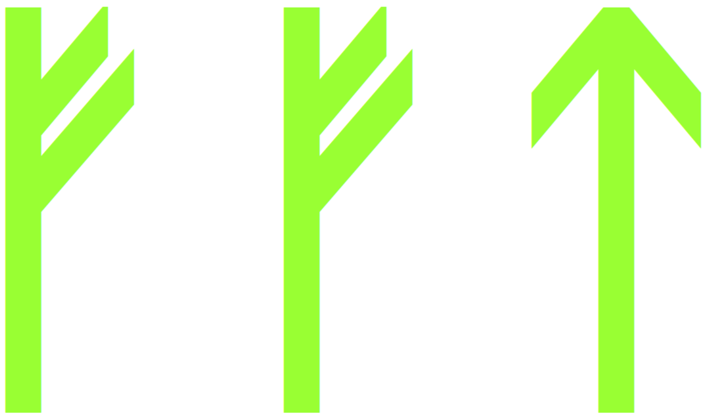
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

BSc Thesis in Computer Science

Mette Marie Kowalski <bpr314>

Designing and Accelerating a Generic FFT Library in Futhark



Supervisor: Cosmin Eugen Oancea

June 12, 2018

Abstract

The Fast Fourier Transform (FFT) is a computationally intensive operation used in a variety of fields, such as medicinal image processing. This thesis presents an implementation of an FFT library in the data-parallel programming language Futhark. The design of the library is generic with respect to different data-sets and radix, as well as being transparent to the user. This study also explores the extent to which FFT computations can be efficiently and generically expressed in a high-level, hardware-independent language. The results show that the radix has a significant effect on the performance, with a trend of higher radix giving higher performance. Other optimizations, algorithmic and compiler-wise, show a varying increase in performance as well, depending on the hardware. Though the presented implementation is still slower than the industry standard cuFFT, it holds potential and might show even better results as the Futhark compiler is optimized in the future.

Keywords: Fast Fourier Transform, Futhark, Graphics Processing Unit, Data-Parallel, Performance, Fusion

Contents

Chapter 1

Introduction

Computation on graphics hardware and the design of General-Purpose Graphics Processing Units (GPGPUs) is an area of ongoing research, as performance of graphics hardware is rapidly increasing [1]. High performance and low cost are the main factors that have caused interest in this are [2]. The generated need for programmability is what calls for high-level, data-parallel programming languages such as Futhark, developed by the APL Group at the Department of Computer Science at the University of Copenhagen (DIKU) [3].

The Discrete Fourier Transform (DFT) is a mathematical conversion of a sequence of function samples into a function of frequency. The reverse conversion is called Inverse DFT. Due to the relatively high asymptotic time complexity of $\mathcal{O}(N^2)$, several algorithms with complexity $\mathcal{O}(NlogN)$ [4] have been developed in order to increase efficiency. They are collectively known as Fast Fourier Transforms (FFTs). FFTs are used in a variety of fields, such as medicinal image processing, digital recording and quantum mechanics, therefore there is a need for performance-oriented implementations of FFTs.

Previous research has looked into the benefits of using GPGPUs for performing FFT computations, and for organizing generic FFTs libraries, which are easy to use. For example, Owens et al., 2007 [1] explains the motivations and techniques used in general purpose GPU computation. Both Govindaraju, Lloyd, Dotsenko, Smith & Manferdelli, 2008 [2] and Nukada, Ogata, Endo, & Matsuoka, 2008 [5] independently achieve a significant speed-up to any existing FFT implementation such as cuFFT, using NVIDIA CUDA. Matteo Frigo, 1999 [6] designed a specialized compiler that automatically discriminates the optimal way to compute the FFT for a specific data-set and hardware. Jørgensen & Hansen, 2018 [7] implement a simple FFT library in Futhark.

1.1 Thesis Objectives

The objective of this thesis is two-fold. The first objective is to investigate to what extent FFT computations can be efficiently and generically expressed in a high-level, hardware-independent language such as Futhark. Here the study examines both (1) algorithmic improvements, such as the effect of the radix on performance, and (2) general compiler optimization strategies, such as efficient exploitation of inner parallelism at the level of the CUDA block.

The second objective is to study the software engineering concerns that allow several implementations of FFTs to be combined into one library in a way that is transparent for the user. Specific instances are a generic representation of the radix that minimizes code clones, and techniques (based on the inspector-executor model) for analyzing the current data-set and discriminating the optimal implementation for the specific data-set.

1.2 Thesis Structure

Chapter 2 gives an overview of the background for the topics discussed in this thesis. It gives an introduction to the Futhark programming language, including the most common Second-Order Array Combinators (SOACs). After this follows a short section on GPU architecture and the idea of the GPGPU. Finally, the related works briefly mentioned in this introduction will be explained slightly more in-depth.

Chapter 3 explains the mathematical theory behind DFTs and FFTs needed in order to understand the implementation in **Chapter 4**. After giving the intuition for the DFT including time complexity, it shows how Cooley-Tukey - one of the most common FFT algorithms - is derived from the DFT formula. First, the pseudocode for the common recursive version of this algorithm is shown and the process is exemplified graphically with a computation tree. The time complexity for the algorithm follows from the tree and is proved by induction. Now a parallel version of Cooley-Tukey is derived in order to give the intuition for and show the similarity with the Stockham FFT algorithm. This points out the difference between different radix by showing the pseudocode for radix-2 and radix-4. The last part of this chapter explains the concept of factorization as used in hierarchical FFT computation, first by giving the mathematical derivation from the DFT to the formula and then by outlining the algorithm step-wise.

Chapter 4 takes the theory from the previous chapter and translates it into usable Futhark code. After outlining the way that the code is distributed over three different modules, the rationale behind this structure is explained. Here, a radix-2 specialized FFT implementation in Futhark is shown and the necessary changes to implement a generic representation of the radix are explained. Thereafter, the implementation of each module is discussed in more detail, always referring back to the theoretic background in **Chapter 3**. First, the main functionality of the FFT algorithm in the main FFT module is explained, after which the implementation of a single iteration is zoomed in on. Lastly, the planner-executor module combines several implementations into a module with user transparency. In the second part of the chapter, the difference between global and shared memory implementations is explained. Here, it is also shown how to pre-compute the twiddle factors for a global memory implementation, and the optimal way to implement factorization in Futhark is discussed.

Chapter 5 analyzes and discusses the performance of our implementation, while **Chapter 6** provides a conclusion.

Chapter 2

Background

For readers unfamiliar with some of the topics discussed in this thesis, this chapter gives a short overview of the Futhark programming language, general purpose GPU programming and related work on these topics.

2.1 Futhark

Futhark is a high-level, data-parallel, functional array language developed by the APL Group at the Department of Computer Science at the University of Copenhagen (DIKU) [3; 8; 9]. It is named after the Runic alphabet, which is why the front page of this thesis has the letters "FFT" written in Elder Futhark on it. The main Futhark compiler generates optimized code to run on the GPU and is currently implemented via OpenCL, but the language itself is hardware agnostic. Futhark also has a C compiler that generates C-code to run on the CPU, but this should primarily used for debugging/experimentation. The intended use for Futhark is to provide acceleration of the computationally heavy parts of an application, which are typically small. This is already possible despite the fact that the language is not fully realized yet, since the generated code from the compiler is easily integrable with other languages.

For example, an experiment has been reported [10], where a substantial subset of the interpreted and hence slow APL language [11] has been automatically translated to Futhark, accelerated on GPGPUs, and integrated within Python applications using a Futhark-to-Python code generator. This builds on earlier work on inter-operating computer algebra systems [12; 13].

Futhark has a few built-in functions called Second-Order Array Combinators (SOACs) that compile to parallel code. The difference between First-Order Array Combinators (FOACs) and SOACs is that FOACs always perform the same operation (i.e. concatenate two arrays), while the exact functionality of SOACs depend on the function they take as an input. The SOACs account for much of the data-parallelism in the code. The following gives a quick overview of their functionalities.

- `map` takes a function and any non-zero amount of arrays as its input, and applies the function to each element of the array, returning the resulting new array. It is a so-called an array transformer. Usually, the operation of applying the function is executed in parallel for all elements. Using the notation $[n]\alpha$ to denote an array of length `n` of elements of type α, and the notation $[\mathtt{a}_1, \ldots, \mathtt{a}_n]$ to denote an array literal, the type and semantics of `map` are formally defined as:

$$(\alpha \;\rightarrow\; \beta) \;\rightarrow\; [n]\alpha \;\rightarrow\; [n]\beta$$
$$\texttt{map f } [\mathtt{a}_1, \ldots, \mathtt{a}_n] = [\texttt{f } \mathtt{a}_1, \ldots, \texttt{f } \mathtt{a}_n]$$

- `reduce` is an array aggregator, taking a binary-associative operator, a neutral element and an array as its input and returning the result of combining the array elements via the operator. Its type and semantics are:

$$(\alpha \;\rightarrow\; \alpha \;\rightarrow\; \alpha) \;\rightarrow\; \alpha \;\rightarrow\; [n]\alpha \;\rightarrow\; \alpha$$
$$\texttt{reduce } \oplus \texttt{ e } [\mathtt{a}_1, \ldots, \mathtt{a}_n] = \texttt{e} \oplus \mathtt{a}_1 \ldots \oplus \mathtt{a}_n$$

  Compositions of `map` and `reduce` are aggressively fused into an operator named `redomap`, which has an efficient GPU implementation [14]. A `redomap` nested inside a `map` is also efficiently supported [15].

- `scan` is similar to `reduce` but instead of aggregating one single result, it returns one result for each prefix of the input array. Its type and semantics are:

$$(\alpha \;\rightarrow\; \alpha \;\rightarrow\; \alpha) \;\rightarrow\; \alpha \;\rightarrow\; [n]\alpha \;\rightarrow\; [n]\alpha$$
$$\texttt{scan } \oplus \texttt{ e } [\mathtt{a}_1, \ldots, \mathtt{a}_n] = [\mathtt{a}_1,\ \mathtt{a}_1 \oplus \mathtt{a}_2, \ldots,\ \mathtt{a}_1 \oplus \ldots \oplus \mathtt{a}_n]$$

- `scatter x inds vals` updates (in place) the elements of array `x` at the indices provided in `inds` with the values provided in `vals`. If an index falls outside the bounds of `x` then the corresponding update is ignored. Using the notation $*[n]\alpha$ to denote the type of an array which is "consumed" / created by an in-place update, `scatter`'s type is:

$$*[n]\alpha \;\rightarrow\; [m]\texttt{int} \;\rightarrow\; [m]\alpha \;\rightarrow\; *[n]\alpha$$

  Since `scatter` puts a high pressure on the memory system, the compiler attempts to fuse a `scatter` with the `map` operators that produce its input indices and/or values arrays whenever possible. The FFT implementation presented in this thesis relies heavily on `scatter` and on the ability of the compiler to fuse compositions like this.

2.2 Parallelism and GPGPUs

While CPU architectures exploit a limited amount of parallelism, GPGPUs are massively-parallel systems, supporting tens-to-hundred of thousands of hardware threads. They were originally intended for graphic processing and thus their architecture was optimized to efficiently process independent computations on pixels. Nowadays, they are used mainstream to accelerate general-purpose computations, for example option pricing [16] in finance domain, and may offer impressive speedups in comparison to CPU systems.

Futhark already optimizes code with the aforementioned SOACs, but in order to efficiently implement FFTs, the used algorithms must include a high amount of parallel operations and/or loops. As the next chapter will show, some algorithms are better than others. By analyzing the way they work, they can sometimes be re-written to increase parallelism.

One possible drawback in GPGPUs is the high latency of global memory, which is a few orders of magnitude slower than registers. However, GPGPU architectures address this somewhat by offering another type of memory known as *shared memory* (scratchpad memory), which has a latency comparable to that of registers. It can be viewed as a user-programmable cache. However, there is a limited supply (∽64k) of shared memory on each GPU multiprocessor. One can choose to either implement the FFT calculation using global or shared memory and both ways will be discussed in **Chapter 4**.

2.3 FFT Related Work

While this chapter has already reviewed various related work, for example on the design of the Futhark language, the remainder of this section discusses work related to FFT implementations aimed at execution on GPGPU.

Matteo Frigo, 1999 [6] explains the design and implementation of the FFT compiler `genfft`, which was developed for the open-source C FFT subroutine library Fast Fourier Transform in the West (FFTW). The compiler has a specialized case for real input and generates 95% of the performance-critical code for an arbitrary input length. Furthermore, `genfft` creates a plan transparent to the user, depending on factors such as input size and underlying hardware. The compiler uses a variety of well-known algorithms but has independently ”found” further simplifications and optimizations for these.

Govindaraju, Lloyd, Dotsenko, Smith & Manferdelli, 2008 [2] present some new FFT algorithms designed specifically to perform well on the GPU using the NVIDIA CUDA architecture. They execute smaller input size FFTs in shared memory and larger ones in global memory or using hierarchical FFT algorithms. Just like in Frigo, 1999 [6], the choice depends on the input size as well as the specific hardware. The implementation is a mixture of known and new FFT algorithms. The speed-up is 2-4 that of the cuFFT CUDA implementation.

Nukada, Ogata, Endo, & Matsuoka, 2008 [5] implement a 3-D FFT kernel in CUDA that is faster than any existing FFT implementation on the GPU at the time of publication. The kernel is designed to take full advantage of the GPU architecture, as well as identifying and facing many of the challenges that arise in general purpose GPU computation, specifically regarding FFTs. The presented 3-D FFT algorithm is optimized for the CUDA architecture by exploiting the way memory works on the GPU. They achieve a factor 3+ speed-up compared to cuFFT.

Owens et al., 2007 [1] explain the motivation and background for developing General-Purpose Graphics Processing Units (GPGPUs) and introduce the techniques used for this. They also survey the status of the research in the field at the time.

Jørgensen & Hansen, 2018 [7] implement a simple, radix-2 specialized FFT library for real and complex input in Futhark, using Cormen and Stockham algorithms.

Chapter 3

Theory

Fast Fourier Transforms (FFTs) are algorithms that implement the Fourier Transform (FT) in an efficient way. They are used in a variety of fields, such as medicinal image processing, digital recording and quantum mechanics; Therefore there is a need for performance-oriented implementations of FFTs.

The Discrete Fourier Transform (DFT) is the discrete form of the FT, while the inverse operation of summing the frequency components to get the time domain representation, is called the Inverse Discrete Fourier Transform (IDFT). This section will explain the mathematical background of the DFT and IFT, perform a short time complexity analysis, and discuss how one can speed up both of these transforms with some smart algorithms.

3.1 Discrete Fourier Transforms

First, let us take a look at the formal declaration of the DFT and IDFT:

$$X_k = \sum_{n=0}^{N-1} x_n \cos \frac{2\pi}{N}kn - i \sin \frac{2\pi}{N}kn, \qquad k = 0, 1, ..., N-1 \tag{3.1}$$

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cos \frac{2\pi}{N}kn + i \sin \frac{2\pi}{N}kn,, \qquad n = 0, 1, ..., N-1, \tag{3.2}$$

([17]). Here, (3.1) represents the DFT while (3.2) represents the IDFT. The notation $i = \sqrt{-1}$ is used. Note that the only difference between the two formulae is the multiplication by $\frac{1}{N}$ in (3.2), as well as the different signs of i.

Commonly, the declarations are referred to in a more compact form, recalling Euler's formula $e^{i\theta} = \cos\theta + i \sin\theta$. This gives following equations:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i\frac{2\pi}{N}kn}, \qquad k = 0, 1, ..., N-1 \tag{3.3}$$

$$x_n = \frac{1}{N} \sum_{n=0}^{N-1} X_k e^{i\frac{2\pi}{N}kn}, \qquad n = 0, 1, ..., N-1 \tag{3.4}$$

To reduce even further, one can define $W_N = e^{-i\frac{2\pi}{N}}$ and write:

$$X_k = \sum_{n=0}^{N-1} x_n W_N^{kn}, \qquad k = 0, 1, ..., N-1 \tag{3.5}$$

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k W_N^{-kn}, \qquad n = 0, 1, ..., N-1 \tag{3.6}$$

In the future, I will mainly refer to (3.5) and (3.6) when mentioning the DFT and IDFT, respectively.

3.1.1 Time Complexity

A short analysis of equation (3.5) shows clearly that the time complexity of the DFT is $\mathcal{O}(N^2)$. The formula sums N products of complex numbers, i.e. between the fuction/sequence value x_k and the factor W_N^{-kn}.

This complexity is suboptimal and several algorithms with complexity $\mathcal{O}(NlogN)$ [4] have been proposed to increase efficiency. They are collectively known as Fast Fourier Transforms (FFTs).

3.2 Fast Fourier Transforms

There are many variants of FFT algorithms, but most of them are based on Cooley-Tukey and Stockham. These are the two main algorithms that will be discussed here, as they have a similar approach to the problem, but differ in execution.

3.2.1 Cooley-Tukey

The first paper to propose an FFT was published in 1965 by James Cooley and John Tukey, resulting in the name Cooley-Tukey for the proposed algorithm.

The main idea behind this algorithm is to follow the divide-and-conquer algorithm design paradigm and divide the problem into several smaller ones that are easier to solve, and then to combine these solutions. The term radix is used to describe how many parts the DFT is split into. The most common form is the radix-2 FFT, in which (3.3) is divided into two parts:

$$\begin{aligned} X_k &= \sum_{n=0}^{N-1} x_n e^{-i\frac{2\pi}{N}kn} \\ &= \sum_{n=0}^{N/2-1} x_{2n} \cdot e^{-i\frac{2\pi}{N}2kn} + \sum_{n=0}^{N/2-1} x_{2n+1} \cdot e^{-i\frac{2\pi}{N}2k(n+1)} \end{aligned} \tag{3.7}$$

Here, the left side of the summation is the sum of the even indexes of the DFT, while the right side is the sum of the uneven indexes.

However, in order to decrease the running time it would be nice if the two sums were more similar. To achieve this, the right-hand exponential function can be broken in two:

$$\begin{aligned}
X_k &= \sum_{n=0}^{N/2-1} x_{2n} \cdot e^{-i\frac{2\pi}{N}2kn} + e^{-i\frac{2\pi}{N}k} \cdot \sum_{n=0}^{N/2-1} x_{2n+1} \cdot e^{-i\frac{2\pi}{N}2kn} && (3.8)\\
&= \sum_{n=0}^{N/2-1} x_{2n} \cdot e^{-i\frac{2\pi}{N/2}kn} + e^{-i\frac{2\pi}{N}k} \cdot \sum_{n=0}^{N/2-1} x_{2n+1} \cdot e^{-i\frac{2\pi}{N/2}kn} && (3.9)\\
&= \sum_{n=0}^{N/2-1} x_{2n} \cdot W_{N/2}^{kn} + W_N^k \cdot \sum_{n=0}^{N/2-1} x_{2n+1} \cdot W_{N/2}^{kn} && (3.10)
\end{aligned}$$

This shows that computing an FFT of length N can be reduced to recursively solving two FFTs of length N/2 - one corresponding to the even and odd indices of the original sequence, respectively. Furthermore, note that in the original equation 3.5: $X_k = \sum_{n=0}^{N-1} x_n W_N^{kn}$, the twiddle factor W_N^{kn} has been calculated N times, since the sum goes up to N. In 3.10, the twiddle factor (here $W_{N/2}^{kn}$) is the same in both of the summations up to N/2, hence it only needs to be calculated N/2 times, half as often as originally.

For further optimization, one might take a look at the FFT of $k + N/2$, expressed $X_{k+N/2}$. Substituting into 3.10, the result is:

$$X_{k+N/2} = \sum_{n=0}^{N/2-1} x_{2n} \cdot W_{N/2}^{n(k+N/2)} + W_N^{k+N/2} \cdot \sum_{n=0}^{N/2-1} x_{2n+1} \cdot W_{N/2}^{n(k+N/2)} \tag{3.11}$$

This equation is quite similar to 3.10, except that the twiddle factors are different. If the twiddles in 3.11 can somehow be re-written to look more like the ones in 3.10, one might be able to use fewer operations to calculate the two equations. The calculations below demonstrate how to do that:

$$\begin{aligned}
\mathbf{W_{N/2}^{n(k+N/2)}} &= e^{\frac{-i2\pi nk+N/2}{N/2}}\\
&= e^{\frac{-i2\pi nk}{N/2}} \cdot e^{\frac{-i2\pi nN/2}{N/2}}\\
&= e^{\frac{-i2\pi nk}{N/2}} \cdot e^{\frac{-i2\pi n2N}{2N}}\\
&= W_{N/2}^{kn} \cdot e^{-i2\pi n}\\
&= W_{N/2}^{kn} \cdot 1\\
&= \mathbf{W_{N/2}^{kn}}
\end{aligned}$$

$$\begin{aligned}
\mathbf{W_N^{k+N/2}} &= e^{\frac{-i2\pi k+N/2}{N}}\\
&= e^{\frac{-i2\pi k}{N}} \cdot e^{\frac{-i2\pi N}{2N}}\\
&= W_N^k \cdot e^{-i\pi}\\
&= W_N^k \cdot -1\\
&= -\mathbf{W_N^k}
\end{aligned}$$

The result of substituting these re-written twiddle factors back into 3.11 is:

$$X_{k+N/2} = \sum_{n=0}^{N/2-1} x_{2n} \cdot W_{N/2}^{kn} - W_N^k \cdot \sum_{n=0}^{N/2-1} x_{2n+1} \cdot W_{N/2}^{kn} \tag{3.12}$$

Recalling 3.10:

$$X_k = \sum_{n=0}^{N/2-1} x_{2n} \cdot W_{N/2}^{kn} + W_N^k \cdot \sum_{n=0}^{N/2-1} x_{2n+1} \cdot W_{N/2}^{kn},$$

you can see that the only difference between the two equations is the twiddle factor W_N^k, which changes sign, while the other twiddle factor $W_{N/2}^{kn}$ remains the same. It's still not evident how this minimizes the running time from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$. A better understanding is achieved when describing the algorithm in pseudocode. The most common version of Cooley-Tukey is the recursive one, but one of the next sections will also show how to derive a non-recursive, parallel version and compare the two.

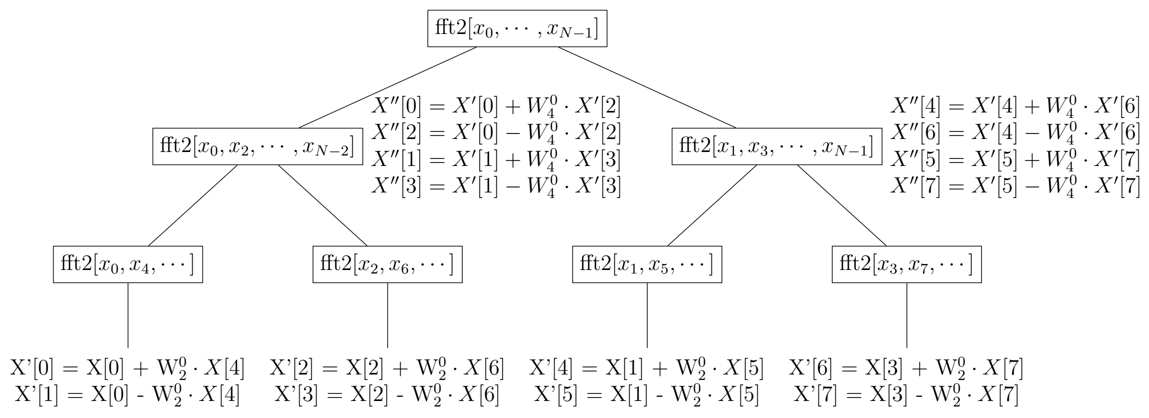
Pseudocode for the recursive Cooley-Tukey algorithm (Radix-2)

```
 1: function FFT-CT-REC(X,R,N,S)
 2:     if N=1 then X[0]
 3:     else
 4:         X[0,..,N/R-1] = FFT-CT-REC(X,R,N/R,2S)
 5:         X[N/R,...,N-1] = FFT-CT-REC(X+S,R,N/R,2S)
 6:         for k=0; N/R-1; k++ do
 7:             t ← $W_N^k$· X[k+N/R]
 8:             xk ← X[k]
 9:             X[k] ← xk+t
10:             X[k+N/R] ← xk - t
11:         end for
12:     end if
13: end function
```

This function should be called with R=2 and S=1. As described in the previous section, the algorithm divides the input into an odd and an even part and then recursively processes those parts. This is represented in the pseudocode in lines 4-5 with a recursive call to the same function. The stride (S) is multiplied by two in each call, as the indexing changes in each iteration; first one takes every second number (as this is radix-2), then every fourth and so on. Furthermore, the size of the input is divided by two (the radix) in each iteration. So all this corresponds to the two sums in 3.10.

The loop in line 6 combines the smaller DFTs into bigger ones, using the fact that 3.10 and 3.12 are so similar. The right-hand sum of 3.10 is calculated first and is then added to/ subtracted from the left-hand side in order to calculate X_k and $X_{k+N/2}$, respectively.

Computation tree for recursive Cooley-Tukey algorithm (Radix-2)


fft2[$x_0, \cdots, x_{N-1}$]
fft2[$x_0, x_2, \cdots, x_{N-2}$]
$X''[0] = X'[0] + W_4^0 \cdot X'[2]$
$X''[2] = X'[0] - W_4^0 \cdot X'[2]$
$X''[1] = X'[1] + W_4^0 \cdot X'[3]$
$X''[3] = X'[1] - W_4^0 \cdot X'[3]$
fft2[$x_1, x_3, \cdots, x_{N-1}$]
$X''[4] = X'[4] + W_4^0 \cdot X'[6]$
$X''[6] = X'[4] - W_4^0 \cdot X'[6]$
$X''[5] = X'[5] + W_4^0 \cdot X'[7]$
$X''[7] = X'[5] - W_4^0 \cdot X'[7]$
fft2[$x_0, x_4, \cdots$]
fft2[$x_2, x_6, \cdots$]
fft2[$x_1, x_5, \cdots$]
fft2[$x_3, x_7, \cdots$]
X'[0] = X[0] + $W_2^0 \cdot X[4]$
X'[1] = X[0] - $W_2^0 \cdot X[4]$
X'[2] = X[2] + $W_2^0 \cdot X[6]$
X'[3] = X[2] - $W_2^0 \cdot X[6]$
X'[4] = X[1] + $W_2^0 \cdot X[5]$
X'[5] = X[1] - $W_2^0 \cdot X[5]$
X'[6] = X[3] + $W_2^0 \cdot X[7]$
X'[7] = X[3] - $W_2^0 \cdot X[7]$

Time complexity and parallelism of Cooley-Tukey

Both the pseudocode and computation tree demonstrate clearly how the algorithm divides the input sequence into two new sequences of size N/2. It then recursively divides those into half, resulting in four sequences of size (N/2)/2 and so on. Combining the results of the sub-problems has the complexity $\mathcal{O}(N)$, as this simply assembles a list of size N. Assuming that N is a power of two, so the input is divided completely equal, the following recurrence is derived:

$$T(N) = 2T(N/2) + \mathcal{O}(N),$$

with the base-case $T(1) = \mathcal{O}(1)$ because a sequence of length 1 is processed at line 2 where the element is returned unmodified. $T(N/2)$ corresponds to the recursive call in lines 4-5, while the addition of $\mathcal{O}(N)$ corresponds to the loop at line 6.

Using this recurrence, one can use induction to prove the time complexity of $\mathcal{O}(N \lg N)$.

- Express N as $N = 2^k$ and the complexity of N as C_{2^k}.
- For the base case, $N = 2^0 = 1$ and $C_{2^0} = 1$.
- We want to show that $C_{2^k} = 2^k \lg 2^k = k2^k$.
- Now we assume that $C_{2^{k-1}} = (k-1)2^{k-1}$.

Using the recurrence, it is apparent that:

$$\begin{aligned} C_{2^k} &= 2C_{2^{k-1}} + 2^k \\ &= 2(k-1)2^{k-1} + 2^k \\ &= (k-1)2^k + 2^k \\ &= k2^k \end{aligned}$$

This means the time complexity of $\mathcal{O}(N \lg N)$ can be proved by induction.

As for parallelism, the loop in the algorithm is actually parallel in the sense that it can be computed for different values of k at the same time. This is because none of the operations in the loop use any values that are computed in other iterations. It is evident when looking at the computation tree, as each of the leaves can be computed by themselves and combing two nodes is also independent from combining two other nodes at the same level. However, there are certainly dependencies between the different levels of the tree. The next section will take a look at the parallel Cooley-Tukey algorithm that can be derived from the computation tree.

Pseudocode for parallel Cooley-Tukey algorithm (Radix-2)

```
 1: function FFT-CT-PARALLEL(X,N,R,bits)
 2:     R = 2
 3:     Y = malloc space
 4:     Z = malloc space
 5:     twiddles ← map (x -> twiddle(x)) [$1 \cdots n-1$]
 6:     offset ← 0
 7:     for i=0; i<bits; i++ do
 8:         for j=0; j<N/R-1; j++ do
 9:             Ns ← $2^i$
10:             t ← twiddles[offset+(j%Ns)]
11:             idx ← ((j/Ns)*Ns*R) + (j%Ns)
12:             Y[idx] ← X[idx] + t
13:             Y[idx + Ns] ← X[idx + N/R] - t
14:             Z ← X
15:             X ← Y
16:             Y ← Z
17:             offset ← offset + $2^i$
18:         end for
19:     end for
20: end function
```

The parallel Cooley-Tukey algorithm is quite similar to the recursive one, but instead of depth-first, it uses a breadth-first search to calculate the nodes of the computation tree. The twiddle factors are precomputed as there is a fixed amount of them. Double buffering is used to ensure the parallelism of the inner loop; first, the computed values for the output array are re-written into an entirely new array Y (lines 12-13). Then, the current output array is saved into Z (line 13) and afterwards the new value is written into the output array (line 15). Finally, Y is changed to contain the values of the output array. This results in a lack of data dependency in the inner loop, which may now be executed in parallel.

3.2.2 Stockham

One of the many other FFT algorithms based on Cooley-Tukey is the Stockham autosort FFT [18]. Let us take a look at the Stockham algorithm for radix-2 and 4. So far this study has only looked at radix-2, but there is an endless amount of different radix for FFT algorithms. The radix-4 algorithm divides the input into four pieces instead of two, which means one needs to compute four indexes and four values in each iteration.

Pseudocode for the Stockham radix-2 algorithm

```
function FFT-STOCKHAM(X,N,R)
    R = 2
    Z = malloc space
    Y = malloc space
    for Ns = 1; Ns < N; Ns *= R do
        for j=0; j<N/R ; j++ do
            t ← $W_{Ns \cdot R}^{j\%Ns}$· X[j+N/R]
            Y[(j/Ns)*Ns*R + (j%Ns)] ← X[j] + t
            Y[(j/Ns)*Ns*R + (j%Ns) + Ns] ← X[j] - t
            Z ← X
            X ← Y
            Y ← Z
        end for
    end for
end function
```

Pseudocode for the Stockham radix-4 algorithm

```
function FFT-STOCKHAM(X,N,R)
    R = 4
    Z = malloc space
    Y = malloc space
    for Ns = 1; Ns < N; Ns *= R do
        for j=0; j<N/R ; j++ do
            $t_1$ ← $W_{Ns \cdot R}^{j\%Ns}$· X[j+N/R]
            $t_3$ ← $W_{Ns \cdot R}^{j\%Ns}$· X[j+3N/R]
            Y[(j/Ns)*Ns*R + (j%Ns)] ← X[j] + X[j+2*N/R] + $t_1 + t_3$
            Y[(j/Ns)*Ns*R + (j%Ns) + Ns] ← X[j] - X[j+2*n/r] + $t_1 - t_3$
            Y[(j/Ns)*Ns*R + (j%Ns)+2*Ns ← X[j] + X[j+2*N/R] - $t_1 - t_3$
            Y[(j/Ns)*Ns*R + (j%Ns)+3*Ns ← X[j] - X[j+2*n/r] - $t_1 + t_3$
            Z ← X
            X ← Y
            Y ← Z
        end for
    end for
end function
```

Similar to the parallel Cooley-Tukey algorithm, the algorithm uses a breadth-first approach to calculate all the values, which promotes parallel execution as discussed previously. However, it uses a different sorting network than Cooley-Tukey. But since the parallel Cooley-Tukey algorithm has already been derived, this one should be easy to understand. Note that the twiddle factors could be calculated beforehand, but as they commonly are not, that optimization hasn't been included in the pseudocode. Most of the inner loop is parallel due to the double-buffering; each of the four calculations reads from one array and writes to another, similar to the parallel Cooley-Tukey algorithm. Furthermore, each iteration is independent from the others.

3.2.3 Factorization and Hierarchical FFT

Factorization of an input sequence has the potential to optimize the calculation of an FFT further. If the length N of an sequence can be factorized so that N $= N_1 \cdot N_2$, one can also re-write $n = n_1N_2 + n_2$ and $k = k_1 + k_2N_1$ and then substitute into the DFT formula 3.5 ($X_k = \sum_{n=0}^{N-1} x_n W_N^{kn}$):

$$\begin{aligned}
X_{k_1+k_2N_1} &= \sum_{n_1n_2=0}^{N_1N_2-1} x_{n_1N_2+n_2} W_{N_1N_2}^{(k_1+k_2N_1)(n_1N_2+n_2)} \\
&= \sum_{n_2=0}^{N_2-1}\sum_{n_1=0}^{N_1-1} x_{n_1N_2+n_2} \cdot e^{\frac{-2\pi i}{N_1N_2}(k_1+k_2N_1)(n_1N_2+n_2)} \\
&= \sum_{n_2=0}^{N_2-1}\sum_{n_1=0}^{N_1-1} x_{n_1N_2+n_2} \cdot e^{\frac{-2\pi i}{N_1N_2}(k_1n_1N_2+k_1n_2+k_2N_1n_1N_2+k_2N_1n_2)} \\
&= \sum_{n_2=0}^{N_2-1}\sum_{n_1=0}^{N_1-1} x_{n_1N_2+n_2} \cdot e^{\frac{-2\pi i}{N_1}k_1n_1} \cdot e^{\frac{-2\pi i}{N}k_1n_2} \cdot e^{-2\pi i k_2 n_1} \cdot e^{\frac{-2\pi i}{N_2}k_2N_1n_2} \\
&= \sum_{n_2=0}^{N_2-1}\left[\left(\sum_{n_1=0}^{N_1-1} x_{n_1N_2+n_2} W_{N_1}^{k_1n_1}\right) W_N^{k_1n_2}\right] W_{N_2}^{k_2n_2}, \qquad (3.13)
\end{aligned}$$

[6]. Note that the factor $e^{-2\pi i k_2 n_1}$ vanishes, as $e^{-2\pi i x} = 1$ for all integers x.

Imagine the input sequence as a 2-D matrix of size N_1 x N_2. The FFT can be computed according to 3.13 by following these steps, known as hierarchical FFT computation:

1. Perform N_2 FFTs column-wise. This corresponds to the inner sum in 3.13. The column number N_2 is fixed from the outside, while the row number N_1 is invariant.[i]

2. Multiply with the twiddle factors $W_N^{k_1n_2}$.

3. Perform N_1 FFTs row-wise; one FFT per N_1, which are the number of rows. This corresponds to the outer sum in 3.13. These are performed on the result from step 2.

4. Transpose the result from step 3.[ii]

The best way to divide N depends on a variety of factors, such as the length of the input sequence and the block-size of the system, but this issue will be addressed when discussing how to implement factorization.

[i]An easy way to compute this is in parallel is to transpose the array, then perform the FFTs row-wise (on the previous columns), and then transpose back the result.

[ii]The original matrix has the dimensions N_1 x N_2, but the indexing of the result $X_{k_1+k_2N_1}$ is that of the transposed matrix, which has the dimensions N_2 x N_1.

Chapter 4

Modularity

The main objective of this thesis is to study a performance-oriented generic FFT implementation in Futhark, as well as the software engineering concerns that would allow several implementations of FFTs to be combined into one library in a way that is transparent for the user.

The suggested implementation contains a generic representation of the radix that minimizes code clones from the current Futhark FFT library, and analyzes the current data-set to discriminate the optimal implementation for the specific data-set. It also allows the user to choose between using a global or shared memory FFT implementation, and uses algorithmic optimization such as pre-computation of the twiddle factors.

This chapter describes the implementation and organization of the FFT library. The extended source code can be found in the appendix.

4.1 Modules

In order to maintain a high level of flexibility and reduce code complexity, I split the implementation of the library into three Futhark modules with the following main functionalities:

1. **The FFT iteration.** A module that implements the calculation of an FFT for a sequence that has the length of the radix (2, 4, 8 etc.).
2. **The main functions of the FFT algorithm.** A parametric module that receives an iteration module and implements the skeleton of the FFT algorithm; a loop with $\lg N$ iterations which updates and shuffles the elements in each iteration.
3. **The creation and execution of a plan.** A module that creates a plan by taking the input size N and pre-computes all the information that depends on N only. Then, that plan can be executed as many times as needed. The user receives no information on how the plan was created, i.e. what radix is used etc.

I will describe the implementation of these on a general level in the following sections, and zoom in on more specific parts of the implementation in relation to global vs. shared memory FFTs afterwards.

4.1.1 Rationale of the Structure

This section takes a look at the current radix-2 specialized Futhark implementation of an FFT library, and explains what changes were necessary in order to implement a generic representation of the radix.

```
1   let fft_iteration [n] (forward: f32) (ns: i32) (data: [n]complex) (j: i32) :
2                                               (i32, complex, i32, complex) =
3     let angle = (-2f32 * forward * f32.pi * r32 (j % ns)) / r32 (ns * radix)
4     let (v0, v1) = (data[j], data[j+n/radix] complex.*
5                        (complex.mk (f32.cos angle) (f32.sin angle)))
6     let (v0, v1) =  (v0 complex.+ v1, v0 complex.- v1)
7     let idxD = ((j/ns)*ns*radix) + (j % ns)
8     in (idxD, v0, idxD+ns, v1)
```

Listing 4.1: FFT iteration function for radix-2

This function implements the functionality of a single FFT iteration, more precisely one iteration of the inner for-loop in the Stockham radix-2 algorithm (section 3.2.2). The twiddle factor is computed in lines 4-5, where the angle or $W_{Ns\cdot R}^{j\%Ns}$ is multiplied with X[j+N/R]. The return values are calculated by adding the twiddle factor to X[j] and subtracting it from X[j+N/R] in line 6. The indices calculated in lines 7-8 are the exact same ones as in the Stockham algorithm.

The `fft_iteration` function is called by the `fft'` function, which corresponds to `fft-stockham` as presented in (section 3.2.2).

```
1   let fft' [n] (forward: f32) (input: [n]complex) (bits: i32) : [n]complex =
2     let input  = intrinsics.cosmin_flatten (copy (intrinsics.unflatten
3                                            (n/radix, radix, input)))
4     let output = intrinsics.cosmin_flatten (copy (intrinsics.unflatten
5                                            (n/radix, radix, input)))
6     let ix = iota(n/radix)
7     let NS = map (radix**) (iota bits)
8     let (res,_) =
9       loop (input': *[n]complex, output': *[n]complex) = (input, output)
10      for ns in NS do
11        let (i0s, i1s, v0s, v1s) =
12          unsafe (unzip
13            (map (fft_iteration forward ns input') ix))
14        in (scatter output' (i0s ++ i1s) (v0s ++ v1s), input')
15    in res
```

Listing 4.2: FFT main function for radix-2

Denoting `radix` by `r`, the array `NS` created at line 7, holds values [$\mathtt{r}^0$, $\mathtt{r}^1$,..., `n/r`] (because `bits` is $\log_r$ `n` and `iota q`, creates the array `[0,...,q-1]` for some `q`). The loop between lines 9 and 14, executes sequentially `bits` iterations, in which the values of `ns` are (consecutively) taken from the elements of array `NS`. The variables whose values are variant to the loop, i.e., `input'` and `output'`, are initialized to `input` and `output` and are bound in order to the result of the loop body (line 14) for the next iteration.

Note that the implementation of the loop uses double buffering, since the `scatter`-updated value of `output'` is bound to loop-variant array `input'` for the next iteration. Similarly, the `map` at line 13 implements the innermost loop in the FFT pseudocode, and operates on array `ix`, which is constant and is initialized at line 6 with `[0,...,n/r-1]`. The result of the `map` is `unzipped`, resulting in 2 arrays of indexes and 2 arrays of values (r = radix). Finally, the two index and the two values arrays are concatenated into two arrays, based on which, the array combinator `scatter` distributes the results (line 14).

If one was to implement these functions for radix-4, `fft_iteration` would return a tuple of four indices and four values. It follows that the return type would be different from radix-2. This goes on to have an effect on line 11 in `fft'`, which would also have four arrays of indexes and four arrays of values, so that `scatter` would have to operate at line 14 on a concatenation of four index and four value arrays, respectively. As all of these differences are type-based, an evident solution is to create abstracted types that depend on the radix.

First, a return type for the `fft_iteration` function is needed: a tuple of r indices and r values (r=radix). Second, that type needs to be "multiplied", i.e. a tuple of r arrays of indices and r arrays of values, which is the result of mapping the iteration onto the iterator *ix*. This will call for specialized `unzip` and `concatenate` to fit the abstract types. The following two sections will show how to implement this into modules.

4.1.2 The FFT Iteration Module

This module implements the functionality of a single FFT iteration with abstract types to fit any radix. One module of this type must created for each radix.

```
module type fft_iteration = {
  type tuple_inds_vals
  type iter_ret

  val radix: i32
  -- | Concatenate the specified tuples of indexes or values;
  -- | result type is not specialized, as it needs to fit into scatter
  val concat_inds: tuple_inds_vals -> []i32
  val concat_vals: tuple_inds_vals -> []complex

  -- | A regular unzip returns an array of tuples, but this one
  -- returns the specific type we created
  val unzip_it: [](iter_ret) -> tuple_inds_vals
  -- | One iteration of the fft loop
  val fft_iter: f32 -> i32 -> []complex -> i32 -> iter_ret
  -- | One iteration of the fft loop with pre-computed twiddles
  val fft_iter_precomp: i32 -> i32 -> []complex -> []complex
                        -> i32 -> iter_ret
}
```

Listing 4.3: FFT iteration module type

The types of indexes and values will be the same for all kinds of FFT iterations, but the module has two abstract types that depend on the radix:

- `tuple_inds_vals`: A tuple of r arrays of indexes and r arrays of values to write to these, where r is the radix of the FFT iteration. This is used in the main FFT module as a return type of mapping the iteration onto the input.
- `iter_ret`: A tuple of r indexes and r values, where r is the radix. This is the return type for one iteration.

For the main function of the module, there are two variations: one with pre-computed twiddle factors, and one without. There are some code clones between the two variations, which is something that could be improved in the future. I will discuss pre-computation of the twiddle factors in the section regarding global memory FFTs and only explain the regular version here.

```
1  module fft_iteration2: (fft_iteration) = {
2    type ind = i32
3    type vl = complex
4    type inds = []ind
5    type vals = []vl
6    type tuple_inds_vals = (inds, inds, vals, vals)
7    type iter_ret = (ind, ind, vl, vl)
8
9    let radix = 2i32
10   let concat_inds ((a,b,_,_): tuple_inds_vals) = concat a b
11   let concat_vals ((_,_,c,d): tuple_inds_vals) = concat c d
12
13   let unzip_it (x: [](iter_ret)) = unzip x
14   let fft_iter [n] (forward: f32) (ns: i32) (data: [n]complex) (j: i32) : (iter_ret) =
15     let angle = (-2f32 * forward * f32.pi * r32 (j % ns)) / r32 (ns * radix)
16     let (v0, v1) = (data[j], data[j+n/radix] complex.*
17                     (complex.mk (f32.cos angle) (f32.sin angle)))
18     let (v0, v1) =  (v0 complex.+ v1, v0 complex.- v1)
19     let idxD = ((j/ns)*ns*radix) + (j % ns)
20     in (idxD, idxD+ns, v0, v1)
```

Listing 4.4: FFT iteration module for radix-2

This implementation only differs from the radix-2 specialized one (see subsection 4.1.1) in using the abstract return type `iter_ret` (line 7) instead of a regular tuple of indexes and values.

```
module fft_iteration4: (fft_iteration) = {
  type ind = i32
  type vl = complex
  type inds = []ind
  type vals = []vl
  type tuple_inds_vals = (inds, inds, inds, inds,
                          vals, vals, vals, vals)
  type iter_ret = (ind, ind, ind, ind, vl, vl, vl, vl)

  let radix = 4i32
  let concat_inds ((a,b,c,d,_,_,_,_): tuple_inds_vals) = a ++ b ++ c ++ d
  let concat_vals ((_,_,_,_,a,b,c,d): tuple_inds_vals) = a ++ b ++ c ++ d

  let unzip_it (x: [](iter_ret)) = unzip x
  let twiddle (a: complex) : complex =
    complex.mk (complex.im a) (- (complex.re a))

  let fft_iter [n] (forward: f32) (ns: i32) (data: [n]complex) (j: i32)
                                                        : (iter_ret) =
    let angle = (-2f32 * forward * f32.pi * r32 (j%ns)) / r32 (ns * radix)
    let tw = complex.mk (f32.cos angle) (f32.sin angle)
    let a0 = data[j]
    let a1 = tw complex.* (data[j+n/radix])
    let a2 = data[j+2*n/radix]
    let a3 = tw complex.* (data[j+3*n/radix])

    let tw = tw complex.* tw
    let a2 = tw complex.* a2
    let a3 = tw complex.* a3

    let b0 = a0 complex.+ a2
    let b1 = a0 complex.- a2
    let b2 = a1 complex.+ a3
    let b3 = twiddle (a1 complex.- a3)
```

```
35
36      let v0 = b0 complex.+ b2
37      let v2 = b0 complex.- b2
38      let v1 = b1 complex.+ b3
39      let v3 = b1 complex.- b3
40
41      let idxD = ((j/ns)*ns*radix) + (j % ns)
42      in (idxD, idxD+ns, idxD+2*ns, idxD+3*ns, v0, v1, v2, v3)
```

Listing 4.5: FFT iteration module for radix-4

The radix-4 module implementation follows the algorithm described in section 3.2.2. Although it returns a different tuple than the radix-2 module, it fits the `iter_ret` type specified in this module (line 8). All other functions and types are also specialized for radix-4. Lines 21-38 are slightly more verbose than the pseudocode in order to make the computation of radix-4 FFT values more clear.

Trivial functionality of the iteration module is an implementation of the radix (line 2), functions to concatenate the indexes and values with each other (lines 11-12), as well as an unzip function for the specified types (line 14).

4.1.3 The Main FFT Module

This module implements the main functionality of the FFT for complex and real input.

```
module type fft_module = {
  val radix: i32
  -- | Find out whether input size is a power of the radix
  val powOfR: i32 -> bool
  -- | FFT of complex numbers
  val fft [n]: [n](f32, f32) -> bool -> [n](f32, f32)
  -- | FFT of 2-D array of complex numbers for factorization
  val fact: [][](f32,f32) -> i32 -> [](f32, f32)
  -- | FFT of real numbers
  val fft_real [n]: [n]f32 -> bool -> [n]f32
}
```

Listing 4.6: FFT main module type

Modules with this type should be parametric, meaning they take an FFT iteration module as input and produce an FFT module. This way, they use the same radix as the specified iteration module.

```
module fft_module(Iter: fft_iteration): fft_module = {
  let radix = Iter.radix
  ...
}
```

Listing 4.7: Parametric FFT module

The function of interest in this module is `fft`, which computes the FFTs for complex input. For real numbers, the input is converted into complex numbers, transformed with `fft` and then converted back to real format, as the following code exemplifies:

```
  let fft_real [n] (data: [n]f32) (precomp: bool): ([n]f32) =
    map (\r -> complex.re r)
        (fft (map (\r -> complex.mk_re r) data) (precomp))
```

Listing 4.8: FFT for real input

The boolean parameter *precomp* specifies whether a pre-computation of the twiddle factors should be performed. If the input is real, that information is sent on to `fft`, otherwise it is specified there.

```
let generic_fft [n] (forward: bool) (data: [n](f32,f32)) (precomp: bool):
                                                        [n](f32,f32) =
  let n_bits = logR n
  let forward' = if forward then 1f32 else -1f32
  in if (precomp == false) then take n (fft' forward' data n_bits)
  else take n (fft_precomp' forward' data n_bits)

let fft [n] (data: [n](f32, f32)) (precomp: bool): [n](f32, f32) =
  generic_fft true data precomp
```

Listing 4.9: FFT wrapper function

The `fft` function is actually a wrapper function that calls a more generic function, which is a wrapper for forward and inverse FFTs. For inverse FFTs, `generic_fft` is called with the parameter *forward* set to false. The generic function now calls `fft'`, which is where the actual implementation of the FFT algorithm is hidden. Depending on whether pre-computation of the twiddle factors is demanded, `fft_precomp'` can be chosen as well, but this will be discussed in the global memory FFT chapter.

```
 1  let fft' [n] (forward: f32) (input: [n]complex) (bits: i32) : [n]complex =
 2    let input  = intrinsics.cosmin_flatten (copy (intrinsics.unflatten
 3                                              (n/radix, radix, input)))
 4    let output = intrinsics.cosmin_flatten (copy (intrinsics.unflatten
 5                                              (n/radix, radix, input)))
 6    let ix = iota(n/radix)
 7    let NS = map (radix**) (iota bits)
 8    let (res,_) =
 9      loop (input': *[n]complex, output': *[n]complex) = (input, output)
10      for ns in NS do
11        let (inds_vals: Iter.tuple_inds_vals) =
12          unsafe (Iter.unzip_it
13            (map (Iter.fft_iter forward ns input') ix))
14        in (scatter output' (Iter.concat_inds inds_vals)
15                            (Iter.concat_vals inds_vals), input')
16    in res
```

Listing 4.10: FFT main function for generic radix

This implementation is quite similar to the radix-2 specialized one (see section 4.1.1), but it is using a version of `fft_iter` that returns the abstract type `iter_ret` - a tuple of r indices and r values, r being the radix. This type is "hidden" in the above code though, because the above function uses the iteration function in a `map` over $[1..N/R]$ (lines 6 and 13). This returns an array of length (N/radix) of iteration results instead of a single result of type `iter_ret`.

As the indices and values need to be separated, essentially the SOAC `unzip` (line 12) is used to change the array of iteration results into a tuple of r arrays of indices and r arrays of values. This is the second abstract type from the iteration module, `tuple_inds_vals`.

The use of the abstract type calls for a specialized `unzip` function that returns this type, which has been implemented in the iteration module. Futhermore, in order to concatenate the indices and values with each other respectively in a generic way, two specialized `concatenate` functions are implemented, which will concatenate either the r index arrays (line 14) or the r value arrays (line 15) with each other. In the function, the `map` in line 13 and `scatter` in line 14 are fused together by the compiler in order to increase parallelization.

A trivial functionality of the fft module is the function `powOfR` to determine whether the input size is a power of the radix, which can be useful in deciding how to split the input for shared memory FFTs. The module also contains a function `fact` to compute FFTs of 2-dimensional input for factorization. This will be discussed in detail in an upcoming section on shared memory FFTs.

4.1.4 The Planner-Executor Module

This module implements a planner to determine the best FFT implementation for a given input, as well as an executor that executes the plan.

```
module type fft_planex = {
  type n_t
  -- | A record of information about the plan for the input
  type plan_t

  -- | Create a plan how to calculate the fft
  val planner: i32 -> i32 -> plan_t
  -- | Execute the plan; calculate the fft
  val executor [n]: [n]n_t -> plan_t -> [n]n_t
  val planex [n]: [n]n_t -> i32 -> [n]n_t
  val planex_batch [n][m]: [n][m]n_t -> i32 -> [n][m]n_t
}
```

Listing 4.11: FFT planner/executor module

The type `plan_t` is a record with information on how to calculate the FFT for the given input. The only parameter needed to develop a plan is the size of the input and for shared memory the block size as well. For global memory, the plan only includes the radix, but for shared memory, there is also the block size of the GPU environment and the number of splits. The choice for these values is made by the `planner` function, which will be explained more thoroughly in the two upcoming sections.

After the best plan has been chosen, it must be executed by passing the input as well as the plan record on to the `executor` function. The plan is then executed with the given input, either by using the main function from the FFT main module, or by splitting the data as determined by the plan and then using the FFT function on the different segments. This will be discussed in more detail in the two following sections. Finally, the function `planex` is a combining wrapper for the `planner` and `executor` and `planex_batch` does the same for a batch of FFTs.

4.2 Global and Shared Memory FFTs

One objective of this thesis is an efficient exploitation of the GPU architecture in order to increase the performance of FFTs in Futhark. Here it makes sense to look at memory latency. The latency of global memory on the GPU is a few orders of magnitude higher than register latency, but there is another type of memory known as *shared memory*, which has a latency comparable to that of register latency. It can be viewed as a user-programmable cache. However, there is a limited supply (∽64k) of shared memory on each GPU multiprocessor.

This means one can choose to either implement the FFT calculation using global or shared memory. The global memory implementation is a naive implementation that mainly follows the code examples from the previous sections. However, there are a few possible optimizations, such as pre-computation of the twiddle factor that will be discussed. The other way is called hierarchical FFT and is aimed at implementing the sequential loop of $\lg N$ iterations in shared memory with the goal of increasing performance. An example and explanation of hierarchical FFT implementation follows after the global memory section.

4.2.1 Implementation of Global Memory FFTs

This section will follow the FFT implementation that uses only global memory through the different modules. Pre-computation of the twiddle factors will be explained after a brief description of what happens in the planner-executor module.

```
 1  module fft_globalmem  : (fft_planex) = {
 2    type n_t = f32
 3    type plan_t = {rad: i32}
 4
 5    let planner (n) (block_sz: i32) : plan_t =
 6    -- Preferrably use radix 8
 7      let plan_rad = if      (fft_mod8.powOfR n == true) then 8
 8                     else if (fft_mod4.powOfR n == true) then 4 else 2
 9      in {rad = plan_rad}
10
11    let executor [n] (data: [n]n_t) (plan: plan_t) : [n]n_t =
12      if      (plan.rad == 8) then fft_mod8.fft_real data true
13      else if (plan.rad == 4) then fft_mod4.fft_real data true
14      else                         fft_mod2.fft_real data true
15
16    let planex [n] (data:[n]n_t) (block_sz: i32) =
17      let plan = planner n block_sz
18      in executor data plan
19
20    let planex_batch [n][m] (data: [n][m]n_t) (block_sz: i32) : [n][m]n_t =
21      let plan = planner n block_sz
22      in map (\f -> executor f plan) data
23  }
```

Listing 4.12: Global FFT planner/executor module

Since the input size for the FFT functions must be a power of the radix, the module uses `powOfR` to determine whether the input can be used for a certain FFT implementation. The algorithm chooses the highest possible radix (lines 6-9), since this typically translates to higher performance, assuming that the sequence is large enough to fully utilize the GPU parallelism. In order to use pre-computation of the radix, the main FFT module function is called with the *precomp* parameter set to true (lines 12-14).

In a naive FFT implementation, each of the $N \lg N$ iterations computes exactly N/r twiddle factors (r=radix), which leads to some redundant computation. To observe this, recall that in each FFT iteration (see 4.1.2), the angle W_N^k for the factor is computed using the following formula:

$$W_N^k = (2\pi(j\%ns))/(ns \cdot r),$$

where j indicates which index from $[0 \cdots N/r - 1]$ is being computed and ns is the value of the iterator $NS = [r^0, r^1, \cdots, r^{k-1}]$. For a certain value of ns, the number of distinct twiddle factors is in fact ns, due to the modulo operation. For example, for $ns = 1$ there is one twiddle factor and so on upto $ns = r^{k-1}$. So the total number of distinct twiddle factors for all iterations (i.e., elements $ns \in NS$) is:

$$1 + r + r^2 + \cdots + r^{k-1} = \frac{r^k - 1}{r - 1} = \frac{n - 1}{r - 1}$$

```
 1  let fft_precomp' [n] (forward: f32) (input: [n]complex) (bits: i32) : []complex =
 2    let input  = intrinsics.cosmin_flatten
 3                 (copy (intrinsics.unflatten (n/radix, radix, input)))
 4    let output = intrinsics.cosmin_flatten
 5                 (copy (intrinsics.unflatten (n/radix, radix, input)))
 6
 7    -- Precomputation of twiddle factors
 8    let wk_ns =
 9      map (\ind ->
10        let (n',ind') =
11          loop (s,ind) = (1,ind) while ind >= 0 do
12            (s * radix, ind - s)
13        let k' = ind' + (n' / radix)
14        let angle = (-2f32 * forward * f32.pi * (r32 k')) / (r32 n')
15        in  complex.mk (f32.cos angle) (f32.sin angle)
16      ) (iota ((n-1)/(radix-1)))
17
18    let ix = iota(n/radix)
19    --let NS = map (radix**) (iota bits)
20    let (res,_,_) =
21    loop (input': *[n]complex, output': *[n]complex, offset: i32) =(input, output, 0)
22        for i < bits do
23          let ns = radix ** i
24          let (inds_vals: Iter.tuple_inds_vals) =
25            unsafe (Iter.unzip_it
26              (map
27                (Iter.fft_iter_precomp ns offset wk_ns input') ix))
28          in (scatter output' (Iter.concat_inds inds_vals)
29                              (Iter.concat_vals inds_vals),
30                                input', offset+(radix**i))
```

Listing 4.13: Global FFT main function

The pre-computation of the angles happens in the main FFT module inside the `fft_precomp'` function. To compute all angles prior to any FFT calculation, a `map` over an array of all needed angles is performed (lines 9-16). This works generically for all radix, as the only variables are the size of the input and the radix.

The rest of the function is quite similar to the `fft'` function (4.1.3), except that it has an offset parameter which is used in the iteration module to access the twiddle factor. This amounts in some code clones that should fixed in future revisions of the code.

```
1    let fft_iter_precomp [n] (ns: i32) (offset: i32) (wk_ns: []complex)
2                             (data: [n]complex) (j: i32) : (iter_ret) =
3      let k = j % ns
4      let wk_n = unsafe wk_ns[offset+k]
5      let (v0, v1) = (data[j], data[j+n/radix] complex.* wk_n)
6      let (v0, v1) =  (v0 complex.+ v1, v0 complex.- v1)
7      let idxD = ((j/ns)*ns*radix) + (j % ns)
8      in (idxD, idxD+ns, v0, v1)
```

Listing 4.14: Global FFT iteration radix-2

There is no big difference between the iteration with pre-computed twiddle factors and without, except that the twiddle factor does not need to be calculated. Instead, it is taken from `wk_ns[offset+k]` at line 4, which requires *offset* as a parameter.

4.2.2 Implementation of Hierarchical FFTs

This section looks at the implementation of the hierarchical FFT algorithm with factorization as described in subsection 3.2.3.

```
module fft_sharedmem  : (fft_planex) = {
  type n_t    = f32
  type plan_t = {rad: i32, block: i32, num: i32}

  let planner (n) (block_sz: i32) : plan_t =
    -- Always use r4 if input is power of 4
    let plan_rad = if (fft_mod4.powOfR n == true) then 4 else 2
    let plan_block    = (plan_rad * block_sz)
    let plan_num = if (n <= plan_block) then 0
                   else 1
    in {rad = plan_rad, block = plan_block, num = plan_num}

  let executor [n] (data: [n]n_t) (plan: plan_t) : [n]n_t =
    let no_splitting (data': [n](n_t, n_t)) : [n](n_t, n_t) =
      if (plan.rad == 2) then fft_mod2.fft data' true
      else fft_mod4.fft data' true

    let splitting1 (data': [n](n_t, n_t)) (plan': plan_t) :
                                            [n](n_t, n_t) =
      let n1 = plan'.block
      let n2 = n/(plan'.block)
      let data' = unflatten n1 n2 data'
      let data' = if (plan'.rad == 2) then fft_mod2.fact data' n
                  else fft_mod4.fact data' n
      in data'
    let data_real = map (\r -> complex.mk_re r) data
    let res = if (plan.num == 0) then (no_splitting data_real)
              else (splitting1 data_real plan)
    in map (\r -> complex.re r) res

  let planex [n] (data:[n]n_t) (block_sz: i32) : [n]n_t =
    let plan = planner n block_sz
    in executor data plan

  let planex_batch [n][m] (data: [n][m]n_t) (block_sz: i32) : [n][m]n_t =
    let plan = planner n block_sz
    in map (\f -> executor f plan) data
}
```

Listing 4.15: Shared FFT planner/executor module

In order to use shared memory, the size of the input N_x on each block should be at most the product of the radix and the block size: $N_x \leq r \cdot bs$. This is because the input is divided into r FFTs of size N/r, so they can be run in parallel on different blocks. This suggests that factorization as described in subsection 3.2.3 should be used. The planner calculates the maximum block size size and saves it as a part of the planner object (line 8). Only radix-2 and radix-4 can be used, and this will be explained in chapter 5, as it has to do with the block size for the machines that the tests were perfomed on.

Furthermore, the plan includes the information of how often the input should be split. If the input is smaller than the block size, it doesn't need to split, but otherwise it does (lines 9-10). As splitting is only implemented one (i.e. one split into two factors), this is a binary choice. Finally, the plan is executed in the executor.

The executor transforms the input into complex data and checks whether the input should be split or not, if not it uses the regular global memory FFT with precomputation of the twiddle factors (line 15). Otherwise, it splits the data greedily into a 2-D matrix of size N_1 x N_2, where N_1 is the highest possible number, i.e. the block size. N_2 is the remaining values of the input (line 21). It then uses the factorization function discussed below, and in the end transforms the result back to real output.

```
 1  let fact [n1][n2] (data: [n1][n2](f32,f32)) (n: i32) : [](f32, f32) =
 2    let data = transpose data
 3    let data = map (\d -> fft d false) data
 4    -- multiplication with twiddle factors
 5    let data = map (\(j2,row) ->
 6                  map (\(i1,v) ->
 7                    let angle = (-2f32 * f32.pi * (r32 i1) * (r32 j2)) / (r32 n)
 8                    let twiddle = complex.mk (f32.cos angle) (f32.sin angle)
 9                    in v complex.* twiddle
10                   ) (zip (iota n1) row)
11                ) (zip (iota n2) data)
12    let data = transpose data
13    let data = map (\d -> fft d false) data
14    in flatten (transpose data)
```

Listing 4.16: Hierarchical FFT main function

This function in the main FFT module corresponds to the factorization algorithm described in subsection 3.2.3. Lines 2-3 implement step 1, N_2 column-wise FFTs, by computing the row-wise FFTs on the transposed array, which is transposed back in line 12. Lines 5-11 implement step 2, multiplication with the twiddle factors $W_N^{k_1 n_2}$. Line 13 implements step 3, N_1 row-wise FFTs and finally, step 4 is performed by transposing the entire matrix one last time.

Chapter 5

Evaluation

This chapter presents the performance results of the implementation discussed in chapter 4. Tests were performed on the DIKU GPU1 machine with an NVIDIA GK110B GPU, as well as an AMD machine with a FirePro AMD GPU. The exact test results can be found in the appendix. For each test, a baseline result for performance on the CPU is provided to compare with. As the current implementation does not support padding, the applicable data-sets for each radix varied.

5.1 AMD

5.1.1 Method

Benchmarking for global memory was done using the Futhark benchmarking tool `futhark-bench` on constant work test files. All test files were compiled using `--compiler=futhark-opencl`, except when generating data to show the CPU performance, in which case the `futhark-c` compiler was used. The size N of the input files was 2^k where $k = [6, 10, 15, 16, 18, 20, 22, 24]$. The constant work test files create a testing infrastructure that generates an equal amount of work for each size by creating a map of size W on top, so that the workload is that the highest k (in our case, it's 24). Recalling that the time complexity / work load of FFT is $\mathcal{O}N \lg N$, it should be $24 \cdot 2^{24}$ for an input of size 2^{24}. Then for k < 24, W can be found in the following way:

$$\begin{aligned} W \cdot k2^k &= 24 \cdot 2^{24} \\ W &= (24 \cdot 2^{24})/(k2^k) \\ W &= (24 \cdot 2^{24})/(N \lg N) \end{aligned}$$

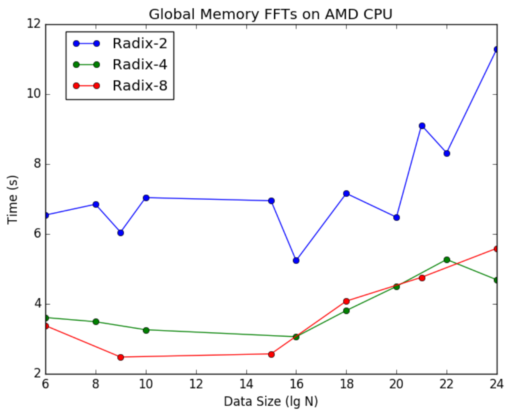
Hierarchical FFTs were not tested on the AMD GPU, as the results are clear from the NVIDIA test in the next section and we didn't expect an improvement on AMD. Instead, a different kind of shared memory test was executed by compiling the input with the flag `FUTHARK_INCREMENTAL_FLATTENING=1`. This enables shared memory access and thus only works for low inputs, i.e. where the data-set is smaller than the block size.

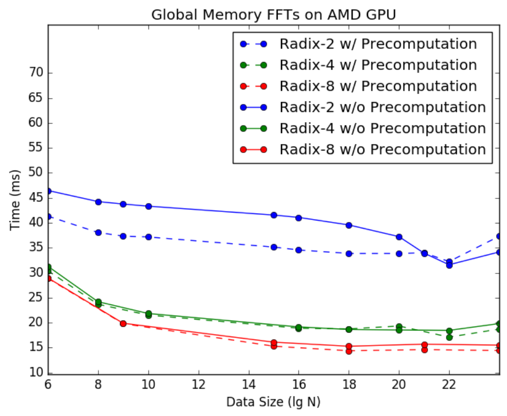
It was not possible to provide a cuFFT baseline, as cuFFT only works on NVIDIA machines.

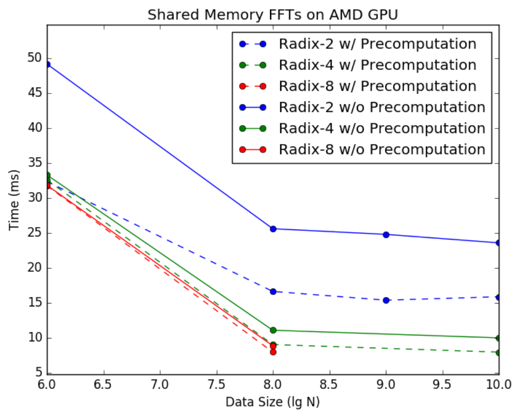
5.1.2 Expectations

- Higher radix to mostly perform better than lower radix.
- Shared memory to mostly perform better than global memory.
- Pre-computation of twiddle factors to perform better than none.

5.1.3 Results and Discussion







Again, increasing the radix improves the performance in most cases. As on the NVIDIA machine, radix-4 performs around twice as well as radix-2, while the difference between radix-4 and radix-8 is much smaller. Contrary to the NVIDIA GPU test, twiddle factor pre-computation does give a speed-up on the AMD GPU. It is especially significant for radix-2, and generally most impressive in shared memory.

The shared memory implementation is quite effective, giving just close to a 3x speed-up compared to global memory in the best cases. Unfortunately it is applicable on only a few datasets. However, radix-8 performs almost the same as radix-4 in shared memory. This is because the shared memory version uses $r \cdot 2 \cdot 2$ (r=radius) words per thread. When r=8, this amount is too high and affects occupancy; there will not be enough memory space for the right amount of blocks on the device. One way to fix this might be to pre-compute all twiddle factors and not just the angles.

5.2 NVIDIA

5.2.1 Method

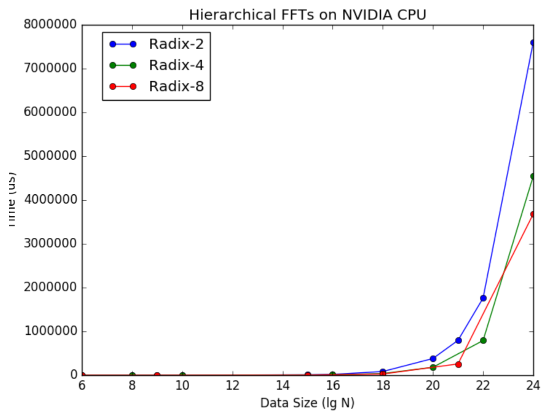
Testing on the NVIDIA GPU was done in the same way as on the AMD GPU, with a few differences:

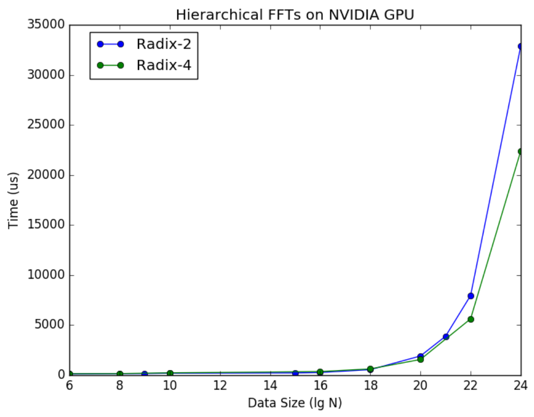
For hierarchical FFTs with factorization, benchmarking was done without constant work, in order to show how the current implementation is only working optimally for a limited input size upto ca. 2^{22}. To compare our implementation to the CUDA FFT library cuFFT, a small program was written in CUDA that creates a batch of FFTs like the one used in the Futhark tests.

5.2.2 Expectations

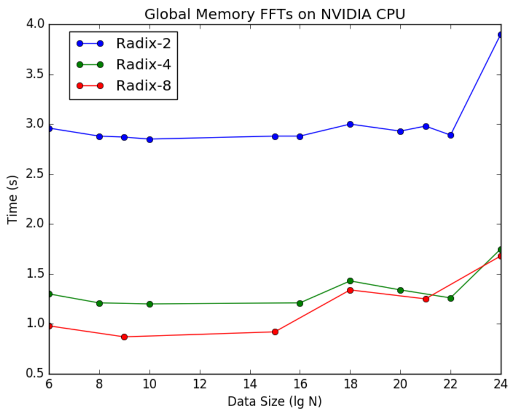
- Higher radix to mostly perform better than lower radix.
- Hierarchical FFT implementation to be optimal up to input size of 2^{22}.
- Pre-computation of twiddle factors to perform better than none.

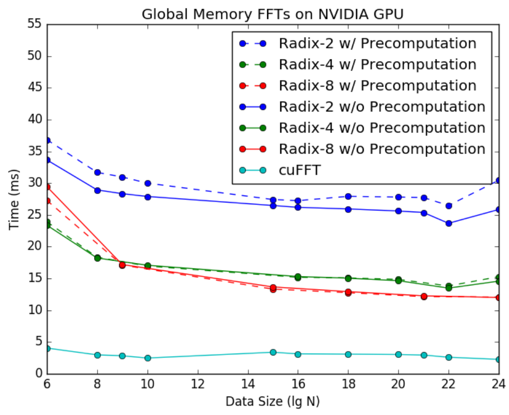
5.2.3 Results and Discussion





The performance of the hierarchical FFT implementation decreases after passing input size 2^{22}. This was to be expected, as the implementation can only split the input into two factors. Recalling that the size of one factor N_x should be $N_x \leq r \cdot bs$ (see subsection 4.2.2), where $bs = 1024$ on NVIDIA, the result will be $N_x \leq 2^{11}$ for radix-2 and $N_x \leq 2^{12}$ for radix-4. The highest input to be split upto once in this way is then $N = 2^{11} \cdot 2^{11} = 2^{22}$ for radix-2 and $N = 2^{12} \cdot 2^{12} = 2^{24}$ for radix-4. The input is quite flat until it reaches N_x, since it uses the global memory implementation until that point.





As expected, a higher radix usually accounts for better performance, both on the CPU and GPU. However, the increase in performance is significantly higher when comparing radix-2 and 4 (ca. 2x speed-up), than radix-4 and 8. Interestingly, the pre-computation of twiddle factors did not give the anticipated optimization. For radix-2, it gives a visible overhead, while it gives around the same performance as no pre-computation for radix-4 and 8. This suggests that on an NVIDIA machine, it is more costly to access the stored twiddle factors in memory, than to compute them ”on the go”.

Chapter 6

Conclusion

The thesis presents the implementation of a modular, user-transparent FFT library in Futhark, using a generic representation of the radix to minimize code clones. The library analyzes the input data-set and discriminates the optimal implementation to calculate its FFT.

The results show that the radix has a significant effect on the performance, with a trend of higher radix giving higher performance. The improvement was more noticeable between radix-2 and 4, than between radix-4 and 8. The effect of twiddle factor pre-computation depends on the specific hardware. For NVIDIA, it lessened the performance, indicating that it is more costly to access the stored twiddle factors in memory, than to compute them ”on the go”. However, pre-computing them did give a speed-up of up to 2x on AMD GPU. Factorization also seems like a promising approach, but as its implementation was restricted, it wasn’t possible to fully test the effects.

There are still ample improvements to be done to the implementation presented in this thesis. Most prominently, it will be interesting to see the results for implementing factorization that splits the input data into more than just two components, since the results for one splitting are promising. Furthermore, there are still some code clones in the implementation, particularly in the FFT iteration module. A minimization of these is desirable. Lastly, one can experiment with padding of the input in order to be able to use the optimal radix for a given input, since at this point, a radix-X can only be used on input that is a power of X.

Bibliography

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

[2] Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High performance discrete fourier transforms on graphics processors. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 2:1–2:12, Piscataway, NJ, USA, 2008. IEEE Press.

[3] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates. *SIGPLAN Not.*, 52(6):556–571, June 2017.

[4] James W. Cooley and John W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comput.*, 19:297–301, 1965.

[5] Akira Nukada, Yasuhiko Ogata, Toshio Endo, and Satoshi Matsuoka. Bandwidth intensive 3-d fft kernel for gpus using cuda. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 5:1–5:11, Piscataway, NJ, USA, 2008. IEEE Press.

[6] Matteo Frigo. A fast fourier transform compiler. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 169–180, New York, NY, USA, 1999. ACM.

[7] David P.H. Jørgensen and Kasper A. Hansen. Bachelor thesis: Fast fourier transformation in furthark, January 2018.

[8] Troels Henriksen and Cosmin E. Oancea. Bounds Checking: An Instance of Hybrid Analysis. In *Procs. of ACM SIGPLAN Int. Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, pages 88:88–88:94, New York, NY, USA, 2014. ACM.

[9] Troels Henriksen, Martin Elsman, and Cosmin E. Oancea. Size Slicing: A Hybrid Approach to Size Inference in Futhark. In *Procs. of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC'14, pages 31–42, New York, NY, USA, 2014. ACM.

[10] Troels Henriksen, Martin Dybdal, Henrik Urms, Anna Sofie Kiehn, Daniel Gavin, Hjalte Abelskov, Martin Elsman, and Cosmin Oancea. APL on GPUs: A TAIL from the Past, Scribbled in Futhark. In *Procs. of the 5th Int. Workshop on*

Functional High-Performance Computing, FHPC'16, pages 38–43, New York, NY, USA, 2016. ACM.

[11] Bernard Legrand. *Mastering Dyalog APL*. Dyalog Limited, November 2009.

[12] Cosmin E. Oancea and Stephen M. Watt. Domains and expressions: An interface between two approaches to computer algebra. In *Proceedings of the 2005 International Symposium on Symbolic and Algebraic Computation*, ISSAC '05, pages 261–268, New York, NY, USA, 2005. ACM.

[13] Y. Chicha, M. Lloyd, C. Oancea, and S. M. Watt. Parametric Polymorphism for Computer Algebra Software Components. In *Proc. 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Comput.*, pages 119–130. Mirton Publishing House, 2004.

[14] Troels Henriksen, Ken Friis Larsen, and Cosmin E. Oancea. Design and GPGPU Performance of Futhark's Redomap Construct. In *Procs. of the 3rd ACM SIGPLAN Int. Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'16, pages 17–24, New York, NY, USA, 2016. ACM.

[15] Rasmus Wriedt Larsen and Troels Henriksen. Strategies for regular segmented reductions on gpu. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, FHPC 2017, pages 42–52, New York, NY, USA, 2017. ACM.

[16] Cosmin E. Oancea, Christian Andreetta, Jost Berthold, Alain Frisch, and Fritz Henglein. Financial Software on GPUs: Between Haskell and Fortran. In *Proceedings of the 1st ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '12, pages 61–72, New York, NY, USA, 2012. ACM.

[17] D. Sundararajan. *The Discrete Fourier Transform: Theory, Algorithms and Applications*. World Scientific, 2001.

[18] W. T. Cochran, J. W. Cooley, D. L. Favin, H. D. Helms, R. A. Kaenel, W. W. Lang, G. C. Maling, D. E. Nelson, C. M. Rader, and P. D. Welch. What is the fast fourier transform? *Proceedings of the IEEE*, 55(10):1664–1674, Oct 1967.

Appendix A

Code

A.1 FFT Iteration Module

```
import "/futlib/complex"
module complex = complex f32
type complex = complex.complex

module type fft_iteration = {
  type tuple_inds_vals
  type iter_ret

  val radix: i32
  -- | Concatenate the specified tuples of indexes or values;
  -- | result type is not specialized, as it needs to fit into scatter
  val concat_inds: tuple_inds_vals -> []i32
  val concat_vals: tuple_inds_vals -> []complex

  -- | A regular unzip returns an array of tuples, but this one
  -- returns the specific type we created
  val unzip_it: [](iter_ret) -> tuple_inds_vals
  -- | One iteration of the fft loop
  val fft_iter: f32 -> i32 -> []complex -> i32 -> iter_ret
  -- | One iteration of the fft loop with pre-computed twiddles
  val fft_iter_precomp: i32 -> i32 -> []complex -> []complex
                        -> i32 -> iter_ret
}

-- | Given a radix value, construct a module
-- | implementing specialized fft_iteration for that value

module fft_iteration2: (fft_iteration) = {
  type ind = i32
  type vl = complex
  type inds = []ind
  type vals = []vl
  type tuple_inds_vals = (inds, inds, vals, vals)
  type iter_ret = (ind, ind, vl, vl)

  let radix = 2i32
  let concat_inds ((a,b,_,_): tuple_inds_vals) = concat a b
  let concat_vals ((_,_,c,d): tuple_inds_vals) = concat c d

  let unzip_it (x: [](iter_ret)) = unzip x
  let fft_iter [n] (forward: f32) (ns: i32) (data: [n]complex) (j: i32) : (iter_ret) =
    let angle = (-2f32 * forward * f32.pi * r32 (j % ns)) / r32 (ns * radix)
    let (v0, v1) = (data[j], data[j+n/radix] complex.*
                   (complex.mk (f32.cos angle) (f32.sin angle)))
    let (v0, v1) =  (v0 complex.+ v1, v0 complex.- v1)
    let idxD = ((j/ns)*ns*radix) + (j % ns)
    in (idxD, idxD+ns, v0, v1)

  let fft_iter_precomp [n] (ns: i32) (offset: i32) (wk_ns: []complex)
                           (data: [n]complex) (j: i32) : (iter_ret) =
    let k = j % ns
    let wk_n = unsafe wk_ns[offset+k]
    let (v0, v1) = (data[j], data[j+n/radix] complex.* wk_n)
    let (v0, v1) =  (v0 complex.+ v1, v0 complex.- v1)
    let idxD = ((j/ns)*ns*radix) + (j % ns)
    in (idxD, idxD+ns, v0, v1)
}

module fft_iteration4: (fft_iteration) = {
  type ind = i32
  type vl = complex
  type inds = []ind
  type vals = []vl
  type tuple_inds_vals = (inds, inds, inds, inds,
                          vals, vals, vals, vals)
  type iter_ret = (ind, ind, ind, ind, vl, vl, vl, vl)

  let radix = 4i32
  let concat_inds ((a,b,c,d,_,_,_,_): tuple_inds_vals) = a ++ b ++ c ++ d
  let concat_vals ((_,_,_,_,a,b,c,d): tuple_inds_vals) = a ++ b ++ c ++ d

  let unzip_it (x: [](iter_ret)) = unzip x
  let twiddle (a: complex) : complex =
    complex.mk (complex.im a) (- (complex.re a))

  let fft_iter [n] (forward: f32) (ns: i32) (data: [n]complex) (j: i32)
                                                    : (iter_ret) =
    let angle = (-2f32 * forward * f32.pi * r32 (j%ns)) / r32 (ns * radix)
    let tw = complex.mk (f32.cos angle) (f32.sin angle)
    let a0 = data[j]
    let a1 = tw complex.* (data[j+n/radix])
    let a2 = data[j+2*n/radix]
    let a3 = tw complex.* (data[j+3*n/radix])

    let tw = tw complex.* tw
    let a2 = tw complex.* a2
    let a3 = tw complex.* a3

    let b0 = a0 complex.+ a2
    let b1 = a0 complex.- a2
    let b2 = a1 complex.+ a3
    let b3 = twiddle (a1 complex.- a3)

    let v0 = b0 complex.+ b2
    let v2 = b0 complex.- b2
    let v1 = b1 complex.+ b3
    let v3 = b1 complex.- b3

    let idxD = ((j/ns)*ns*radix) + (j % ns)
    in (idxD, idxD+ns, idxD+2*ns, idxD+3*ns, v0, v1, v2, v3)

  let fft_iter_precomp [n] (ns: i32) (offset: i32) (wk_ns: []complex)
                           (data: [n]complex) (j: i32) : (iter_ret) =
    let k = j % ns
    let wk_n = unsafe wk_ns[offset+k]
    let a0 = data[j]
    let a1 = wk_n complex.* (data[j+n/radix])
    let a2 = data[j+2*n/radix]
    let a3 = wk_n complex.* (data[j+3*n/radix])

    let tw = wk_n complex.* wk_n
    let a2 = tw complex.* a2
    let a3 = tw complex.* a3

    let b0 = a0 complex.+ a2
    let b1 = a0 complex.- a2
    let b2 = a1 complex.+ a3
    let b3 = twiddle (a1 complex.- a3)

    let v0 = b0 complex.+ b2
    let v2 = b0 complex.- b2
    let v1 = b1 complex.+ b3
    let v3 = b1 complex.- b3
    let idxD = ((j/ns)*ns*radix) + (j % ns)
    in (idxD, idxD+ns, idxD+2*ns, idxD+3*ns, v0, v1, v2, v3)
}

module fft_iteration8: (fft_iteration) = {
  type ind = i32
  type vl = complex
  type inds = []ind
  type vals = []vl
  type tuple_inds_vals = (inds, inds, inds, inds,
                          inds, inds, inds, inds,
                          vals, vals, vals, vals,
                          vals, vals, vals, vals)
  type iter_ret = (ind, ind, ind, ind, ind, ind, ind, ind,
                   vl, vl, vl, vl, vl, vl, vl, vl)
  let radix = 8i32
  let concat_inds ((a,b,c,d,e,f,g,h,_,_,_,_,_,_,_,_):
    tuple_inds_vals) = a ++ b ++ c ++ d ++ e ++ f ++ g ++ h
  let concat_vals ((_,_,_,_,_,_,_,_,a,b,c,d,e,f,g,h):
    tuple_inds_vals) =  a ++ b ++ c ++ d ++ e ++ f ++ g ++ h

  let unzip_it (x: [](iter_ret)) = unzip x

  let sqrt2 = complex.mk (0.707106781188f32) (0.0f32)

  let twiddle (a: complex) : complex =
    complex.mk (complex.im a) (- (complex.re a))

  let mul_p1q4(a: complex) : complex =
    let x = complex.mk (complex.re a + complex.im a)
               (complex.im a - complex.re a)
    in x complex.* sqrt2

  let mul_p3q4(a: complex) : complex =
    let x = complex.mk (complex.im a - complex.re a)
                (-(complex.re a) - complex.im a)
    in x complex.* sqrt2

  let fft_iter [n] (forward: f32) (ns: i32) (data: [n]complex) (j: i32)
                                                           : (iter_ret) =
    let angle = (-2f32 * forward * f32.pi * r32 (j%ns)) / r32 (ns * radix)
    let tw = complex.mk (f32.cos angle) (f32.sin angle)
    let a0 = data[j]
    let a1 = tw complex.* (data[j+n/radix])
    let a2 = data[j+2*n/radix]
    let a3 = tw complex.* (data[j+3*n/radix])
    let a4 = data[j+4*n/radix]
    let a5 = tw complex.* (data[j+5*n/radix])
    let a6 = data[j+6*n/radix]
    let a7 = tw complex.* (data[j+7*n/radix])

    let tw = tw complex.* tw --W^2
    let a2 = tw complex.* a2
    let a3 = tw complex.* a3
    let a6 = tw complex.* a6
    let a7 = tw complex.* a7
    let tw = tw complex.* tw --W^4
    let a4 = tw complex.* a4
    let a5 = tw complex.* a5
    let a6 = tw complex.* a6
    let a7 = tw complex.* a7

    let b0 = a0 complex.+ a4
    let b4 = a0 complex.- a4
    let b1 = a1 complex.+ a5
    let b5 = mul_p1q4 (a1 complex.- a5)
    let b2 = a2 complex.+ a6
    let b6 = twiddle (a2 complex.- a6)
    let b3 = a3 complex.+ a7
    let b7 = mul_p3q4 (a3 complex.- a7)

    let a0 = b0 complex.+ b2
    let a2 = b0 complex.- b2
    let a1 = b1 complex.+ b3
    let a3 = twiddle(b1 complex.- b3)
    let a4 = b4 complex.+ b6
    let a6 = b4 complex.- b6
    let a5 = b5 complex.+ b7
    let a7 = twiddle(b5 complex.- b7)

    let v0 = a0 complex.+ a1
    let v4 = a0 complex.- a1
    let v1 = a4 complex.+ a5
    let v5 = a4 complex.- a5
    let v2 = a2 complex.+ a3
    let v6 = a2 complex.- a3
    let v3 = a6 complex.+ a7
    let v7 = a6 complex.- a7

    let idxD = ((j/ns)*ns*radix) + (j % ns)
    in (idxD, idxD+ns, idxD+2*ns, idxD+3*ns,
        idxD+4*ns, idxD+5*ns, idxD+6*ns, idxD+7*ns,
        v0, v1, v2, v3, v4, v5, v6, v7)


  let fft_iter_precomp [n] (ns: i32) (offset: i32) (wk_ns: []complex)
                            (data: [n]complex) (j: i32) : (iter_ret) =
    let k = j % ns
    let wk_n = unsafe wk_ns[offset+k]
    let a0 = data[j]
    let a1 = wk_n complex.* (data[j+n/radix])
    let a2 = data[j+2*n/radix]
    let a3 = wk_n complex.* (data[j+3*n/radix])
    let a4 = data[j+4*n/radix]
    let a5 = wk_n complex.* (data[j+5*n/radix])
    let a6 = data[j+6*n/radix]
    let a7 = wk_n complex.* (data[j+7*n/radix])

    let tw = wk_n complex.* wk_n --W^2
    let a2 = tw complex.* a2
    let a3 = tw complex.* a3
    let a6 = tw complex.* a6
    let a7 = tw complex.* a7
    let tw = tw complex.* tw --W^4
    let a4 = tw complex.* a4
    let a5 = tw complex.* a5
    let a6 = tw complex.* a6
    let a7 = tw complex.* a7

    let b0 = a0 complex.+ a4
    let b4 = a0 complex.- a4
    let b1 = a1 complex.+ a5
    let b5 = mul_p1q4 (a1 complex.- a5)
    let b2 = a2 complex.+ a6
    let b6 = twiddle (a2 complex.- a6)
    let b3 = a3 complex.+ a7
    let b7 = mul_p3q4 (a3 complex.- a7)

    let a0 = b0 complex.+ b2
    let a2 = b0 complex.- b2
    let a1 = b1 complex.+ b3
    let a3 = twiddle(b1 complex.- b3)
    let a4 = b4 complex.+ b6
    let a6 = b4 complex.- b6
    let a5 = b5 complex.+ b7
    let a7 = twiddle(b5 complex.- b7)

    let v0 = a0 complex.+ a1
    let v4 = a0 complex.- a1
    let v1 = a4 complex.+ a5
    let v5 = a4 complex.- a5
    let v2 = a2 complex.+ a3
    let v6 = a2 complex.- a3
    let v3 = a6 complex.+ a7
    let v7 = a6 complex.- a7
```

```
    let idxD = ((j/ns)*ns*radix) + (j % ns)
    in (idxD, idxD+ns, idxD+2*ns, idxD+3*ns,
        idxD+4*ns, idxD+5*ns, idxD+6*ns, idxD+7*ns,
        v0, v1, v2, v3, v4, v5, v6, v7)
}
```

A.2 FFT Main Module

```
import "fft_iteration"

module type fft_module = {
  val radix: i32
  -- | Find out whether input size is a power of the radix
  val powOfR: i32 -> bool
  -- | FFT of complex numbers
  val fft [n]: [n](f32, f32) -> bool -> [n](f32, f32)
  -- | FFT of 2-D array of complex numbers for factorization
  val fact: [][](f32,f32) -> i32 -> [](f32, f32)
  -- | FFT of real numbers
  val fft_real [n]: [n]f32 -> bool -> [n]f32
}

module fft_module(Iter: fft_iteration): fft_module = {
  let radix = Iter.radix

  let powOfR (n: i32) : bool =
    let x = 0i32
    let (x, _) = loop (x,n) while ((1 < n) && (x == 0)) do
      let x = n % radix
      let n = n / radix
      in (x, n)
    let res = if (x == 0) then true else false
    in res


  let fft' [n] (forward: f32) (input: [n]complex) (bits: i32) : [n]complex =
    let input  = intrinsics.cosmin_flatten (copy (intrinsics.unflatten
                                               (n/radix, radix, input)))
    let output = intrinsics.cosmin_flatten (copy (intrinsics.unflatten
                                               (n/radix, radix, input)))
    let ix = iota(n/radix)
    let NS = map (radix**) (iota bits)
    let (res,_) =
      loop (input': *[n]complex, output': *[n]complex) = (input, output)
      for ns in NS do
        let (inds_vals: Iter.tuple_inds_vals) =
          unsafe (Iter.unzip_it
            (map (Iter.fft_iter forward ns input') ix))
        in (scatter output' (Iter.concat_inds inds_vals)
                            (Iter.concat_vals inds_vals), input')
    in res

  let fft_precomp' [n] (forward: f32) (input: [n]complex) (bits: i32) : []complex =
    let input  = intrinsics.cosmin_flatten
                   (copy (intrinsics.unflatten (n/radix, radix, input)))
    let output = intrinsics.cosmin_flatten
                   (copy (intrinsics.unflatten (n/radix, radix, input)))

    -- Precomputation of twiddle factors
    let wk_ns =
      map (\ind ->
        let (n',ind') =
          loop (s,ind) = (1,ind) while ind >= 0 do
            (s * radix, ind - s)
        let k' = ind' + (n' / radix)
        let angle = (-2f32 * forward * f32.pi * (r32 k')) / (r32 n')
        in  complex.mk (f32.cos angle) (f32.sin angle)
      ) (iota ((n-1)/(radix-1)))

    let ix = iota(n/radix)
    --let NS = map (radix**) (iota bits)
    let (res,_,_) =
    loop (input': *[n]complex, output': *[n]complex, offset: i32) =(input, output, 0)
        for i < bits do
          let ns = radix ** i
          let (inds_vals: Iter.tuple_inds_vals) =
            unsafe (Iter.unzip_it
              (map
                (Iter.fft_iter_precomp ns offset wk_ns input') ix))
          in (scatter output' (Iter.concat_inds inds_vals)
                              (Iter.concat_vals inds_vals),
                                  input', offset+(radix**i))
      in res

  let logR (n: i32) : i32 =
    let r = 0
      let (r, _) = loop (r,n) while 1 < n do
        let n = n / radix
        let r = r + 1
        in (r,n)
    in r

  let generic_fft [n] (forward: bool) (data: [n](f32,f32)) (precomp: bool):
                                                        [n](f32,f32) =
    let n_bits = logR n
    let forward' = if forward then 1f32 else -1f32
    in if (precomp == false) then take n (fft' forward' data n_bits)
    else take n (fft_precomp' forward' data n_bits)

  let fft [n] (data: [n](f32, f32)) (precomp: bool): [n](f32, f32) =
    generic_fft true data precomp

  let fact [n1][n2] (data: [n1][n2](f32,f32)) (n: i32) : [](f32, f32) =
    let data = transpose data
    let data = map (\d -> fft d false) data
    -- multiplication with twiddle factors
    let data = map (\(j2,row) ->
                 map (\(i1,v) ->
                   let angle = (-2f32 * f32.pi * (r32 i1) * (r32 j2)) / (r32 n)
                   let twiddle = complex.mk (f32.cos angle) (f32.sin angle)
                   in v complex.* twiddle
                  ) (zip (iota n1) row)
               ) (zip (iota n2) data)
    let data = transpose data
    let data = map (\d -> fft d false) data
    in flatten (transpose data)

  let fft_real [n] (data: [n]f32) (precomp: bool): ([n]f32) =
    map (\r -> complex.re r)
        (fft (map (\r -> complex.mk_re r) data) (precomp))
}
```

A.3 Planner-Executor Module

```
import "fft_module"
import "fft_iteration"

module iteration2 = fft_iteration2
module fft_mod2 = fft_module(iteration2)

module iteration4 = fft_iteration4
module fft_mod4 = fft_module(iteration4)

module iteration8 = fft_iteration8
module fft_mod8 = fft_module(iteration8)

module type fft_planex = {
  type n_t
  -- | A record of information about the plan for the input
  type plan_t

  -- | Create a plan how to calculate the fft
  val planner: i32 -> i32 -> plan_t
  -- | Execute the plan; calculate the fft
  val executor [n]: [n]n_t -> plan_t -> [n]n_t
  val planex [n]: [n]n_t -> i32 -> [n]n_t
  val planex_batch [n][m]: [n][m]n_t -> i32 -> [n][m]n_t
}

module fft_globalmem  : (fft_planex) = {
  type n_t = f32
  type plan_t = {rad: i32}

  let planner (n) (block_sz: i32) : plan_t =
  -- Preferrably use radix 8
    let plan_rad = if      (fft_mod8.powOfR n == true) then 8
                   else if (fft_mod4.powOfR n == true) then 4 else 2
    in {rad = plan_rad}

  let executor [n] (data: [n]n_t) (plan: plan_t) : [n]n_t =
    if      (plan.rad == 8) then fft_mod8.fft_real data true
    else if (plan.rad == 4) then fft_mod4.fft_real data true
    else                         fft_mod2.fft_real data true

  let planex [n] (data:[n]n_t) (block_sz: i32) =
    let plan = planner n block_sz
    in executor data plan

  let planex_batch [n][m] (data: [n][m]n_t) (block_sz: i32) : [n][m]n_t =
    let plan = planner n block_sz
    in map (\f -> executor f plan) data
}

module fft_sharedmem  : (fft_planex) = {
  type n_t    = f32
  type plan_t = {rad: i32, block: i32, num: i32}

  let planner (n) (block_sz: i32) : plan_t =
    -- Always use r4 if input is power of 4
    let plan_rad = if (fft_mod4.powOfR n == true) then 4 else 2
    let plan_block    = (plan_rad * block_sz)
    let plan_num = if (n <= plan_block) then 0
                   else 1
    in {rad = plan_rad, block = plan_block, num = plan_num}

  let executor [n] (data: [n]n_t) (plan: plan_t) : [n]n_t =
    let no_splitting (data': [n](n_t, n_t)) : [n](n_t, n_t) =
      if (plan.rad == 2) then fft_mod2.fft data' true
      else fft_mod4.fft data' true

    let splitting1 (data': [n](n_t, n_t)) (plan': plan_t) :
                                          [n](n_t, n_t) =
      let n1 = plan'.block
      let n2 = n/(plan'.block)
      let data' = unflatten n1 n2 data'
      let data' = if (plan'.rad == 2) then fft_mod2.fact data' n
                  else fft_mod4.fact data' n
      in data'
    let data_real = map (\r -> complex.mk_re r) data
    let res = if (plan.num == 0) then (no_splitting data_real)
              else (splitting1 data_real plan)
    in map (\r -> complex.re r) res

  let planex [n] (data:[n]n_t) (block_sz: i32) : [n]n_t =
    let plan = planner n block_sz
    in executor data plan

  let planex_batch [n][m] (data: [n][m]n_t) (block_sz: i32) : [n][m]n_t =
    let plan = planner n block_sz
    in map (\f -> executor f plan) data
}
```

Appendix B

Benchmarks

B.1 AMD

```
---------------------
-- FirePro AMD GPU --
---------------------

-----------------------------------------------------------------------
-----------------------------------------------------------------------
-- I. RADIX = 2

1. CPU RUNTIMES:
dataset data/input2pow6.in:  6536544.60us (avg. of 5 runs; RSD: 0.18)
dataset data/input2pow8.in:  6848591.50us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow9.in:  6046836.20us (avg. of 10 runs; RSD: 0.13)
dataset data/input2pow10.in: 7040947.40us (avg. of 5 runs; RSD: 0.02)
dataset data/input2pow15.in: 6947698.20us (avg. of 5 runs; RSD: 0.06)
dataset data/input2pow16.in: 5244199.40us (avg. of 5 runs; RSD: 0.10)
dataset data/input2pow18.in: 7160000.60us (avg. of 5 runs; RSD: 0.14)
dataset data/input2pow20.in: 6479949.00us (avg. of 5 runs; RSD: 0.10)
dataset data/input2pow21.in: 9106610.80us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow22.in: 8324531.40us (avg. of 5 runs; RSD: 0.14)
dataset data/input2pow24.in: 11294911.20us (avg. of 5 runs; RSD: 0.02)

2. GPU RUNTIMES: ONLY GLOBAL MEMORY, WITHOUT TWIDDLE PRECOMPUTATION
dataset data/input2pow6.in:    46470.07us (avg. of 30 runs; RSD: 0.05)
dataset data/input2pow8.in:    44250.77us (avg. of 30 runs; RSD: 0.07)
dataset data/input2pow9.in:    43748.83us (avg. of 30 runs; RSD: 0.08)
dataset data/input2pow10.in:   43328.47us (avg. of 30 runs; RSD: 0.08)
dataset data/input2pow15.in:   41551.50us (avg. of 30 runs; RSD: 0.08)
dataset data/input2pow16.in:   41082.73us (avg. of 30 runs; RSD: 0.08)
dataset data/input2pow18.in:   39574.67us (avg. of 30 runs; RSD: 0.08)
dataset data/input2pow20.in:   37232.23us (avg. of 30 runs; RSD: 0.06)
dataset data/input2pow21.in:   33934.27us (avg. of 30 runs; RSD: 0.00)
dataset data/input2pow22.in:   31599.93us (avg. of 30 runs; RSD: 0.01)
dataset data/input2pow24.in:   34159.67us (avg. of 30 runs; RSD: 0.01)
```

```
3. GPU RUNTIMES: ONLY GLOBAL MEMORY + TWIDDLE PRECOMPUTATION
dataset data/input2pow6.in:    41340.47us (avg. of 30 runs; RSD: 0.03)
dataset data/input2pow8.in:    38060.87us (avg. of 30 runs; RSD: 0.06)
dataset data/input2pow9.in:    37334.67us (avg. of 30 runs; RSD: 0.08)
dataset data/input2pow10.in:   37159.17us (avg. of 30 runs; RSD: 0.07)
dataset data/input2pow15.in:   35138.20us (avg. of 30 runs; RSD: 0.06)
dataset data/input2pow16.in:   34559.67us (avg. of 30 runs; RSD: 0.07)
dataset data/input2pow18.in:   33857.00us (avg. of 30 runs; RSD: 0.05)
dataset data/input2pow20.in:   33859.30us (avg. of 30 runs; RSD: 0.04)
dataset data/input2pow21.in:   34004.23us (avg. of 30 runs; RSD: 0.04)
dataset data/input2pow22.in:   32241.37us (avg. of 30 runs; RSD: 0.04)
dataset data/input2pow24.in:   37305.13us (avg. of 30 runs; RSD: 0.05)

4. GPU RUNTIMES: SHARED MEMORY, WITHOUT TWIDDLE PRECOMPUTATION
dataset data/input2pow6.in:    49213.30us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow8.in:    25643.40us (avg. of 10 runs; RSD: 0.07)
dataset data/input2pow9.in:    24826.20us (avg. of 10 runs; RSD: 0.06)
dataset data/input2pow10.in:   23624.50us (avg. of 10 runs; RSD: 0.05)

5. GPU RUNTIMES: SHARED MEMORY, WITH TWIDDLE PRECOMPUTATION
dataset data/input2pow6.in:    32400.80us (avg. of 10 runs; RSD: 0.03)
dataset data/input2pow8.in:    16671.10us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow9.in:    15419.00us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow10.in:   15909.60us (avg. of 10 runs; RSD: 0.10)

-----------------------------------------------------------------------
-----------------------------------------------------------------------
-- II. RADIX = 4

1. CPU RUNTIMES:
dataset data/input2pow6.in:  3607812.60us (avg. of 10 runs; RSD: 0.16)
dataset data/input2pow8.in:  3489618.00us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow10.in: 3255675.60us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow16.in: 3062832.20us (avg. of 10 runs; RSD: 0.10)
dataset data/input2pow18.in: 3811612.80us (avg. of 10 runs; RSD: 0.11)
dataset data/input2pow20.in: 4498532.80us (avg. of 10 runs; RSD: 0.04)
dataset data/input2pow22.in: 5273312.10us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow24.in: 4693372.00us (avg. of 10 runs; RSD: 0.03)

2. GPU RUNTIMES: ONLY GLOBAL MEMORY, WITHOUT TWIDDLE PRECOMPUTATION
dataset data/input2pow6.in:    31302.90us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow8.in:    24203.30us (avg. of 10 runs; RSD: 0.02)
dataset data/input2pow10.in:   21837.70us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow16.in:   19153.20us (avg. of 10 runs; RSD: 0.02)
dataset data/input2pow18.in:   18638.50us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow20.in:   18542.70us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow22.in:   18464.80us (avg. of 10 runs; RSD: 0.03)
dataset data/input2pow24.in:   19823.60us (avg. of 10 runs; RSD: 0.01)

3. GPU RUNTIMES: ONLY GLOBAL MEMORY + TWIDDLE PRECOMPUTATION
dataset data/input2pow6.in:    30409.00us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow8.in:    23688.20us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow10.in:   21517.40us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow16.in:   18891.60us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow18.in:   18747.10us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow20.in:   19328.20us (avg. of 10 runs; RSD: 0.03)
dataset data/input2pow22.in:   17114.40us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow24.in:   18720.80us (avg. of 10 runs; RSD: 0.01)

4. GPU RUNTIMES: SHARED MEMORY, WITHOUT TWIDDLE PRECOMPUTATION
dataset data/input2pow6.in:    33342.20us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow8.in:    11123.40us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow10.in:   10031.30us (avg. of 10 runs; RSD: 0.00)

5. GPU RUNTIMES: SHARED MEMORY, WITH TWIDDLE PRECOMPUTATION
dataset data/input2pow6.in:    32640.60us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow8.in:     9065.40us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow10.in:    7987.50us (avg. of 10 runs; RSD: 0.00)

-----------------------------------------------------------------------
-----------------------------------------------------------------------
-- III. RADIX 8

1. CPU RUNTIMES:
dataset data/input2pow6.in:  3379320.50us (avg. of 10 runs; RSD: 0.04)
dataset data/input2pow9.in:  2484598.40us (avg. of 10 runs; RSD: 0.17)
dataset data/input2pow15.in: 2569843.20us (avg. of 10 runs; RSD: 0.11)
dataset data/input2pow18.in: 4079942.80us (avg. of 10 runs; RSD: 0.10)
dataset data/input2pow21.in: 4763287.50us (avg. of 10 runs; RSD: 0.15)
dataset data/input2pow24.in: 5586481.90us (avg. of 10 runs; RSD: 0.08)

2. GPU RUNTIMES: ONLY GLOBAL MEMORY, WITHOUT TWIDDLE PRECOMPUTATION
dataset data/input2pow6.in:    28967.40us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow9.in:    19894.80us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow15.in:   16104.30us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow18.in:   15284.80us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow21.in:   15697.70us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow24.in:   15502.30us (avg. of 10 runs; RSD: 0.01)

3. GPU RUNTIMES: ONLY GLOBAL MEMORY + TWIDDLE PRECOMPUTATION
dataset data/input2pow6.in:    28892.00us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow9.in:    19811.90us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow15.in:   15292.20us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow18.in:   14375.50us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow21.in:   14597.80us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow24.in:   14419.70us (avg. of 10 runs; RSD: 0.01)

4. GPU RUNTIMES: SHARED MEMORY, WITHOUT TWIDDLE PRECOMPUTATION
dataset data/input2pow6.in:   31872.00us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow9.in:    8891.90us (avg. of 10 runs; RSD: 0.00)

5. GPU RUNTIMES: SHARED MEMORY, WITH TWIDDLE PRECOMPUTATION
dataset data/input2pow6.in:   31804.30us (avg. of 10 runs; RSD: 0.02)
dataset data/input2pow9.in:    7997.20us (avg. of 10 runs; RSD: 0.00)
```

B.2 NVIDIA

```
---------------------
-- NVIDIA GPU --
---------------------

-----------------------------------------------------------------------
-----------------------------------------------------------------------
-- I. RADIX = 2

1. CPU SPLIT Runtimes
dataset data/input2pow6.in:        6.90us (avg. of 10 runs; RSD: 0.14)
dataset data/input2pow8.in:       46.70us (avg. of 10 runs; RSD: 0.09)
dataset data/input2pow9.in:       99.00us (avg. of 10 runs; RSD: 0.08)
dataset data/input2pow10.in:     201.30us (avg. of 10 runs; RSD: 0.05)
dataset data/input2pow15.in:    8610.20us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow16.in:   18679.80us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow18.in:   83809.60us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow20.in:  381296.20us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow21.in:  795459.50us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow22.in: 1761053.80us (avg. of 10 runs; RSD: 0.04)
dataset data/input2pow24.in: 7602942.90us (avg. of 10 runs; RSD: 0.01)

2. GPU SPLIT Runtimes
dataset data/input2pow6.in:      115.20us (avg. of 10 runs; RSD: 0.10)
dataset data/input2pow8.in:      153.90us (avg. of 10 runs; RSD: 0.06)
dataset data/input2pow9.in:      162.50us (avg. of 10 runs; RSD: 0.07)
dataset data/input2pow10.in:     200.60us (avg. of 10 runs; RSD: 0.07)
dataset data/input2pow15.in:     225.60us (avg. of 10 runs; RSD: 0.07)
dataset data/input2pow16.in:     269.80us (avg. of 10 runs; RSD: 0.06)
dataset data/input2pow18.in:     552.80us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow20.in:    1918.40us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow21.in:    3884.50us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow22.in:    7955.20us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow24.in:   32874.10us (avg. of 10 runs; RSD: 0.02)

3. CPU Runtimes
dataset data/input2pow6.in:  2955597.30us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow8.in:  2884452.00us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow9.in:  2865708.40us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow10.in: 2851676.50us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow15.in: 2877989.90us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow16.in: 2879708.80us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow18.in: 2998358.40us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow20.in: 2930226.50us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow21.in: 2982417.20us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow22.in: 2885437.10us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow24.in: 3902226.60us (avg. of 10 runs; RSD: 0.00)

4. GPU RUNTIMES: GLOBAL MEMORY + TWIDDLE PRECOMPUTATION
dataset data/input2pow6.in:    36834.30us (avg. of 10 runs; RSD: 0.02)
dataset data/input2pow8.in:    31685.90us (avg. of 10 runs; RSD: 0.02)
dataset data/input2pow9.in:    30965.50us (avg. of 10 runs; RSD: 0.02)
dataset data/input2pow10.in:   30060.90us (avg. of 10 runs; RSD: 0.02)
dataset data/input2pow15.in:   27418.60us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow16.in:   27241.90us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow18.in:   27922.10us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow20.in:   27809.80us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow21.in:   27732.80us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow22.in:   26476.40us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow24.in:   30453.20us (avg. of 10 runs; RSD: 0.01)


5. GPU RUNTIMES: GLOBAL MEMORY, WITHOUT TWIDDLE PRECOMPUTATION
dataset data/input2pow6.in:    33650.70us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow8.in:    28934.00us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow9.in:    28337.80us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow10.in:   27931.40us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow15.in:   26475.80us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow16.in:   26189.60us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow18.in:   25935.70us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow20.in:   25605.20us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow21.in:   25376.20us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow22.in:   23683.00us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow24.in:   25848.90us (avg. of 10 runs; RSD: 0.00)


-----------------------------------------------------------------------
-----------------------------------------------------------------------
-- II. RADIX = 4

1. CPU SPLIT Runtimes
dataset data/input2pow6.in:        3.10us (avg. of 10 runs; RSD: 0.23)
dataset data/input2pow8.in:       11.80us (avg. of 10 runs; RSD: 0.05)
dataset data/input2pow10.in:     109.70us (avg. of 10 runs; RSD: 0.10)
dataset data/input2pow16.in:    8384.80us (avg. of 10 runs; RSD: 0.02)
dataset data/input2pow18.in:   37673.80us (avg. of 10 runs; RSD: 0.02)
dataset data/input2pow20.in:  181204.80us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow22.in:  791705.00us (avg. of 10 runs; RSD: 0.06)
dataset data/input2pow24.in: 4542222.00us (avg. of 10 runs; RSD: 0.00)

2. GPU SPLIT Runtimes
dataset data/input2pow6.in:      152.50us (avg. of 10 runs; RSD: 0.07)
dataset data/input2pow8.in:      159.90us (avg. of 10 runs; RSD: 0.06)
dataset data/input2pow10.in:     235.00us (avg. of 10 runs; RSD: 0.05)
dataset data/input2pow16.in:     361.60us (avg. of 10 runs; RSD: 0.04)
dataset data/input2pow18.in:     628.10us (avg. of 10 runs; RSD: 0.02)
dataset data/input2pow20.in:    1569.70us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow22.in:    5627.90us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow24.in:   22404.70us (avg. of 10 runs; RSD: 0.03)


3. CPU Runtimes
dataset data/input2pow6.in:  1296727.10us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow8.in:  1213851.40us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow10.in: 1200030.30us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow16.in: 1210837.70us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow18.in: 1433830.80us (avg. of 10 runs; RSD: 0.02)
dataset data/input2pow20.in: 1335269.10us (avg. of 10 runs; RSD: 0.03)
dataset data/input2pow22.in: 1259790.40us (avg. of 10 runs; RSD: 0.05)
dataset data/input2pow24.in: 1751502.20us (avg. of 10 runs; RSD: 0.00)


4. GPU RUNTIMES: GLOBAL MEMORY + TWIDDLE PRECOMPUTATION
dataset data/input2pow6.in:    23920.40us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow8.in:    18345.80us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow10.in:   16953.20us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow16.in:   15182.80us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow18.in:   15089.70us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow20.in:   14851.10us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow22.in:   13831.20us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow24.in:   15272.70us (avg. of 10 runs; RSD: 0.00)


5. GPU RUNTIMES: GLOBAL MEMORY, WITHOUT TWIDDLE PRECOMPUTATION
dataset data/input2pow6.in:    23383.00us (avg. of 10 runs; RSD: 0.02)
dataset data/input2pow8.in:    18219.20us (avg. of 10 runs; RSD: 0.02)
dataset data/input2pow10.in:   17082.10us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow16.in:   15300.30us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow18.in:   15036.40us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow20.in:   14649.30us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow22.in:   13483.60us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow24.in:   14579.40us (avg. of 10 runs; RSD: 0.00)
```

```
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-- III. RADIX 8

1. CPU SPLIT Runtimes
dataset data/input2pow6.in:       2.20us (avg. of 10 runs; RSD: 0.18)
dataset data/input2pow9.in:      13.70us (avg. of 10 runs; RSD: 0.03)
dataset data/input2pow15.in:   2879.10us (avg. of 10 runs; RSD: 0.03)
dataset data/input2pow18.in:  27023.80us (avg. of 10 runs; RSD: 0.02)
dataset data/input2pow21.in: 257328.70us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow24.in: 3679618.00us (avg. of 10 runs; RSD: 0.01)

2. GPU SPLIT Runtimes
dataset data/input2pow6.in:     132.40us (avg. of 10 runs; RSD: 0.06)
dataset data/input2pow9.in:     165.10us (avg. of 10 runs; RSD: 0.05)
dataset data/input2pow15.in:    253.40us (avg. of 10 runs; RSD: 0.03)
dataset data/input2pow18.in:    556.90us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow21.in:   2781.40us (avg. of 10 runs; RSD: 0.01)
dataset data/input2pow24.in:  21521.70us (avg. of 10 runs; RSD: 0.03)

3. CPU Runtimes
dataset data/input2pow6.in:  978961.10us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow9.in:  871656.50us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow15.in: 919394.80us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow18.in: 1338541.40us (avg. of 10 runs; RSD: 0.03)
dataset data/input2pow21.in: 1250893.10us (avg. of 10 runs; RSD: 0.11)
dataset data/input2pow24.in: 1683305.40us (avg. of 10 runs; RSD: 0.00)


4. GPU RUNTIMES: GLOBAL MEMORY + TWIDDLE PRECOMPUTATION
dataset data/input2pow6.in:   27282.40us (avg. of 10 runs; RSD: 0.02)
dataset data/input2pow9.in:   17112.50us (avg. of 10 runs; RSD: 0.02)
dataset data/input2pow15.in:  13330.10us (avg. of 10 runs; RSD: 0.02)
dataset data/input2pow18.in:  12757.30us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow21.in:  12135.10us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow24.in:  12084.10us (avg. of 10 runs; RSD: 0.00)


5. GPU RUNTIMES: GLOBAL MEMORY, WITHOUT TWIDDLE PRECOMPUTATION
dataset data/input2pow6.in:   29448.10us (avg. of 10 runs; RSD: 0.03)
dataset data/input2pow9.in:   17185.30us (avg. of 10 runs; RSD: 0.02)
dataset data/input2pow15.in:  13672.60us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow18.in:  12932.60us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow21.in:  12266.80us (avg. of 10 runs; RSD: 0.00)
dataset data/input2pow24.in:  12009.90us (avg. of 10 runs; RSD: 0.00)
```