increase performance. In contrast to global memory, the lifespan of data stored in shared memory only lasts the execution of a block running the function.

3.1.5 Constant and Texture Memory

Two storage facilities have not yet been discussed, namely *constant memory* and *texture memory*. Constant memory is 64KB, which has a broadcast functionality much like shared memory. It is located however in device memory just like global memory, and thus share the same access time. Each SM does have a constant cache of 8KB. Texture memory has different limits regarding different textures. A 1-dimensional CUDA array has a maximum width of 65536 elements for example. It caches neighboring elements when an access is done, thus is optimized for walking through an array for example. It could be said that constant memory is optimized for temporal locality and texture memory is optimized for spacial locality.

3.2 Combining the FFT and GPU

The FFT is a data access intensive algorithm. Using fast memory when available is of the essence. Global memory is the entry point of the GPU and is therefore unavoidable. It has an access time of between the 400 to 800 clock cycles, for a maximum bandwidth of 128 bytes (a cache line). Registers have an access time of 22 clock cycles in case of a back-to-back register dependency, where the current instruction is using the output of the previous instruction. Shared memory consists of 32 banks. Each bank has a bandwidth of 32 bits per two clock cycles. These differences become much greater with a higher compute capability. Optimizing data access therefore means migrating accesses from global memory to shared memory or registers. We will focus our efforts on optimizing the 1-dimensional FFT. Any optimization for the 1-dimensional FFT also holds for the multi-dimensional FFT, because it consists of 1-dimensional FFTs along each dimension. Per dimension, each 1-dimensional FFT can run in parallel as seen in Section 2.3.

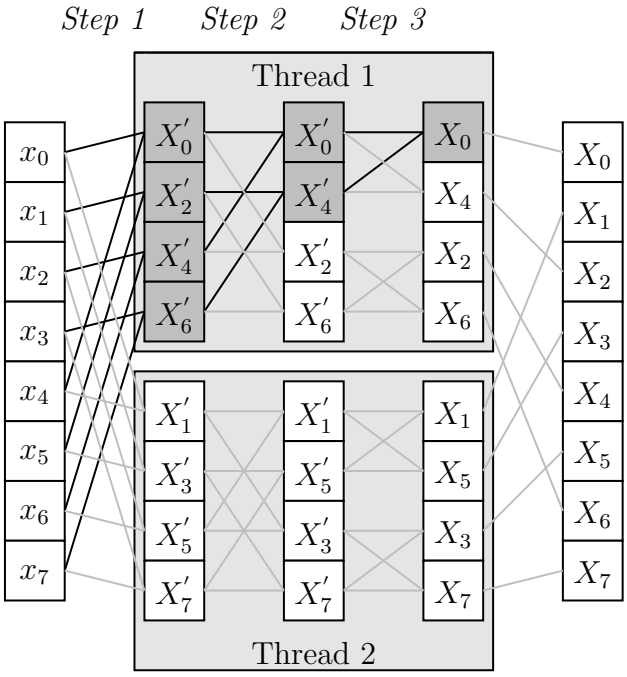
Each stage of the algorithm is discussed starting with the explanation in what ways the computations can be divided among threads in Section 3.2.1. The way input data is processed is discussed in Section 3.2.2. We then continue with how to minimize and optimize instructions in sections 3.2.3 through 3.2.4. We then solve the remainder of the FFT in Section 3.2.5 and handling the output in Section 3.2.6. We finish with an approach calculating the FFT in multiple GPUs in Section 3.2.7

3.2.1 Thread-Level-Parallelism

The way the FFT can be parallelized is dictated by the abundance of data dependencies like the ones shown in Figure 2.5. There are data dependencies between each step of the algorithm, thus *Step 1* must be executed before *Step 2* and so on. This requirement has to be met only partially. Figure 3.6 shows an example of how to divide the labor among threads given these dependencies for a $N = 8$ DIF radix-2 FFT. The threads can run

concurrently without interaction with each other, because they only depend on the input data. We will use the example for the remainder of this chapter.

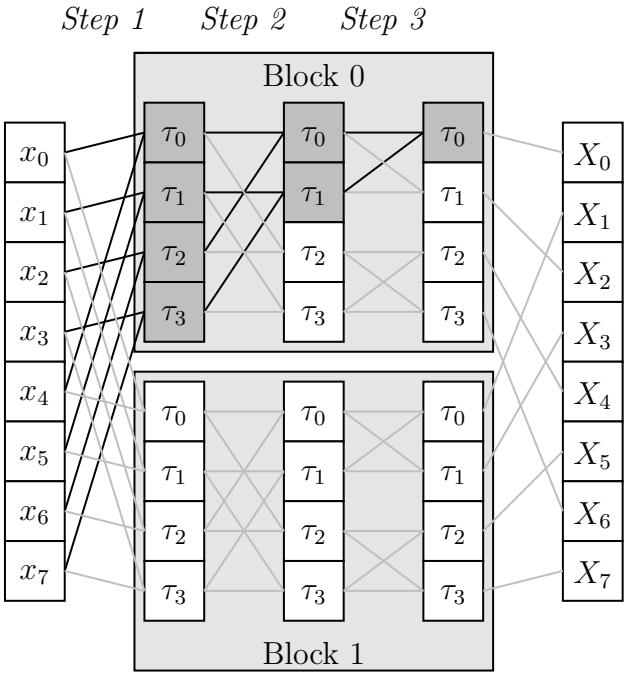
Figure 3.6 Parallelized DIF radix-2



There are ways to increase the number of threads used in Figure 3.6. We could do the division of labor at a later step, giving the opportunity to create more threads. We could use more threads initially, either idling a percentage of the threads or increase communication between them, or we could use a higher radix at the first step. In fact, *the output element at each step can be calculated in parallel*. Measures regarding data dependencies need to be added in this case. When scheduling threads on a CPU, a typical approach is to run each thread on a different core. The data processed by each thread is half the input size given the previous example. This is to big workload for a thread running on a GPU due to restricting parameters given in Section 3.1.2. The Fermi architecture explained in Section 3.1 clearly shows that threads can exchange data on a SM via shared memory and that data exchange between SMs goes through global memory, which is much slower. To keep data exchange between SMs to a minimum, imagine the threads in Figure 3.6 being SMs on a GPU. Each

SM would only be dependent on the input data. Figure 3.7 shows the division of labor for two blocks (running on different SMs) with threads $T = \{\tau_0, \tau_1, \tau_3, \tau_4\}$. Threads within a block must synchronize at each step to obey data dependency restrictions.

Figure 3.7 Division of labor for GPU



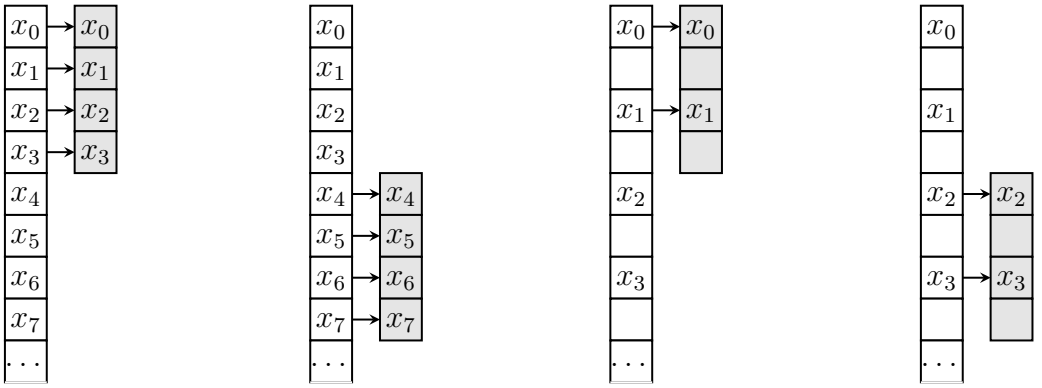
The following sections will break this approach down and discuss optimizations for each stage of the algorithm.

3.2.2 Data Input

The input data is originally stored in global memory and must be transferred to shared memory on the respective SM. The manner in which this is done greatly influences performance. The FFT is an example of having a *simple-strided* access pattern, where each access is for the same number of bytes and the pointer address is incremented with the same amount between accesses. To fully use a cache line, we must do a coalesced memory access as described in Section 3.1.3. For our host architecture, we say that elements are accessed in *stride-ρ* when a warp accesses 4-byte elements and the distance between element

addresses is $\rho \cdot 4$. By this definition, access is coalesced in case of a stride-1 access pattern and is ρ times slower in case of a stride-ρ access pattern. For demonstration purposes from now on, let a warp consist of 4 threads and let a cache line be of bandwidth 16 bytes. Figure 3.8 gives a strided access pattern example with input data $x = \{x_0, x_1, \dots\}$ on the left and a cache line on the right of each figure.

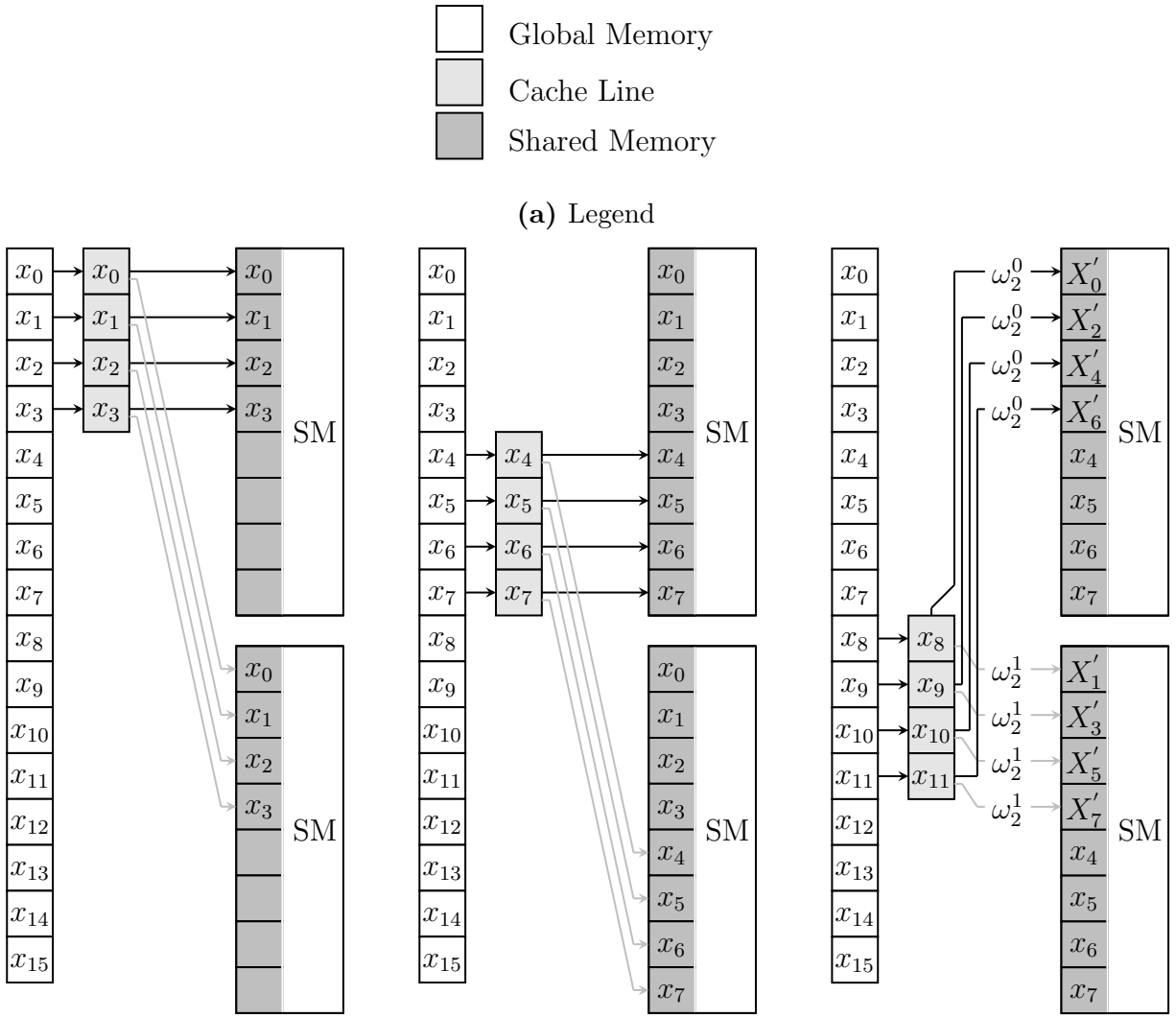
Figure 3.8 Cache line usage in case of strided access patterns



(a) Stride-1: part 1 **(b)** Stride-1: part 2 **(c)** Stride-2: part 1 **(d)** Stride-2: part 2

It follows from Section 2.2 that a DIT radix-r FFT has a stride-r access pattern. For this reason we choose the DIF approach. Although a single thread has a stride-$\frac{N}{r}$ access pattern, when put together in a warp, we get coalesced access. A complex data element consists of a real and imaginary part represented by real numbers. We could store the 4-byte numbers consecutively, creating 8-byte elements. This situation is comparable to a stride-2 access pattern. For a thread to read a complex number it must do two memory accesses. The number of threads being able to read concurrently is thereby divided by two. For this reason it is advisable to create an array holding the real parts and an array holding the imaginary parts. As resources are limited per SM, we won't copy all the input data to shared memory, we will do the first step of the algorithm and reduce the size of the data by the radix used. Figure 3.8 shows an example for a $N = 16$ DIF radix-2 FFT.

Figure 3.9 Stride-1 example for DIF radix-2 FFT



(a) Legend

(b) Action 1

(c) Action 2

(d) Action 3

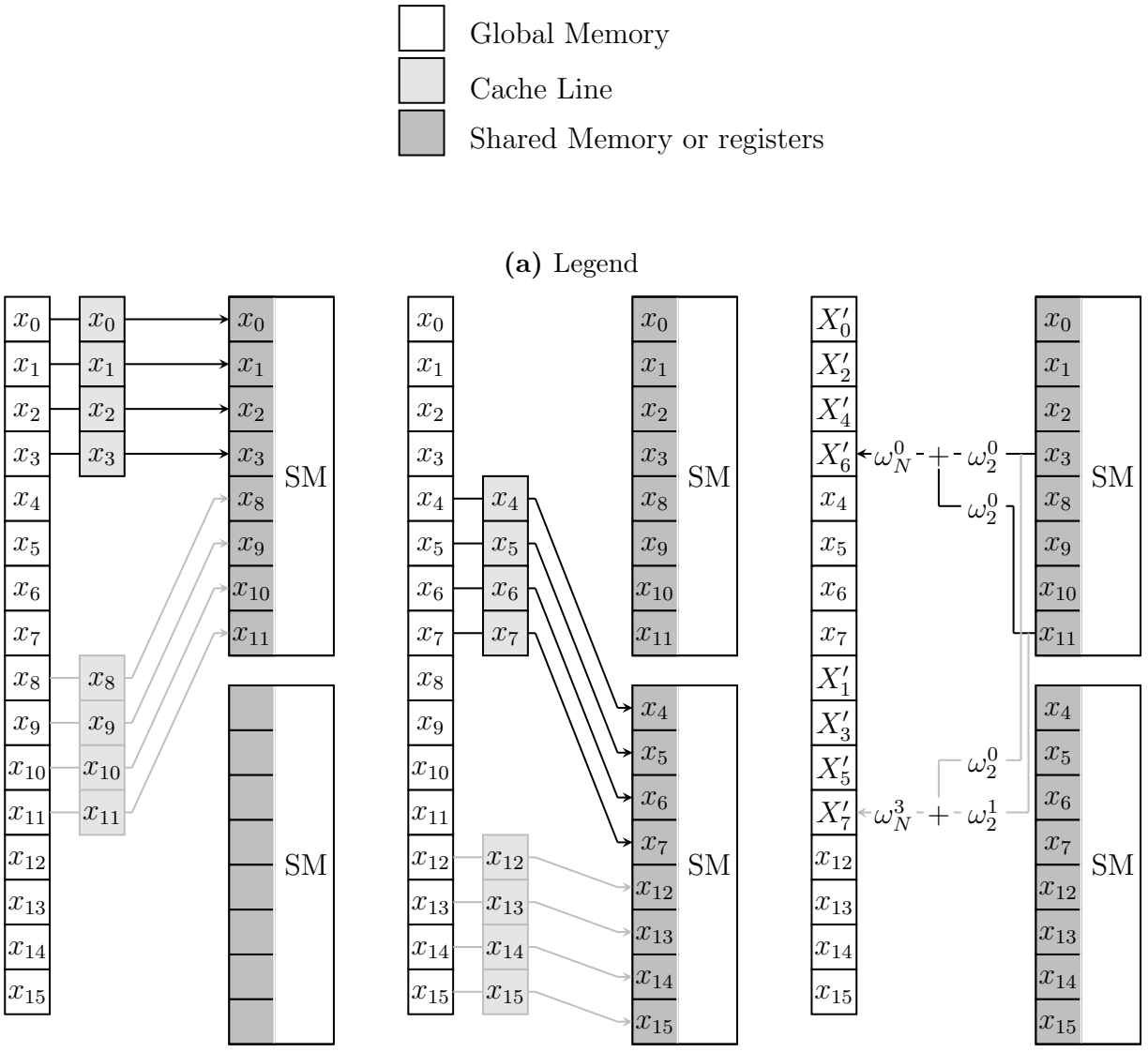
This continues until all elements in global memory have been read and processed. Note that not all necessary computations are visible in Figure 3.9d. They were discussed in Section 2.2.2. With this approach, the data is read coalesced and only once from global memory and the entries in the L2 cache are re-used for every SM. Optimally this should be done by a broadcast mechanism, sending data in one go to all SMs. Unfortunately,

NVIDIA did not implement this for global memory as threads would have to be synchronized globally. At each radix-r step of the algorithm this will cost $(r-1)N$ L2 accesses, but the way the data is accessed gives the opportunity to easily overlap computation and host to GPU data transfer. Not all data needs to be present when the computation starts. We could for instance send half the data and start the algorithm, meanwhile sending the second half. Each radix-r butterfly is calculated in r stages. We refer to this computation type as *distributed butterflies*. While each 128-bit data access is serialized, computation is parallelized. Data transfer is relatively very costly between host and GPU, thus there is a balance to be found when to apply this method. It is, however, transparent how to implement this on multiple GPUs (substituting `SM` in Figure 3.9 with `GPU`). The only dependency would be on the input data. No inter GPU communication is therefore required and each GPU would simply compute a part of the FFT. This also holds for SMs of course. Keep in mind that each GPU would generate interleaved output data. The method can also be used when then GPU has insufficient memory to hold the data. The FFT can be calculated by successive calls to the GPU.

Figure 3.10 shows a different approach that reads the input from global memory one time and has no need to re-use the L2 cache. Global synchronization and a write-back to global memory are required at each step of the algorithm (until the DFTs can fit on a SM), because each SM produces input for another SM. The computations shown in Figure 3.10d is one (unoptimized) radix-2 butterfly which can be calculated by one or two threads. When r threads are used to compute a radix-r step, shared memory must be used and each value is read r times. Computation will be parallelized in this case. When 1 thread is used, values are read 1 time when using registers or shared memory, but the computation will be serialized. When registers are used, there are a couple of things to keep in mind. Using more registers reduces the number of threads that can execute the algorithm as resources must be shared. Registers cannot hold an array that is accessed dynamically, thus an easy indexing scheme is not possible. Variables in registers need to be explicitly called when

they are needed, increasing complexity of the code itself and making it less flexible. In turn, it is hard to guarantee no bank conflicts when using shared memory. Though accesses are coalesced as the cache line is fully used, we can conclude from Figure 3.10a and 3.10b that overlapping data transfer from host to GPU and computation is considerably harder, as access are strided and less predictable.

Figure 3.10 Example for DIF radix-2 FFT



(a) Legend

(b) Action 1

(c) Action 2

(d) Action 3

In any case, it is wise to have data aligned in favor of cache lines to optimize accesses. The way this works for one step is to align the data in a previous step though padding. Let a cache line be of σ-byte bandwidth and the first step will be of radix-r. Aligned data $q = \{q_0, q_1, \ldots, q_j, \ldots, q_N\}$ is created from input data $x = \{x_0, x_1, \ldots, x_j, \ldots, x_N\}$ with the following re-indexing scheme.

$$q_j \quad \leftarrow \quad x_{p \,\cdot\, \frac{j}{s} \,+\, j \mod s}, \tag{3.1}$$

where $s = \frac{N}{r}$ and $p = \sigma \cdot \lceil \frac{s}{\sigma} \rceil$. Padding can be done by a simple algorithm using Equation 3.1, but the fastest way to realize this is to use a CUDA function `cudaMallocPitch` which allocates a 2D array of which each row is padded by a specified amount. An alignment alternative is to transfer the data from CPU to the GPU in pieces of size $\frac{N}{r}$ (when the first step is radix-r), because CUDA automatically aligns each new variable or array. We do have to pass all the pointers to the variables or arrays as arguments.

To retain coalesced access for multiple steps we must go through shared memory as an intermediate step. This is shown by Algorithm 3.1.

Algorithm 3.1 Coalesced access on misaligned data

▷ x is the input array in global memory

▷ n is the sub-DFT size

▷ N is the input size

▷ ID is the thread identification nr

▷ $THREADS$ is the total nr of threads

function READ(x, $xshared$,$offset$, n)

$offset\_mod = offset \mod 32$

for $i \leftarrow ID$ **to** $n + offset\_mod : i = i + THREADS$ **do**

$j = i - offset\_mod$

if $j \geq 0$ **then**

$xshared[i] = x[offset + j]$

end if

end for

end function

procedure FFT(x, n, N)

$xshared$

for $offset \leftarrow 0$ **to** $N : offset = offset + n$ **do**

READ$(x, xshared, offset, n)$

// calculate n sized fft from offset to offset+n

// write back coalesced

end for

end procedure

3.2.3 Pre-calculating Twiddle Factors

Time can be saved by pre-calculating twiddle factors. These can be used to create optimized butterflies as seen in Section 2.2.4 or just as replacement for the computation otherwise to be done. This can possibly cost time. Instead of calculating the twiddle factor, you will need to do a memory access. Depending on the location of the data, this could actually slow the algorithm down. ϕ will represent a data element, in our case a complex number. Complex numbers consist of two part. A real and an imaginary part, represented by ϕ_r and ϕ_i, respectively. The appropriate twiddle factor will be represented by ω, and also consists of a real and imaginary part. We now demonstrate the multiplication of an input data element with the twiddle factor, or in general, two complex numbers. The definition was given in Section 2.1.1.

$$\phi = \omega \cdot \phi,$$

$$\begin{aligned} \phi_r &= \omega_r \cdot \phi_r - \omega_i \cdot \phi_i, \\ \phi_i &= \omega_i \cdot \phi_r + \omega_r \cdot \phi_i. \end{aligned}$$

This multiplication can be rewritten or simplified in occurrence of 1, -1 or 0, i.e. multiplying with 0 is zero, etc. Table 3.2 shows the rewritten multiplications in case we know one of these values is encountered.

Table 3.2 Simplified complex multiplications

		Real			Imaginary		
ω_r	ω_i	$\phi_r \neq 0$		$\phi_i \neq 0$	$\phi_r \neq 0$		$\phi_i \neq 0$
0	0	0			0		
0	1	0	$-$	ϕ_i	ϕ_r		
0	-1	0	$+$	ϕ_i	-ϕ_r		
0	*	0	$-$	$\omega_i \cdot \phi_i$	$\omega_i \cdot \phi_r$		
1	0	ϕ_r			0	$+$	ϕ_i
1	1	ϕ_r	$-$	ϕ_i	ϕ_r	$+$	ϕ_i
1	-1	ϕ_r	$+$	ϕ_i	-ϕ_r	$+$	ϕ_i
1	*	ϕ_r	$-$	$\omega_i \cdot \phi_i$	$\omega_i \cdot \phi_r$	$+$	ϕ_i
-1	0	-ϕ_r			0	$-$	ϕ_i
-1	1	-ϕ_r	$-$	ϕ_i	ϕ_r	$-$	ϕ_i
-1	-1	-ϕ_r	$+$	ϕ_i	-ϕ_r	$-$	ϕ_i
-1	*	-ϕ_r	$-$	$\omega_i \cdot \phi_i$	$\omega_i \cdot \phi_r$	$-$	ϕ_i
*	0	$\omega_r \cdot \phi_r$			0	$+$	$\omega_r \cdot \phi_i$
*	1	$\omega_r \cdot \phi_r$	$-$	ϕ_i	ϕ_r	$+$	$\omega_r \cdot \phi_i$
*	-1	$\omega_r \cdot \phi_r$	$+$	ϕ_i	-ϕ_r	$+$	$\omega_r \cdot \phi_i$
*	*	$\omega_r \cdot \phi_r$	$-$	$\omega_i \cdot \phi_i$	$\omega_i \cdot \phi_r$	$+$	$\omega_r \cdot \phi_i$

We can further optimize the table if we know we are dealing with real input data and eliminate additional multiplications, subtractions and additions. The output of one step of a FFT algorithm is complex data, thus we can only take advantage of the knowledge that we are using real data at the first step of the algorithm. This can be used as a counter weight against increasing the radix (and thus the complexity) at the first step to reduce SM interactions.

3.2.4 Instruction-Level-Parallelism

A factor in the performance of a warp is *instruction-level-parallelism* (ILP). There are two ways of holding up a warp or consecutive warps: *data dependencies* and *branch instructions*. The following simple example of instructions cannot be executed at the same time, because there is a data dependency between instruction (1) and (3): variable a is used in the third instruction to determine the value of d.

$$\begin{aligned} (1) \quad a &= b \\ (2) \quad c &= c + 1 \\ (3) \quad d &= a + b \end{aligned}$$

Data dependencies cannot be prevented in case of the FFT, but branch instructions can be kept to a minimum. CUDA offers the ability to let each thread in a block execute different instructions. As each thread executes the same code, this is done by branching on *thread ID*. How this thread ID is determined is up to the programmer. Table 3.3 contains the CUDA build-in variables that can be used for identification purposes, of which the value is determined at run-time. The value d must be substituted for x, y or z.

Table 3.3 CUDA run-time indexing

`gridDim.`d	Number of blocks on d-axis of grid
`blockDim.`d	Number of threads on d-axis of block
`blockIdx.`d	d-coordinate of current block
`threadIdx.`d	d-coordinate of current thread

Section 3.2.1 and 3.2.2 shows how we can choose a radix-r for the first step and divide the data over r blocks. Figure 3.9 shows that each SM performs different calculations. This can be achieved by branching on block ID. Branching instruction are conditional instructions that control the flow of execution (an if-statement, while-loop, for-loop, etc.). The following example assumes the same setup as in Section 3.2.1 and shows the calculations to be done by a block, not a thread.

Algorithm 3.2 DIF radix-2 FFT with branching

if blockIdx.x == 0 **then**

$X^0 \leftarrow x_{0\ldots\frac{N}{2}-1} \cdot \omega_2^0$

$X^0 \leftarrow X^0 + x_{\frac{N}{2}\ldots N} \cdot \omega_2^0$

$X^0 \leftarrow X^0 \cdot \omega_N^0$

else if blockIdx.x == 1 **then**

$X^1 \leftarrow x_{0\ldots\frac{N}{2}-1} \cdot \omega_2^0$

$X^1 \leftarrow X^1 + x_{\frac{N}{2}\ldots N} \cdot \omega_2^1$

$X^1 \leftarrow X^1 \cdot \omega_N^{0\ldots\frac{N}{2}}$

end if

The CUDA indexing scheme can be used to remove the branching instructions (if-statements), which creates the following algorithm.

Algorithm 3.3 DIF radix-2 FFT without branching

$j \leftarrow$ blockIdx.x

$X^j \leftarrow x_{0\ldots\frac{N}{2}-1} \cdot \omega_2^0$

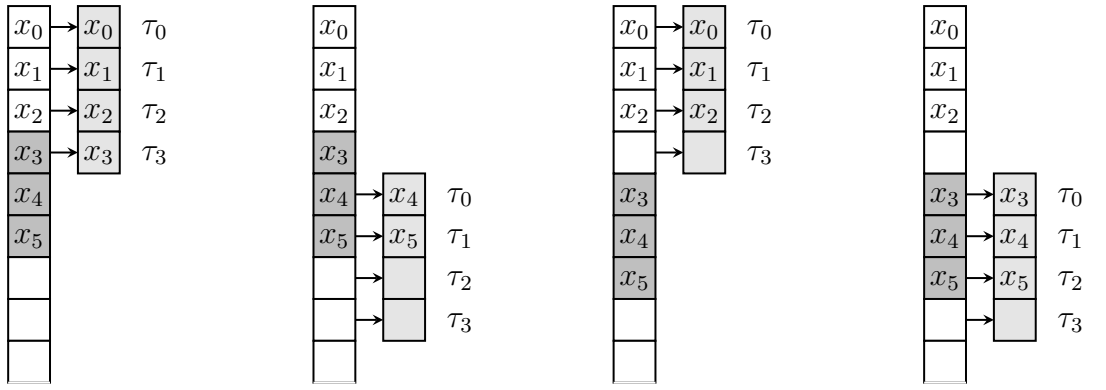
$X^j \leftarrow X^j + x_{\frac{N}{2}\ldots N} \cdot \omega_2^j$

$X^j \leftarrow X^j \cdot \omega_N^{j\cdot 0\ldots\frac{N}{2}}$

This can be broken apart to create the instructions a thread would execute. In turn, thread IDs can be used for this purpose. For loops or similar instructions are required to loop over $0 \ldots \frac{N}{2}$. Branching instructions can hurt run time if the threads within a warp follow a different execution path, as execution will be serialized in this case.

Data alignment as been discussed in Section 3.2.2. Again we use a warp size of 4 and a 16-byte cache line. When data is not aligned in favor of cache lines we get non-coalesced access. Not only that, we also need a branch instruction to stop over-fetching and we also miss align threads $T = \{\tau_0, \tau_1, \tau_2, \tau_3\}$. This is illustrated in Figure 3.11 on input data of size $N = 6$ on which we perform radix-2 (at the first step).

Figure 3.11 Aligning data and threads



(a) Not aligned 1 **(b)** Not aligned 2 **(c)** Aligned 1 **(d)** Aligned 2

With the aligned data we can use thread τ_0 to fulfill the data dependency between elements x_0 and x_3, and thread τ_j for elements x_j and $x_{j+\frac{N}{2}}$. In the end, we discard the value τ_3 has calculated. Aligning data thus adds simplicity and reduces branch instructions and register usage. As a result of this approach, the number of threads we use must be a multiple of a cache line size.

As an example of seemingly hidden ILP, we have added Algorithm 3.4. In a previous state, we have stored values from global or shared memory into the register array R. For an array to comprise of register, the indexing must be done by constant values or must not depend on variables determined at run time. Illustrated is a radix-2 butterfly where the twiddle factors have been pre-calculated and stored in constant memory (Wr and Wi). The run time should be the same for each algorithm, but the algorithms on the right is around **20**% faster. The underlying assembly must be different, thus always be on the lookout for hidden instruction-level-parallelism.

Algorithm 3.4 ILP radix-2 butterfly

```
// ... some computations
for(int p = 0; p < 2; p++) {
    real Rr = 0;
    real Ri = 0;

    for(int i = 0; i < 2; i++) {
        int ind = (p*i)%2;
        Rr += R[i][0] * Wr[ind][2]
           - R[i][1] * Wi[ind][2];
        Ri += R[i][0] * Wi[ind][2]
           + R[i][1] * Wr[ind][2];
    }
    // ... some computations
    // ... write back Rr and Ri
}
```

```
// ... some computations
for(int p = 0; p < 2; p++) {
    real Rr = 0;
    real Ri = 0;

    for(int i = 0; i < 2; i++) {
        int ind = (p*i)%2;
        Rr += R[i][0] * Wr[ind][2];
        Rr -= R[i][1] * Wi[ind][2];
        Ri += R[i][0] * Wi[ind][2];
        Ri += R[i][1] * Wr[ind][2];
    }
    // ... some computations
    // ... write back Rr and Ri
}
```

3.2.5 Solving DFTs on a Streaming Multiprocessor

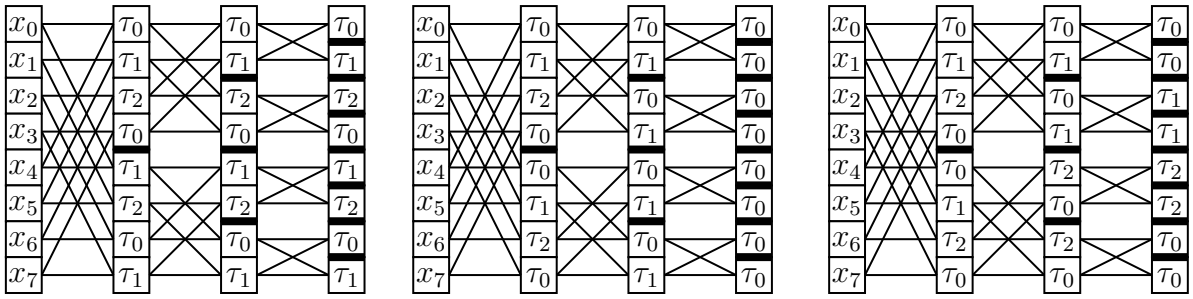
Like a divide-and-conquer strategy, we create sub-DFTs from a DFT with one or multiple steps of a FFT algorithm as we discussed in the previous sections. Typically, multiple different butterflies are needed when continuing to solve the sub-DFTs with the FFT algorithm. One of the most restricting factors on the number of threads you can run concurrently is the number of registers they require. The more complex and comprehensive CUDA functions or *kernels* become, the more registers they require to run. It is for this reason that we need to keep things simple. An alternative would be to launch CUDA kernels sequentially, and divide function operations among them. Spread out the instructions as it were. However, this is a costly thing to do. There is an initialization cost when launching kernels and synchronization is required between them to ensure one is finished when another begins. Kernels can run concurrently, but they must then share resources, defeating the purpose

of creating multiple kernels in the first place. A solution to keeping kernels simple is to take advantage of compile time optimizations. If we know the problem size, we can simply produce the right butterflies and put them in the kernel. We can then maximize the number of threads that will execute the kernel and thus maximize concurrency. Note that a higher radix uses more registers. The number of registers used by a kernel is determined by the function with the highest register count. The weakest link in the chain, sort of speak. Lower radixes are then run with less threads as you normally would. To put it in perspective, each SM can run 1536 threads and has 32K 32-bit registers. Each thread would only be able to make use of $\frac{32 \cdot 1000}{1536} \approx 21$ registers when running at full occupancy, 31 register at $\frac{2}{3}$ occupancy and 63 registers (the maximum, see table 3.1) at $\frac{1}{3}$ occupancy.

Global memory access has been optimized in Section 3.2.2, we now look at shared memory access on SMs. In particular when a DFT is smaller than the shared memory available, in which case no communication between SMs is required to complete the FFT algorithm. When dealing with 2-dimensional input data, we perform 1-dimensional FFTs along each dimension. To optimize data accesses, we must transpose the matrix when switching from one dimension to the next. Optimizing the transpose with CUDA programming is discussed in [19]. It also shows how padding data can increase shared memory bandwidth in case where it reduces bank conflicts. Let M be a 32x32 matrix which is stored such that consecutive row elements are stored in consecutive banks. Remember we have 32 banks. When a column of M is accessed, we access different data elements within the same bank, resulting in conflicts. The accesses are serialized as a result. By padding M to be a 32x33 matrix, accesses will be spread out over all available banks as element (0,0) will be in bank 0, element (1,0) will be in bank 1, element (2,0) will be in bank 2, and so on.

Two values need to be combined in a radix-2 butterfly. Processing the two values must therefore be done by one thread. Figure 3.12 shows three thread configuration propositions.

Figure 3.12 Thread configuration

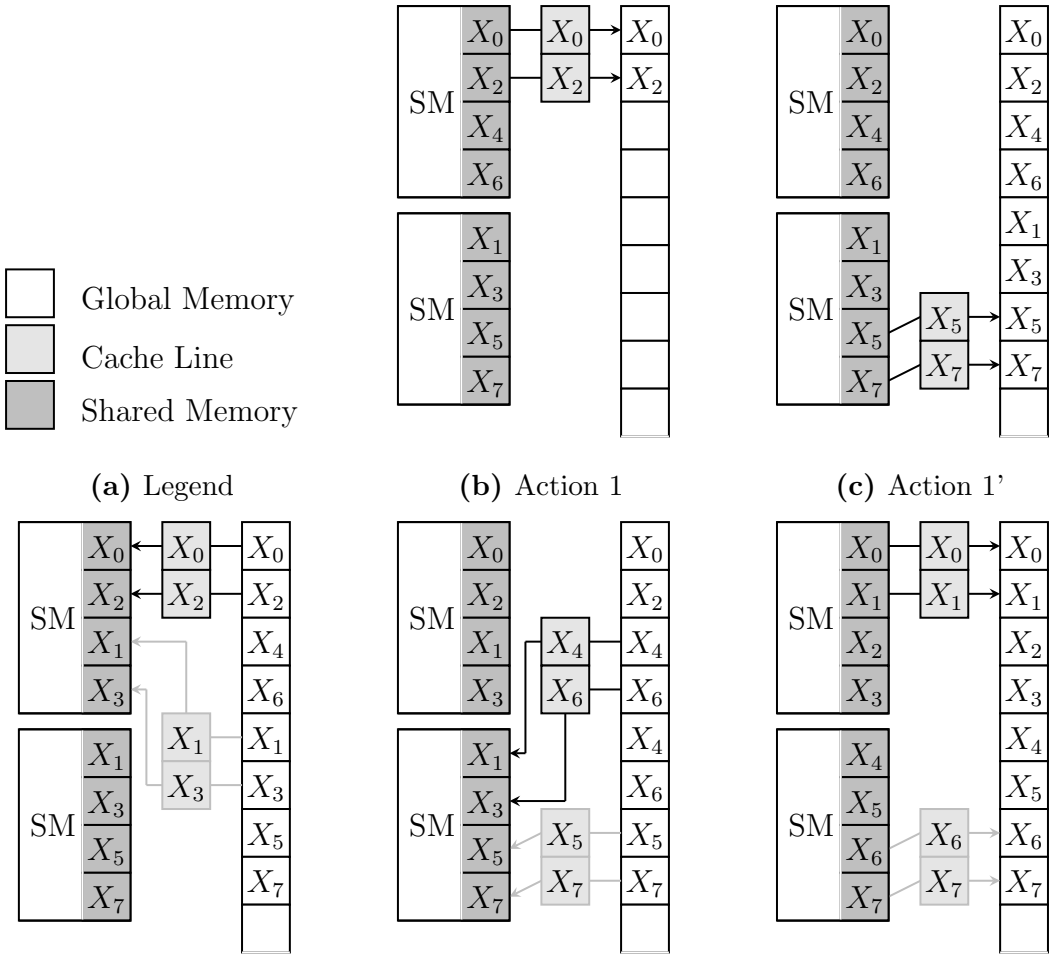


Let a warp contain 3 threads, exactly the number of threads (τ_0, τ_1, τ_2) we use to execute the algorithm. We choose an awkward number of threads compared to the input data size, as this situation will likely occur in practice. DFTs are separated by thick lines. Figure 3.12a will loop threads over the data, calculating distributed butterflies. Each thread calculates a part of the radix-2 butterfly, requiring twice the number of data accesses. This grows linearly as the radix gets higher. We need three warps to execute, and we have one idle thread in the final warp. Figure 3.12b puts thread 0 at the start of every DFT. Each full butterfly is calculated by one thread. Data is read only one time from shared memory, but as DFTs decrease in size, the number of warps required increases. Figure 3.12c mixes the previous two and loops the threads with on restriction. A butterfly must be calculated by one thread. As can be seen, *always* avoiding bank conflicts is not trivial and perhaps impossible. We therefore leave this as future work. Two factors that way in on bank conflict reduction is the number of threads chosen to execute the algorithm (yet we have little room to experiment as we need to keep the warp size in mind along with all other restricting factors previously discussed) or data padding. To compare, Figure 3.12a always has full shared memory bandwidth, but needs more warps and more data accesses. Figure 3.12c needs the fewest warps, the least data accesses, but eventually produces bank conflicts and serialized accesses. As we only have 32 banks and are working with much larger data sets, the conflicts occur at a very late and short state. Figure 3.12c would be the preferred method with low radixes as computations in comparison to accesses are low.

3.2.6 Data Output

As seen in Section 2.2.2, the output of a DIF FFT with naturally ordered input is interleaved after one step. The interleaving of data can be prevented, but this requires an additional buffer of which the size is equal to the input size and it requires the use of shared memory to do coalesced global memory accesses. We take advantage of the shared memory bank system to first order the data residing on each SM after all steps have finished. We then have interleaved output that needs to be written back to global memory before it leaves the GPU. The first step of the algorithm has determined how interleaved this output is, i.e. radix-r will result in r partitions of interleaved data and writing it back without extra measure results in stride-r access pattern and thus a r times slower write-back than optimal. We take radix-2 as an example in Figure 3.13 and reduce the size of a cache line for demonstration purposes to 8 bytes.

Figure 3.13 Interleaved radix-2 output write-back



(a) Legend **(b)** Action 1 **(c)** Action 1'

(d) Action 2 **(e)** Action 2' **(f)** Action 3

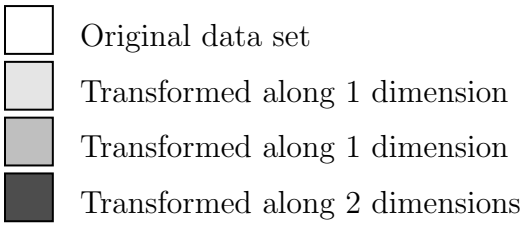
We first write the interleaved output to global memory coalesced with *action 1*. Note that $\frac{N}{r^2}$ elements per SM are copied back and forth which can be avoided. *Action 2* then copies elements to their respective SM, also coalesced. Before *action 3*, we sort the elements with the advantage of using shared memory and finally write them to global memory.

3.2.7 Multi-GPU FFT

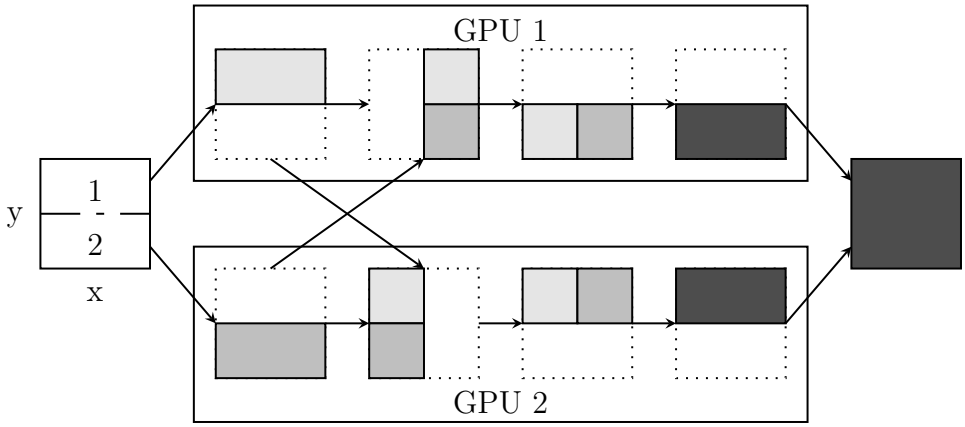
Let us consider the case using 2 GPUs and 2-dimensional input data. Figure 3.14b would be the standard way of dividing data among GPUs. Each would calculate the FFT along the x-axis of half the data. Then, the GPUs need to be synchronized after which they must share data. Depending on the architecture, this can be very costly. Think for instance of a setup where GPUs are connected to different hosts. The cost of inter GPU data sharing goes up as inter host data sharing is required to do so. The remainder is quite simple as the transformed data is transposed for access efficiency reasons and the transformations are done along the second dimension.

Of total run time, 38.8% is spend on inter GPU communication according to the implementation presented in [7]. This cost was reduced by the use of multiple data streams, proposed by [21]. The method introduced by Figure 3.9 can be used the perform a multi-GPU FFT as illustrated by Figure 3.14b. Each GPU would produce interleaved output which thus needs to be combined at a later stage. This can be prevented. Lets consider a one GPU implementation and a radix-2 butterfly. Thread 0 would produce the outputs X_0 and X_1, which would be stored at index 0 and $\frac{N}{2}$. Instead, these must be stored at index 0 and 1. In order to do so, we must go through shared memory (in order to write coalesced to global memory) and we must use an additional buffer at least the size of the currently being processed FFT (to prevent writing to index 1 by thread 0 before it is read by thread 1). Immediately visible in Figure 3.14 is the absence of inter GPU data sharing. The additional computation time required calculating a distributed butterfly outweighs the cost of global synchronization and data sharing, as specially when communication cost is high. This scenario would not work for single GPU implementations as the communication cost between SMs through the L2 cache is too low.

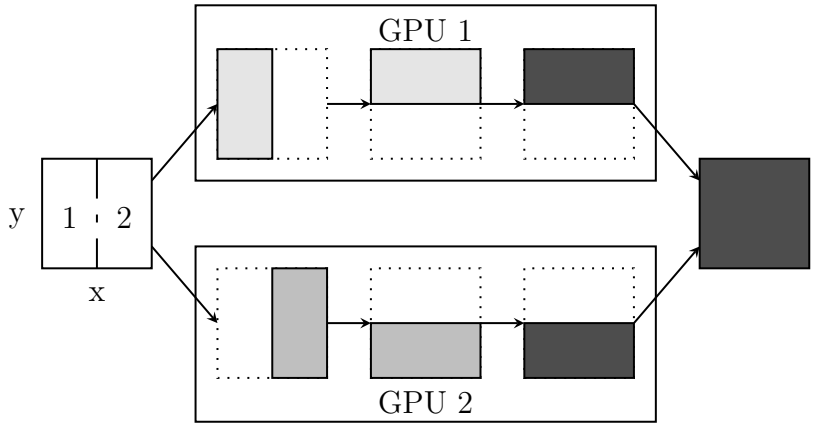
Figure 3.14 Multi-GPU FFT approaches



(a) Legend



(b) Normal butterflies



(c) Distributed butterflies

Chapter 4

Results

This chapter tries to give an insight into the approaches discussed in Section 3.2. The theoretical gain is discussed with a low level explanation in Section 4.1. We go one to show the comparison between some well known algorithms in Section 4.2. For testing we have used NVIDIA GTX 480 graphics cards. It has 16 streaming multiprocessors (SMs) of which 15 can be used by a CUDA program.

We start with the discussion on theoretical gain of using higher radix butterflies in Section 4.1, based on the knowledge of NVIDIA documentation. We then compare our work to some well known algorithms in Section 4.2. Future work is discussed in Section 4.3, followed by the conclusion in Section 4.4.

4.1 Theoretical Gain

NVIDIA does not disclose on many architectural details and also hides the operations that are truly executed when running a CUDA program. Even NVIDIA's (pseudo-)assembly PTX is not the actual assembly that is run on the GPU. Access times and access policies are also barely discussed. Thus ultimately, many programs must be written to thoroughly map the behavior of the architecture. Table 4.1 is the result of the attempt to do so. The programs resulting in these values have been written to the best of our knowledge and might not be totally accurate and must therefore only be used as an indication. We can see that the L2 cache is approximately 4x faster than DRAM (L2 cache miss). The L2 cache on the Kepler GK110 offers up to 2x of the bandwidth per clock cycle available in Fermi

and it is twice as large[6]. This would likely result in a higher performance regarding the FFT algorithm.

Table 4.1 GForce GTX 480 warp data access times

Strategy	Clock cycles
L1 hit	18
L2 hit	248
atomic operation L2 hit	580
atomic operation L2 hit 2x conflict	590
atomic operation L2 hit 4x conflict	600
L2 miss	1060
atomic operation L2 hit 8x conflict	1140
atomic operation L2 miss	1360
atomic operation L2 miss 2x conflict	1380
atomic operation L2 miss 4x conflict	1490
atomic operation L2 miss 8x conflict	1900
atomic operation L2 hit 16x conflict	2200
atomic operation L2 miss 16x conflict	2980
atomic operation L2 hit 32x conflict	3910
atomic operation L2 miss 32x conflict	4680

We will first look at the costs of two butterflies and compare them to data access times. The method described by Figure 3.10 which applies a step of a DIF radix-r FFT will be used to compared a four step DIF radix-2 FFT and a one step DIF radix-16 FFT, because that would result in the same amount ans same sized DFTs. Table 4.2 holds the cost of several operations according to [3], which we will use in the comparison.

Table 4.2 Operation cost compute capability 2.0

Operation	Operations per clock cycle per SM	Clock cycles per warp
single precision floating point addition, subtraction and multiply	32	1
32-bit integer addition, subtraction and compare	32	1
32-bit integer multiply	16	2
32-bit floating point sine and cosine	4	8

This is of course not accurate as we have influential factors like back-to-back register dependencies taking 22 clock cycles[3] and access latencies that possibly can be hidden by other calculations and pipe-lining given the right conditions, but we will use it as an indication.

It takes 23808 clock cycles to process all optimized radix-2 butterflies in four steps. A radix-16 step without pre-calculating twiddle factors leads to 88704 clock cycles. Pre-calculating twiddle factors costs to 63954 clock cycles, which is a considerable reduction by **28%**. The higher radix butterfly is favored by roughly **57%** when adding the cost of global memory access time previously calculated. When the ratio of the number of instructions with no off-chip memory operands to the number of instruction with off-chip memory operands is low, more warps are required to hide data access latency[3]. When this ratio is 30, 20 warps are required to hide all access latency ($\frac{48}{20}$ occupancy). The radix-16 approach has a **84%** better overlap than the radix-2 approach. Using a higher radix thus improves the overlap between data access and computation, and because access latency is poorly hidden in the first place, it should yield a performance improvement.

4.2 Algorithm Comparison

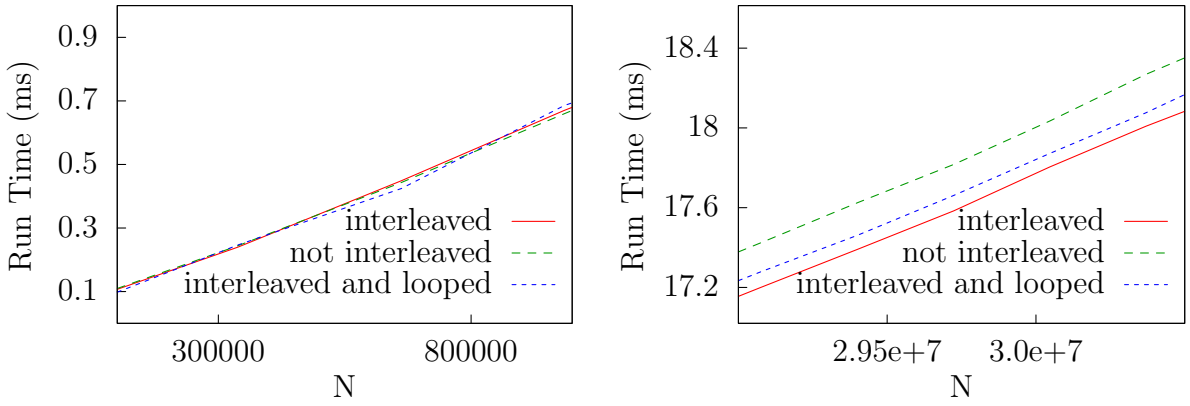
The graphics card in the experiments is represented by the NVIDIA GTX 480 (177.4 GB/s bandwidth) with CUDA Toolkit 5.0 and compute capability 2.0. The CPU is represented by an Intel Xeon E5620 (25.6 GB/s bandwidth).

We start with experiments for the elemental components of a GPU FFT implementation in Section 4.2.1. Section 4.2.2 will make an analysis of two well known FFT implementations and compare them with our implementation.

4.2.1 Elemental Components

Important parts in every kernel are the synchronization method and access patterns. Table 4.1 focuses on the later. We have used a radix-2 butterfly and tried three different access methods. All patterns access global memory in a coalesced fashion. Say we run the kernel with 512 threads and 15 blocks. *Interleaved* refers to the access pattern where the first 512 values (of each sub-DFT) are accessed by the first block, the second 512 values are accessed by the second block, etc. *Not interleaved* is when the first block accesses the first 15^{th} of the data, the second block accesses the second 15^{th} of the data, etc. *Interleaved and looped* is the same as interleaved, but the start of each sub-DFT does not have to be accessed by the first block. It thereby divides the work better among multiple blocks when sub-DFTs get smaller.

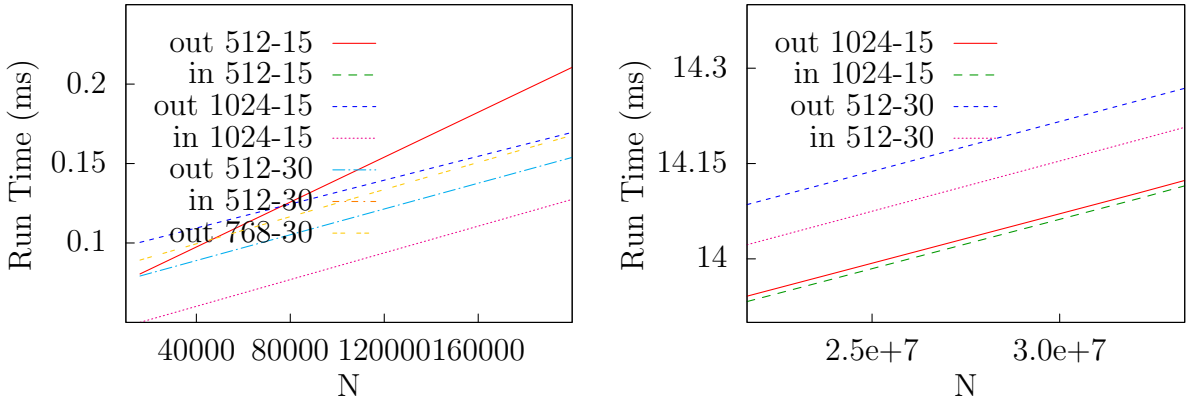
Figure 4.1 Access patterns comparison



At the beginning we see little difference between the approaches. Though, as the sub-DFT sizes get smaller *interleaved and looped* is a preferred method. Yet, due to additional control code, this method is slower otherwise. We see that *interleaved* is better in the long run. We will use this approach in further implementations.

We have tried to map synchronization behavior with the radix-2 kernel which runs at 21 registers when optimized. Table 4.2 shows the results. Each example consists of a keyword *in* or *out*, which refers to synchronization being done inside or outside the kernel, and 2 numbers representing the kernel launch configuration. The first number equals the number of threads used and the second equals the number of block used.

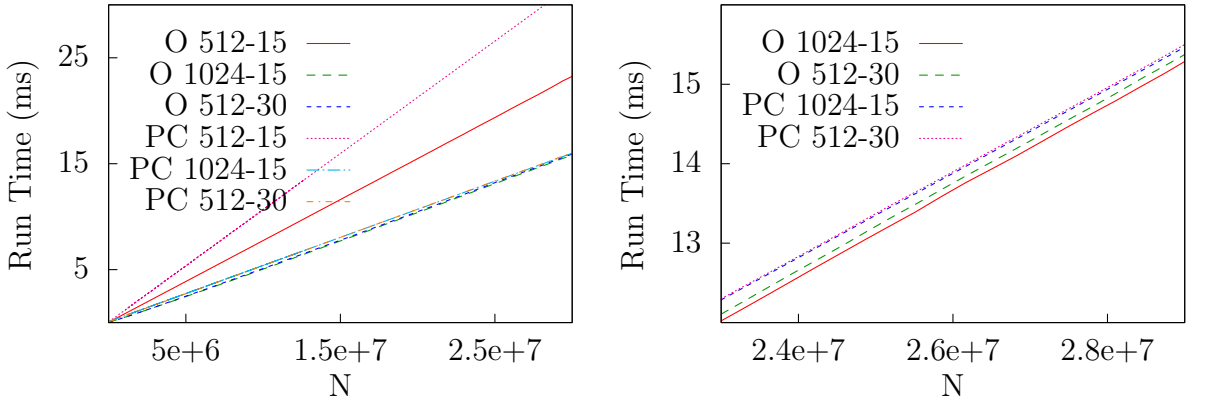
Figure 4.2 Optimized radix-2 kernel synchronization cost



When occupancy is equal (the kernels are configured to launch with the same number of threads) we see that synchronization is preferred inside the kernel. As the global synchronization function used for this experiment is very basis, perhaps some adjustments can be made to further optimize the run time.

Lets now consider at different butterfly implementations. *Pre-calculated* (PC) substitutes nearly all trigonometric function calls with constant memory calls. *Optimized* (O) is when the trigonometric function calls are substituted by floating point values, and when a 0, -1 or 1 is encountered the values are optimized out. This approach results in the Gentleman-Sande butterfly shown in Figure 2.7. Butterflies have also been developed for purely *real* (R) input values, which was shown in Table 3.2. The only kernel that can achieve full occupancy is the optimized radix-2 kernel. As many real numbers are inserted in the code when the radix goes up (including control code), this method is slow for higher radix as it consumes too many registers resulting in register spilling. The higher radix achieved without spilling is the pre-calculated radix-16, running at 63 registers. Table 4.3 compares a radix-2 butterfly at different occupancies. Higher compute capability graphics cards allow kernels to use more registers. Future work would include further increasing radix and see where the threshold lies when the use of a higher radix is less beneficial.

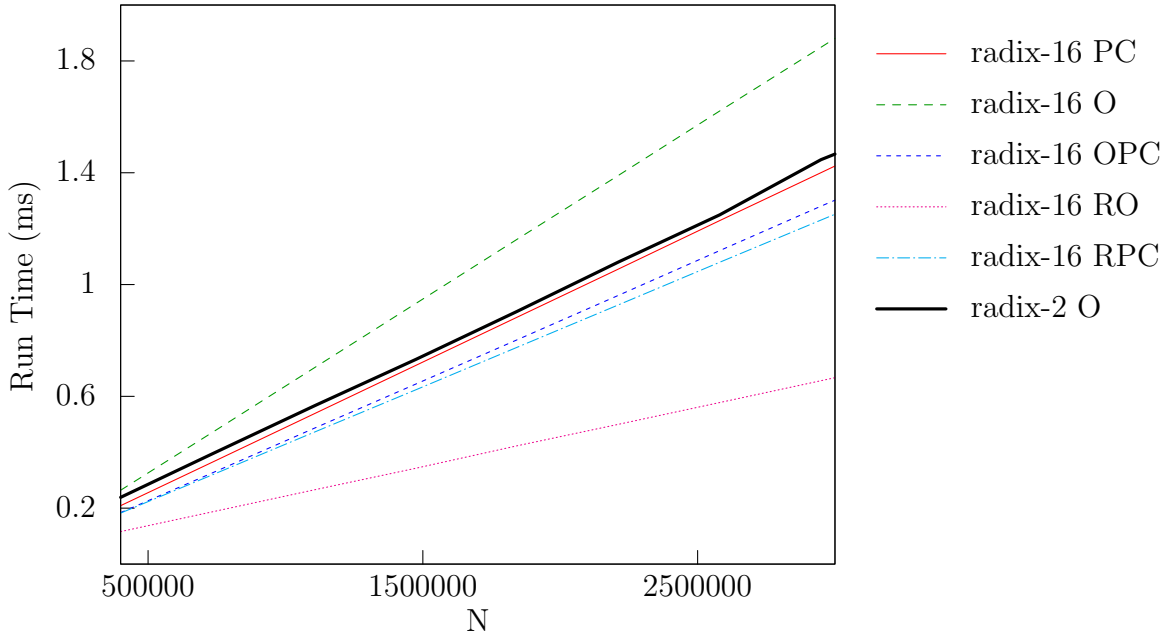
Figure 4.3 Radix-2 pre-calculated vs optimized



We see in the long run that the optimized version beats the pre-calculated version, but

barely. It is an indication that the SMs are not running at maximum throughput. The configuration missing is the full occupancy radix-2 butterfly which has the best run time and will therefore be used to test and see if a higher radix is indeed faster. This is done in Figure 4.4. Data sizes have been chosen to be as fair as possible, i.e., every SM, block and thread does the same amount work and all accesses are coalesced. The comparison is between four radix-2 steps and one radix-16 step, as each creates the same sized sub-DFTs in the end.

Figure 4.4 radix-2 head to head with radix-16



The optimized radix-16 butterfly is much slower than the radix-2 butterfly. The previous discussion on optimized butterflies should be enough to explain why this butterfly uses too many register and that its run time is influenced by register spilling. The interesting butterflies are radix-16 PC and OPC as they run the full calculation and beat the full occupancy radix-2 butterfly at only $\frac{1}{3}$ occupancy. The performance improvement can be explained by the following: the radix-2 kernel must access global memory more times and the SMs are not running at max throughput. Results have been put together in Table 4.3

Table 4.3 Performance increase radix-16 vs radix-2, in percentages

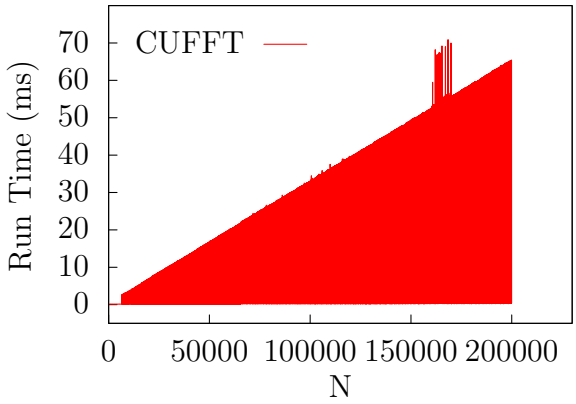
Radix-16 butterfly	AVG	MAX
Pre-calculated	4.74	15.29
Optimized and pre-calculated	15.86	31.35
Real input optimized	113.95	120.49
Real input optimized and pre-calculated	19.38	32.05

NVIDIA's FFT implementation CUFFT is optimized for data sizes of the form $2^a \cdot 3^b \cdot 5^c \cdot 7^d$[5]. Figure 4.4 suggests that this form must be extended to $2^a \cdot 3^b \cdot 5^c \cdot 7^d \cdot 11^e \cdot 13^f$ as higher radixes are less costly (for big data input sizes which don't allow L2 cache optimizations).

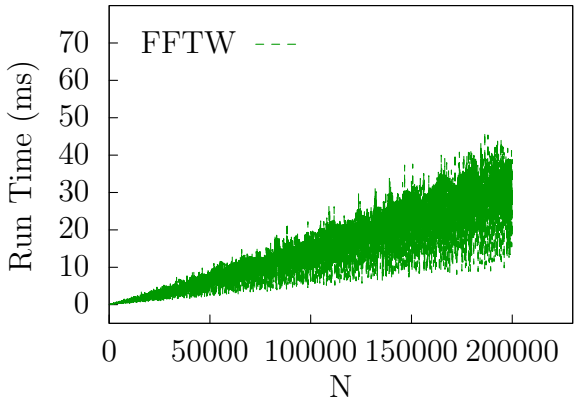
4.2.2 The Giants

The most well known FFT algorithms out there are FFTW and CUFFT. We have used FFTW 3.3.2 and CUFFT from CUDA Toolkit 5.0. Figure 4.5 shows the run time of the preparation and calculation of the transform by the two algorithms.

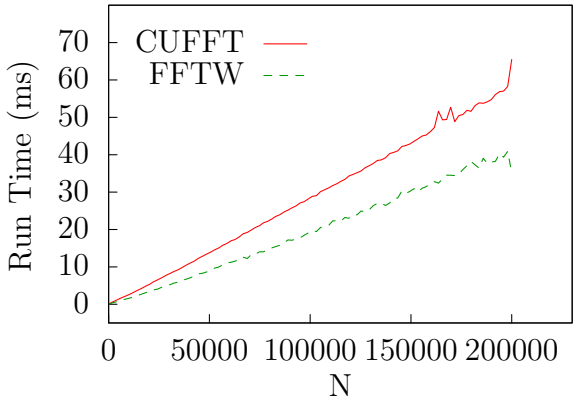
Figure 4.5 CUFFT and FFTW comparison



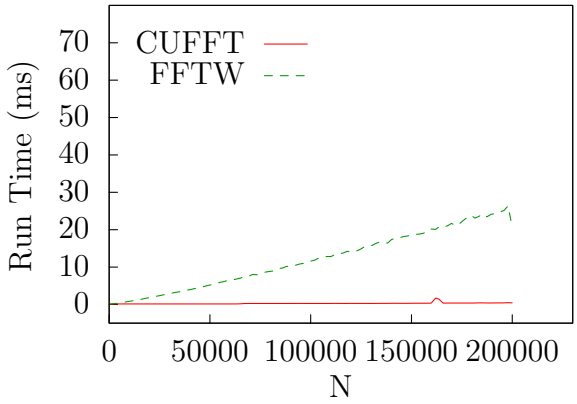
(a) CUFFT



(b) FFTW



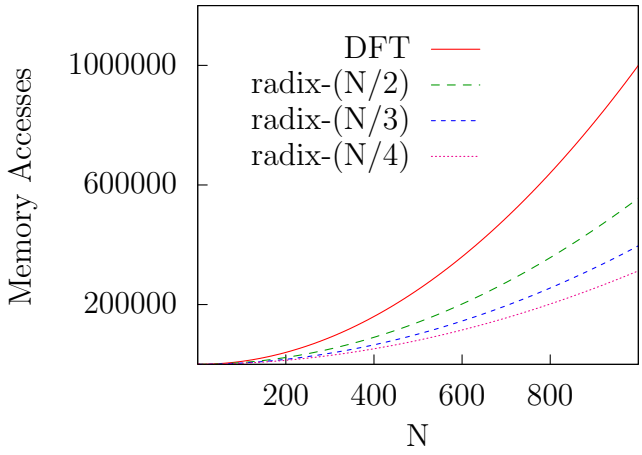
(c) Bezier curve



(d) Bezier curve of points below original berzier curve

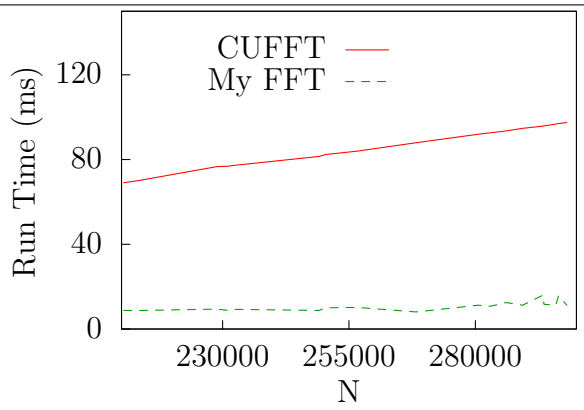
Surprisingly, we see that CUFFT performs worse than FFTW on average. We do see that CUFFT is the fastest in particular data sizes. Yet out of all tested data sizes only **14.67**% performs well. This means that **85.33**% performs poorly and only **10.15**% of these data sizes are primes. Figure 4.6 shows how run times should look like in terms of data accesses when a DFT was used.

Figure 4.6 Memory accesses compared to radix used



From this we conclude that a DFT was not used to calculated the bad performing data sizes. CUFFT uses an implementation of *Bluestein's* algorithm which can calculate a FFT of arbitrary data size with complexity $\mathcal{O}(n \log n)$[23]. There are many data sizes that can be calculated faster. Lets consider size $N = 210432$. CUFFT's run time is 69.04 ms using Bluestein. It can also be calculated with the following decimation step sequence: 2-2-2-2-2-2-2-2-2-3-137, resulting in a lower run time. Some of these data sizes are shown in Figure 4.7. Mind that the trailing radix-137 decimation has been calculated using a simple DFT implementation and is responsible for about 94% of the total run time. This can be further optimized.

Figure 4.7 My fft vs CUFFT

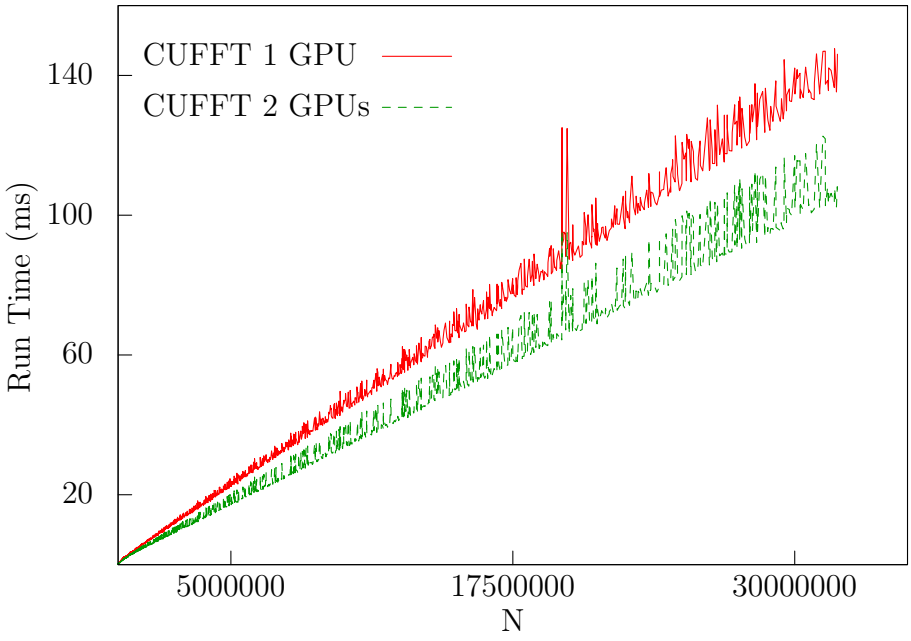


In the end the same conclusions can be taken as the developers of CUFFT have, with regard to their faster run times. All tests and implementations have lead to comparable run times, which are therefore not shown. Some contributing factors in developing an optimized FFT implementation are: (1) move as much of the computation as possible to registers; (2) do more work per thread to increase bandwidth and throughput at low occupancy; (3) put each butterfly in its own kernel, it must otherwise run at a lower occupancy than it normally would; (4) have separate kernels for aligned and misaligned global memory data; (5) Use synchronization within the kernel as much as possible, still obeying (3) and (4); (6) use a different kernel for different sub-DFT sizes. Initially large FFTs must be handled with a breath-first approach. As soon as a sub-DFT becomes smaller than the L2 cache size switch to depth-first, as the L2 cache would optimally be re-used. This again applies to sub-DFT sizes of less than the shared memory on a SM, allowing to do the remaining calculations only using shared memory.

There are exceptions. One could think about putting multiple butterflies in one kernel, such that when sub-DFT sizes decrease to fit in shared memory, on could continue many steps while only using shared memory. Ultimately, each data size has an optimal solution.

Up until now we have seen the behavior of CUFFT running on one GPU. Figure 4.8 compares CUFFT running on one and two GPUs. CUFFT can run on multiple GPUs with the method described in Section 3.2.7. If computation is long enough, using multiple GPUs will yield shorter run times. To make a fair comparison we have chosen input sizes of the form: (1) it conforms to CUFFTs recommended input form $2^a \cdot 3^b \cdot 5^c \cdot 7^d$; (2) all input sizes must be dividable by 2; (3) no bigger than 32 million points. We will be using 2 GPUs, thus the input size must be dividable by 2 and if the by CUFFT fastest calculated input sizes yield shorter run times, then other input sizes will yield shorter run times as they take longer to compute.

Figure 4.8 Using multiple GPUs



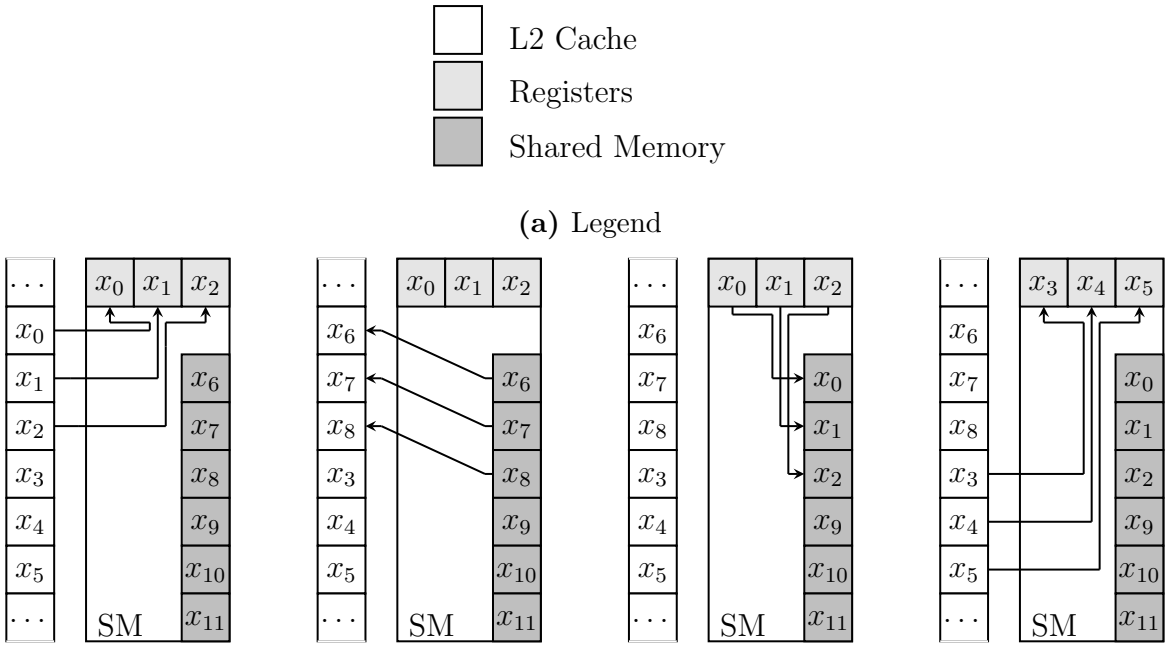
Very small data sizes perform better on 1 GPU. As the size grows we see that the use of 2 GPU leads to a performance increase. The maximum performance gain over all calculated data sizes is **1.39x**, and **1.23x** on average. This means that other data sizes have even higher gains using multiple GPUs.

4.3 Future Work

A trend has been set where CPU and GPU architectures grow towards each other. CPUs gain more cores and GPUs have a full hierarchical caching mechanism. Global memory accesses that resort to DRAM on a L2 cache miss are the single most costly operations on the GPU. Reducing them is key in performance. One way to do so is choosing a higher radix at each step. Using a higher radix increases the computational complexity but reduces the number of steps required to complete the FFT algorithm and thus reducing the number of access to global memory. Increasing radix has a limit of course, as the cost of higher computation complexity stops outweighing the benefit of reduced global memory accesses at some point.

Another way to reduces cache misses is to move computation away from DRAM as much as possible, and concentrate efforts more toward faster memories like L1, L2, shared memory and registers. At each step of a FFT we get smaller DFTs. If such a DFT can fit into L2, there is no need to write back to DRAM for the remainder of the algorithm for that DFT, then continuing on to the next DFT like depth-first approach. Obviously the more non-DRAM memory we have, the less we need to access DRAM. We can extend L2 with shared memory and write back and forth between them with registers. We only use a few register for this so we can keep thread occupancy as high as possible. This is shown in Figure 4.9.

Figure 4.9 L2 cache extension



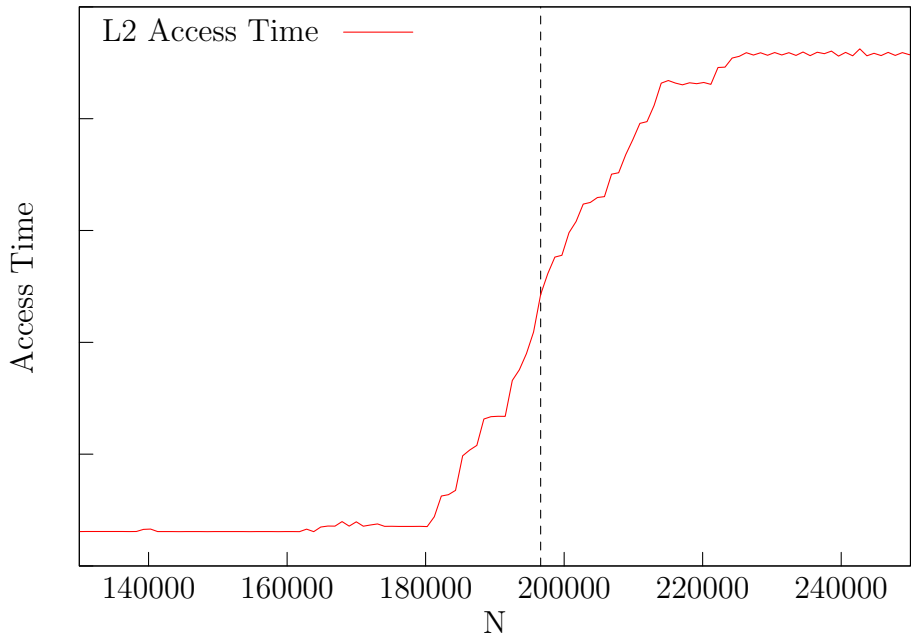
(a) Legend **(b)** Action 1 **(c)** Action 2 **(d)** Action 3 **(e)** Action 4

This continues until all elements have been swapped. There are operations not visible in Figure 4.9. Lets assume a 12-byte cache line (three values) for the previous example. One cannot simply write to specific locations in the L2 cache. Old cache lines get evicted and are replaced by new ones. By normal behavior the approach would not work: *action 1* accesses x_0 though x_2 and will therefore not be evicted when writing x_6 through x_8. In stead, x_3 through x_5 will be evicted, resulting in cache misses in the future. To fix this, we need to use two eviction policies at the same time. *Parallel Thread Execution* (PTX) is the assembly version for NVIDIA's CUDA. It allows the programmer to have more control. PTX can be generated by passing the `ptx` flag to NVIDIA's `nvcc` compiler, allowing the programmer to see what optimizations the compiler has done. The caching policy can be chosen at compile time and is used throughout execution. PTX allows to change this policy on a per access basis. This can be done by writing inline-PTX assembly straight into the CUDA C program[1]. In our case we need to set the *evict-first* flag as *action 1* ensuring the right cache line gets evicted when writing to L2. This then also works when SMs are

not synchronized. In the case though where the dirty bit is set, data needs be written back to DRAM. This can be avoided by using the same mechanism, except we won't swap the data, but only the values.

If a data size is chosen less than the L2 size, this should mean that whatever radix we use, we only resort to DRAM once, and use the L2 cache for the remainder of the computations (between steps). The technique just described and the technique described by Figure 3.10 depend on this information. *NVIDIA Visual Profiler* (NVVP) was used to investigate the amount of L2 cache misses. From the portion of the computation that assumes the get only cache hits, actually got 89% cache misses. Figure 4.10 shows access times increase due to cache misses much earlier than the L2 cache size of 768KB. This means other data is stored on the L2 cache than just our array or we are not allowed to use the full L2 cache.

Figure 4.10 L2 cache hit analysis



The dashed line indicates the size of the L2 cache. N represents the array size consisting of 4-byte elements and access time refers to average access time of a warp. We see access time go up at around 180000 (indicating cache misses), which is less than the size of the

L2 cache. Choosing data sizes greater than this would therefore result in a misalignment of data regarding the FFT algorithm: the first element has been evicted by the time the last will be written. When the first element is accessed at the next step, it will result in a cache miss. This behavior goes on like a cascading affect, explaining the high miss rate. We have much less cache control than we would have on a CPU, yet each new graphics card has bigger and faster caches making it more and more important. The ability to perform page locking or in this case cache line locking for instance, would lead to performance improvements for certain applications. Unfortunately, cache locking is not supported by PTX. Yet if it were, ignoring this threshold would likely lead to a decrease in performance as the unaccounted space is probably used by the 16^{th} SM to manage the others and is therefore necessary.

The approach described by Figure 3.9 would not be beneficial on a single GPU. When calculating distributed butterflies, $(r-1)N$ L2 cache accesses are required when performing a radix-r step. We have previously concluded that data accesses are by far the most time consuming operations when calculating a butterfly. The L2 cache is not nearly fast enough to compete with serialized computations. This should therefore be used in a multi-GPU situation, as relatively extremely expensive communication between GPUs is completely eliminated.

Many CPU algorithms have been optimized in favor of caches. As GPU caches grow larger, so does their importance in optimization. The algorithm has the characteristic that by increasing computational complexity, the number of data access to global memory required goes down. NVIDIA provides hints on how to optimize performance of CUDA applications. These hints mainly regard access latency hiding by computation. In data access intensive applications this can be difficult or even impossible. Improving the use of caches is beneficial in these cases as access time drops and thereby influences total run time. The FFT algorithm is an example where every data element is manipulated multiple times,

increasing the usefulness of caching. When GPU caching capabilities increase further as they have until now, we can start with a whole new series of optimizations as we have seen with the CPU.

4.4 Conclusion

Applications can be optimized by improving computational complexity and/or rely on the exploitation of hardware architectures. With the arrival of programmable graphics cards, many applications are left to be optimized. The Fast Fourier Transform has seen several run time improvements since the use of GPUs. We have seen that the use of a higher radix butterfly, although increasing computational complexity, reduces run time compared to a lower radix butterfly. CUFFT performs well on data input sizes of the form $2^a \cdot 3^b \cdot 5^c \cdot 7^d$. Results suggests that performance is increased when this is extended to the form $2^a \cdot 3^b \cdot 5^c \cdot 7^d \cdot 11^e \cdot 13^f$ for large data sizes.

In the relatively small field of multi-GPU FFT implementations, [7] and [21] have optimized inter GPU communication. We have proposed a multi-GPU FFT implementation without inter GPU communication nor synchronization and have seen performance increases of up to 1.39x using 2 GPUs, compared to CUFFT's fastest run times using only 1 GPU. With regard to caching, it seems that the CUDA platform is not yet mature enough for optimizations at the level we have seen on the CPU. We have proposed a method that simulates cache line locking. There is a relatively small window in which this is applicable in the FFT algorithm but it does produce a performance increase when used right.

Run times of CUFFT have been analyzed, which seems to perform considerably worse when their recommended input form is not met. We have taken a few bad performing non prime data input sizes and improved their run time. Looking at the amount of different optimizations and launch configurations we conclude that ultimately, there is an optimal solution for each FFT input size.

References

[1] Using inline ptx assembly in cuda, 2011. http://mlso.hao.ucar.edu/ hao/acos/sw/cuda/doc/Using_Inline_PTX_Assembly_In_CUDA.pdf.

[2] Cuda c best practices guide, 2012. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html.

[3] Cuda c programming guide, 2012. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[4] Nvidia's fermi: The first complete gpu computing architecture, 2012. http://www.nvidia.com/object/fermi-architecture.html.

[5] Nvidia's fft implementation cufft, 2012. http://docs.nvidia.com/cuda/cufft/index.html.

[6] Nvidia's next generation cuda compute architecture: Kepler gk110, 2012.

[7] Y. Chen, X. Cui, and H. Mei. Large-scale fft on gpu clusters. *ICS '10 Proceedings of the 24th ACM International Conference on Supercomputing*, 2010.

[8] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Co imputation*, 19:297–301, 1965.

[9] P. Duhamel and H. Hollmann. Split radix fft algorithm. *Electronics Letters*, 20:14–16, 1984.

[10] M. Frigo and S. G. Johnson. Fftw: An adaptive software architecture for the fft. *In Proceedings of 1998 IEEE International Conference Acoustics Speech and Signal Processing*, 3:1381–1384, 1998.

[11] W. M. Gentleman and G. Sande. Fast fourier transforms: for fun and profit. *Proceeding AFIPS '66 (Fall) Proceedings of the November 7-10, fall joint computer conference*, pages 563–578, 1966.

[12] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High performance discrete fourier transforms on graphics processors. *SC '08 Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[13] Z. Lili, S. Zhang, M. Zhang, and Z. Yi. Streaming fft asynchronously on graphics processor units. *International Forum on Information Technology and Applications (IFITA)*, 2010.

[14] C. V. Loan. *Computational Framework for the Fast Fourier Transform.* Society for Industrial and Applied Mathematics PhiladelSociety for Industrial and Applied Mathematics Philadelphiaphia, 1992.

[15] J. Lobeiras, M. Amor, and R. Doallo. Fft implementation on a streaming architecture. *Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on Parallel*, 2011.

[16] A. Nukada. Bandwidth intensive 3d fft kernel for gpus using cuda. *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2008.

[17] A. Nukada and S. Matsouka. Auto-tuning 3-d fft library for cuda gpus. *ICS '10 Proceedings of the 24th ACM International Conference on Supercomputing*, 2010.

[18] W. Rudin. *Real and Complex Analysis.* Tata McGraw-Hill, 3rd edition edition, 2006.

[19] G. Ruetsch and P. Micikevicius. Optimizing matrix transpose in cuda, 2009.

[20] A. Sharma and A. K. Varma. Trigonometric interpolation. *Duke Mathematical Journal*, 32:341–357, 1965.

[21] C. P. D. Silva, L. F. Cupertino, D. Chevitarese, M. A. C. Pacheco, and C. Bentes. Exploring data streaming to improve 3d fft implementation on multiple gpus. *Computer Architecture and High Performance Computing Workshops (SBAC-PADW), 2010 22nd International Symposium on*, 2010.

[22] R. Singleton. An algorithm for computing the mixed radix fast fourier transform. *Audio and Electroacoustics, IEEE Transactions on*, 17:93–103, 1969.

[23] P. Swarztrauber. Bluestein's fft for arbitrary n on the hypercube. *Parallel computing*, pages 6–7, 1991.

[24] V. Volkov. Better performance at lower occupancy. http://www.cs.berkeley.edu/ṽolkov/volkov10-GTC.pdf.

[25] S. Winograd. *Arithmetic Complexity Of Computations*. Society for Industrial & Applied Mathematics, 1987.

[26] J. Wu and J. Jaja. Optimized strategies for mapping three-dimensional ffts onto cuda gpus. *Innovative Parallel Computing*, pages 1–12, 2012.

[27] F. Xu and K. Mueller. Real-time 3d computed tomographic reconstruction using commodity graphics hardware. *Physics in Medicine and Biology*, 52:3405 3419, 2007.