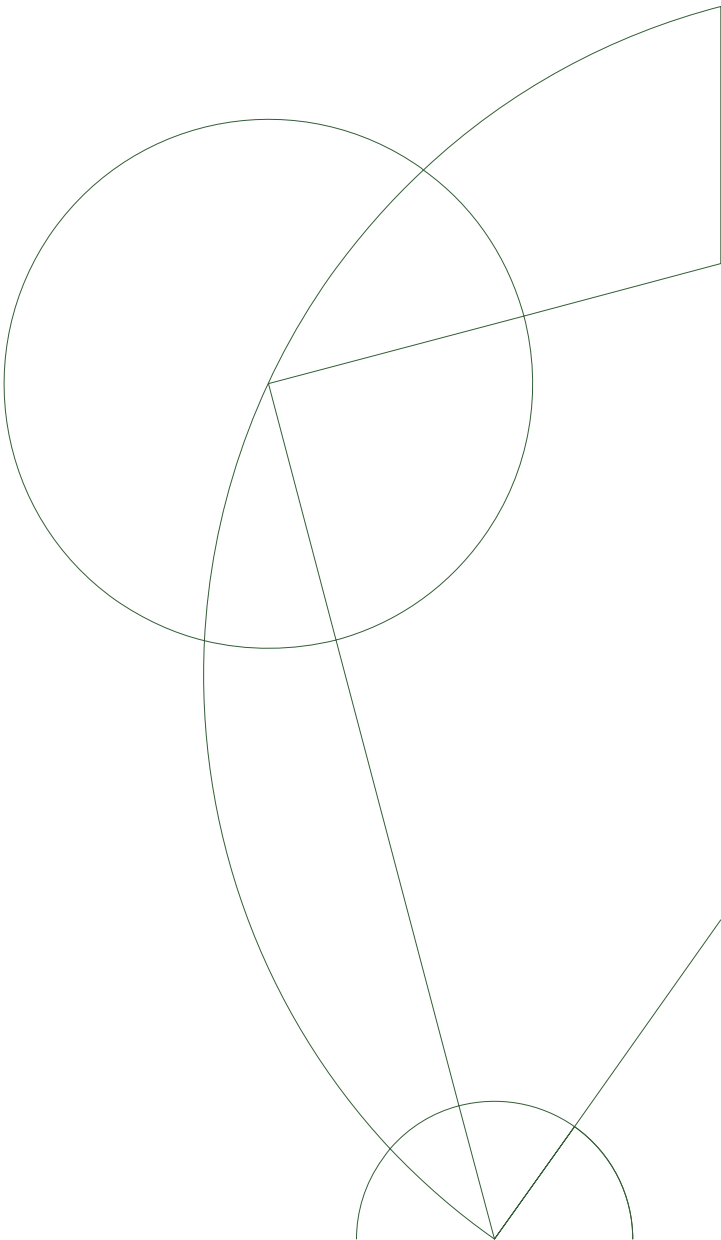
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Project in Computer Science

Kasper Abildtrup Hansen

FFT Generator in Furthark

A prototype Futhark library using FFTW technniques



Supervisor: Ken Friis Larsen

June 22, 2018

Contents

Abstract

In the end of 20th century Frigo and Johnson presented *The Fastest Fourier Transform in the West*, which has shown to be a highly successful approach for computing the discrete Fourier Transform. In this project FFTW techniques are employed to investigate the possibility of implementing a new FFT library in Futhark, `genfft`, which is then compared against the existing radix-2 FFT in Futhark. The results show that even though `genfft` is inefficient on datasets sizes involving prime factors, it is faster than the existing Futhark FFT on large datasets when the dataset sizes involve radices 2, 3 and 4.

1 Introduction

Because of its operational efficiency for computing the discrete Fourier Transform (DFT), the popularity of the fast Fourier Transform (FFT) as well as its capability for inspiring further research has been continuous since first presented by Cooley and Tukey in 1965. In the end of 20th century Frigo and Johnson presented a new approach for computing the DFT named *The Fastest Fourier Transform in the West* (FFTW), which builds on the idea of combining efficient FFTs into a composite algorithm to compute the DFT. The approach has proven to be highly successful and has been included and adopted in several software libraries including NVIDIAs cuFFT library [3].

Futhark[1] is a statically typed, data-parallel, and purely functional array language designed to be compiled to efficient parallel code, which is being developed at the Department of Computer Science, University of Copenhagen. Futhark currently only implements a simple parallel radix-2 FFT library, but an improved FFT library could help prove the computational strength of Futhark, which is why we in this project explore the possibility of improving the current FFT library in Futhark using FFTW techniques. The paper is divided into four parts:

- In the first section the scope of the project is introduced including limitations and restrictions.
- The second section introduces the primary techniques of FFTW and FFTs to be included, and in the third section certain certain aspects of the implementation is presented.
- The fourth section describes the benchmark testing, where the implementation is compared against the existing FFT library.
- Finally, the results are discussed and a conclusion is reached in sections 5 and 6.

1.1 Problem Statement

In this project we will investigate how techniques from FFTW can be used in a FFT library for Futhark by implementing and evaluating a prototype, `genfft`, using these techniques. The evaluation will be done by comparing run-times against the existing FFT library in Futhark, `/futlib/fft`. Because `/futlib/fft` is a relatively simple radix-2 FFT with only a few preparatory computations, we only expect to see speedups for `genfft` compared to `/futlib/fft` on large datasets, where the effect of FFTs with larger radices can be increased, and the slowdown caused by preparatory computations can be minimised.

1.2 Limitations and Restrictions

For simplicity, we have chosen to focus only on 1-D FFTs with complex input. In addition, since there is no asymptotic time difference between forward or inverse FFTs, and although all FFTs are capable of computing them both, we keep it simple by only evaluating the forward FFTs.

[1] https://futhark-lang.org/

2 FFTW Techniques in Futhark

In the following we make a short introduction to the *The Fastest Fourier Transform in the West* (FFTW) by Frigo and Johnson [5], and discuss some of the aspects of adopting FFTW techniques in Futhark. This is followed by a description of the specific FFTs we are using.

2.1 The Fastest Fourier Transform In The West

The FFTW is build on the overall idea of combining several efficient FFT algorithms into a composite algorithm, which is composed during run-time to specifically compute the DFT for some arbitrary N. To achieve this the FFTW incorporates three parts: (1) blocks of C code defining the FFT algorithms (named *codelets* by Frigo and Johnson), (2) a planner, which composes a plan p_i detailing how to combine the codelets based on a predefined strategy and the size of input i, and (3) a codelet generator, which given a plan p_i combines the codelets into a specific set of computations designed to efficiently compute the DFT of i [4, 5].

The codelet generator, `genfft`, is the main part of FFTW, generating the final code using four phases of operations: first, a *creation* phase where `genfft` produces the first version of the code based on some known DFT algorithm, (2) a *simplifier* phase, where it rewrites parts of the code based on algebraic transformations and common-subexpression elimination, (3) a *scheduler* phase, which minimises register spills, and finally (4) an *unparsing* phase, where the code is translated to C [4; p 2].

The advantages of adopting FFTW techniques in a FFT library using Futhark, is that Futhark simplifies certain parts of the process. Specifically, it implicitly handles the three final phases of the code generating process, leaving us to focus solely on implementing an efficient computational strategy and the necessary FFT algorithms. On the downside, there are certain aspects of the planning process, for example, the factorisation of N, which is foremost a iterative process, and therefore could possibly counteract the parallel capabilities of Futhark. Another aspect of this, is that there are no publicly known algorithms capable of integer factorisation in polynomial time, and since factorisation is an important part of choosing the right strategy, we need to employ an efficient method for doing this. In our case we may exploit the fact, that since we are implementing a specific set of FFT algorithms, we only need to granularise the input size to a point, were we are able to decide, which FFT algorithms to apply.

A recent addition to Futhark is the support for higher-order functions (HOFs), which we had hoped to employ in a FFTW-like library. But early testing showed that limitations enforced by Futhark regarding HOFs prohibited efficient usage. In the end, the show-stoppers were the limitations prohibiting the use of HOFs as the results of branching or as elements in an array, which could have proven to be useful in the current context.

2.2 FFT Algorithms

The discrete Fourier Transform (DFT) is defined as

$$F(y) = \sum_{x=0}^{N-1} f(x)\ e^{-i2\pi xy/N},$$

where $f(x)$ is a sequence of N elements and i is the imaginary unit satisfying the property $i^2 = -1$. Computing the DFT is very time consuming and requires $\mathcal{O}(N^2)$ operations, but

symmetries in the DFT recognised by Cooley and Tukey [2] allows for large reductions in the number of computations. This is represented by the fast Fourier Transform (FFT), which requires only $\mathcal{O}(N \log N)$ operations to compute the DFT. The basic idea of the Cooly-Tukey FFT is that the DFT of a composite $N = n_1 n_2$ can be expressed using smaller the DFTs of n_1 and n_2. Although, the Cooley-Tukey FFT is probably the most famous, there exist FFT algorithms based on other approaches, most notably Rader's FFT for prime sized N [12] and Bluestein's FFT for arbitrary N [1; pp 124–133], which are both based on expressing the DFT of N as the convolution of two subsequences.

Based on the fundamental theorem of arithmetic, which states that every integer $i > 1$ is either a prime number or can be expressed as a composite number of primes, we could simplify the planning process by only implementing either Rader's or Bluestein's FFT. Unfortunately, neither is very fast. A way of implementing the convolution needed in both algorithms are by exploiting the fact that the convolution $\mathcal{F}(a, b)$ of subsequences a and b can be expressed as

$$\mathcal{F}(a, b) = \text{inverse } DFT(DFT(a) \cdot DFT(b)),$$

where $X \cdot Y$ denotes the element-wise product of sequences X and Y [12]. The asymptotic number of operations for $\mathcal{F}(a, b)$ is $\mathcal{O}(N \log N)$ when implementing the convolution using efficient FFT algorithms, but the actual number of operations is larger than the Cooly-Tukey FFT, since the convolution involves both a forward and an inverse DFT. Because of this, we obviously need to employ several FFTs to achieve a reasonable efficiency. Since it first was introduced, the official FFTW library has expanded and version 3.3.8 incorporates several FFTs and features for optimising the library [7]. Since our goal is to implement a prototype version of FFTW in Futhark, we instead look at the earlier and more simple version 2.0 [4] as a basis regarding which FFTs to employ. In FFTW verson 2.0 included algorithms are split-radix algorithms, prime factor algorithms, Rader's FFT algorithm, as well as FFT algorithms for some smaller radices.

Radix-2, 3 and 4 FFTs

Govindaraju et al. presents several algorithms for computing the DFT on GPUs, one of which is a radix-2 FFT algorithm by Stockham. The Stockham FFT is based on the approach by Cooley-Tukey, but is more efficient and well-suited for parallel computation because it auto-sorts each stage of the FFT out-of-place [8]. It is also the algorithm used in the existing FFT library in Futhark, making it an obvious choice for including in a FFTW Futhark prototype.

Since the approach by Cooley-Tukey only requires N to be a composite, FFT algorithms for some radix $R \in \mathbb{N}$ can be composed in a way similar to the FFT of radix-2. There is, of course, a difference in how the smallest possible R-point DFT is expressed and the number of reducible computations, which is caused by a difference in the symmetry of the DFT of a radix-R sequence compared to a radix-2 sequence. Example of an optimised version of a radix-3 butterfly can be found in Löfgren et al.[11] and an example of a radix-4 butterfly can be seen in Wu [13].

Rader's FFT

To ensure that our algorithm is capable of computing all possible sizes, we can exploit the before-mentioned fundamental theorem of arithmetic and employ Rader's FFT algorithm [12] when N is a prime number. It runs in $\mathcal{O}(N \log N)$ like the Cooly-Tukey FFT, but the

actual number of operations are somewhat higher, since it involves two forward FFTs and one inverse FFT to compute the DFT, as well as permuting the sequences using transformations involving the primitive root of modulo.

Roughly, this means that if $N' = N - 1$ is highly composite, for example, $N' = 8 = 2^3$, the number of operations is more than three times higher than the Cooley-Tukey FFT. If N' is not highly composite, for example, $N' = 22 = 2^1 \cdot 11^1$, then the number of operations is even higher since it requires zero-padding the sequence into a highly composite length.

3 Implementation

The goal of this project is to investigate how FFTW techniques can be used in a Futhark library. To do this, a prototype `genfft` has been implemented consisting of a planner and an executor as described in Section 2.1. The implementation is evaluated by comparing it to the existing Futhark radix-2FFT library `/futlib/fft`. Since the quality of `genfft` relies mainly on the operational speed – not to mention the correctness of the computed DFTs – benchmark programs capable of measuring the singular parts of `genfft` have also been implemented.

This section presents the implementation and validation of the programs. All files are implemented using Futhark version 0.6.0 (Wed Jun 13 07:31:28 2018 +0200) and can be found in the project's GitHub repository [9] or in the included zip-file.

3.1 FFTs

In this version of `genfft` four FFT algorithms have been implemented: FFTs for radices 2, 3 and 4 and Rader's FFT algorithm for prime numbers.

Radix-2, 3 and 4 FFTs

The radix-2, 3 and 4 FFTs are implemented using the same module found in `generic_fft.fut` as a template, and similar to `/futlib/fft`, all three FFTs use the Stockham FFT for arbitrary radices presented by Govindaraju et al. [8; p. 3] as a reference implementation.

The main structure of the radix-2 FFT found in `r2fft.fut` is identical to `/futlib/fft`, except for the zero-padding functionality.

The radix-3 and radix-4 FFTs, which can be found in `r3fft.fut` and `r4fft.fut`, respectively, use the same structure as `r2fft.fut`, and only differs in how the respective FFT butterflies are implemented. In case of `r3fft.fut` the FFT butterfly has been rewritten slightly compared to the reference [11; Table II in p. 3] to save computations.

In addition, all three FFTs employ the Stockham FFT algorithm, both in case of auto-sorting the DFT, but also by including entries `fft_comp` for specifying the structure of the DFT to be computed. An example is shown in Listing 1, where the structure is defined by the values `factor` (equal to how much has been computed before this) and `times` (equal to how many times to apply this specific FFT). Together with specific radix these values are used in computing the `NS` sequence, which in the Stockham FFT dictates how to compute and arrange the input data.

```
1  let fft_comp [n] (forward: bool)
2                   (factor: i32)
3                   (times: i32)
4                   (data: [n](R.t, R.t)) : [n](R.t, R.t) =
5                   ...
```

Listing 1: Entry for specifying the DFT structure.

Rader's FFT

Rader's FFT algorithm is implemented in `rdfft.fut`, and like the three other FFTs, it employs the Stockham FFT algorithm for auto-sorting the DFT and for specifying the structure of the DFT to be computed. The current `rdfft.fut` employs the planner when

N' is highly composite (i.e. no factors of N' are prime) and zero-padding to $N'' = 2^M$ for some $N'' > 2N - 4$, when it is not.

Most of the implementation is a straightforward reference to the original article by Rader [12], although we try employ as much of the parallel capabilities of Futhark as possible. For example, by applying the forward FFT to the two subsequences using `map`.

An important part of Rader's algorithm is the use of number theory, and it requires operations like finding the primitive root of N and checking if a number is prime. All utilitary functionality needed for Rader's FFT can be found `primes.fut`, including an implementation of the Fast Powering Algorithm [10; pp. 25–26].

3.2 Planner

The planner can be found in `planner.fut` and takes of care of a two-step process: analyse the input size N, and create a plan based on the composability of N and a predefined strategy.

Strategy

With only four available FFTs, `planner.fut` starts by focusing on the fast FFTs when decomposing N. Preliminary testing has showed, that `r2fft.fut` is faster than `r4fft.fut` up to data sizes of $N = 2^{19}$, so we only include `r4fft.fut` for data sizes larger than this. Additionally, since we have not included the Bluestein FFT, we need to employ `rdfft.fut` in a composite FFT, which could potentially be very time consuming. The overall strategy implemented in `planner.fut` is:

1. If $N > 2^{19}$ then start by finding the larger possible value of r when dividing N with 4^r.
2. If $N \leq 2^{19}$ or step 1 os done, find the largest possible values of p and q when dividing N (or the remainder) with 2^p and when dividing the remainder with 3^q.
3. If the remainder is not a prime, find the remaining prime factors.

Create Plan

After decomposing N using the above strategy, we know which FFTs to use and how many times to apply them. Creating the plan is then merely a matter of computing the `factor` value used for defining the DFT structure (see Section 3.1), which is equal to

$$\texttt{factor}_i = \begin{cases} \text{If } i = 0 \text{ then } 1 \\ \text{If } i > 0 \text{ then } \texttt{factor}_{i-1} \cdot \texttt{radix}_{i-1}^{\texttt{times}_{i-1}}, \end{cases}$$

where $\texttt{radix}_i$ is the radix of the current FFT to apply, and $\texttt{times}_i$ is number of times to apply the current FFT. For example, if $N = 6 = 2^1 \cdot 3^1$, then $\texttt{factor}_0 = 1$ and $\texttt{factor}_1 = 1 \cdot 2^1 = 2$.

For each step in the plan, `planner.fut` defines these three values, `factor`, `radix` and `times`. For applying some minor type inference and for ease of reference, the values are reported using a domain specific type,

```
plan = {prefactor:  i32, radix:  i32, times:  i32},
```

which is used in the output of the main entry

```
make_plan -> i32 -> []plan.
```

The resulting array is used by the executor, but since it is unreadable in standard output, a function for translating the plan into a readable list of tuples has been implemented in

```
translate_plan -> []plan -> ([]i32, []i32, []i32).
```

3.3 Executor

The executor is found in the main file `genfft.fut` and combines the planner and FFTs into the prototype library. Besides being implemented as a parametric module, the structure is simple and includes the possibilities of making and executing a plan based on some floating point input. Because the computation of the composite DFT is inherently sequential the execution is implemented using a simple for-loop.

3.4 Additional Files

- For benchmarking the library a set of simple programs has been implemented. All benchmark programs use the prefix `bench_` and should be easy identifiable.
- For testing the correctness of the implemented FFTs a set of files are implemented, which test the linearity of 64-bit versions of the FFTs. All these files use the prefix `linearity_` and should be easy identifiable. Because certain FFTs differ in precision (see fx [8]) it is possible for the user to define a certain amount of buffer, when testing for linearity.
- Finally, a single file, `utilities.fut`, containing two utilitary functions used by some of the main files is implemented.

3.5 Validation Testing

All new files except the benchmark programs include test blocks with predefined inputs and expected outputs, and were tested for validity using `Futhark-test`.

In addition, to validating the programs using expected outputs, the four FFT programs were tested for correctness with a linearity test defined in corresponding linearity test files (see Section 3.4). Before testing for correctness the main linearity test file is validated to ensure a valid test. Three of the FFT programs, `r2fft.fut`, `r4fft.fut` and `rdfft.fut` had a 100 percent valid linearity test without any buffer, while `r3fft.fut` achieved a 100 percent valid linearity test, when not counting the last decimal.

All tests are implemented in the included Makefile.

4 Benchmark Tests

In the previous section implementation of the FFTW techniques were presented. In this chapter we present the conducted benchmark tests.

4.1 Methodology

Since this is a prototype version, it will not be fair to compare it against a fully developed parallel library like cuFFTW. Instead we use the existing `/futlib/fft` which is a based on a radix-2 Stockham FFT as a baseline. The current implementation uses zero-padding to adjust input sizes to radix-2, but to simplify the test we only benchmark it using radix-2 values. The `genfft`, on the other hand, will be benchmarked using various sizes to test the efficiency of the current implementation. In addition, the `planner` will be benchmarked independently of `genfft`, to get an idea of how much time it consumes.

All test programs are compiled to GPU-orientated executables using `futhark-opencl`, and run on GPU02 of the DIKU GPU Cluster[2] with the following hardware specifications:

- 1 Supermicro SYS-7047GR-TPRF, 4U/Tower barebone LGA2011, 2x1620W PSU, 8x3.5" htswp trays
- 2 Intel Xeon E5-2650v2, 8-core CPU, 2.6GHz, 20MB cache, 8GT/s QPI
- 8 Samsung 16GB DDR3(128GB total) 1866MHz Reg. ECC server module
- 2 nVidia GeForce GTX 780 Ti, 3072MB, 384 bit GDDR5, PCI-E 3.0 16x
- 1 Intel S3500 serie 240GB SATA SSD
- 1 Seagate Constellation ES.3 4TB 7200RPM SATA 6Gb/s 128MB cache 3,5" HDD

After running the tests, the run-times are collected and organised, and the mean, median and standard deviation values for each of the datasets are compared to validate the test results. Result of the tests are reported using average mean run-times in microseconds. All benchmark tests can be run using the included Makefile, and an overview of test scenarios can be found Table 1.

4.2 Expectations

As described in the problem statement in Section 1.1, the benchmark tests are only expected to show speedups for `genfft` compared to `/futlib/fft` on the largest datasets. On the smallest datasets, we expect the run-times of the `/futlib/fft` to be faster `genfft`. Overall, we expect to see the following when conducting the benchmark tests:

E1: The `genfft` to be fastest on the largest datasets. Because it includes FFTs with different radices, it should be capable of faster computation when the input sizes are larger, despite of a larger overhead.

E2: The `/futlib/fft` to be fastest on the smallest datasets. The existing Futhark FFT is simple and fast, and it is unlikely that the run-times on a prototype of a more comprehensive `genfft` can compete against it.

E3: Planning the FFT will slow down `genfft`. The idea behind the planner is, of course, to increase run-time. But when compared to a more simple radix-2 FFT, then the planner can potentially slow down computation.

[2] See `https://di.ku.dk/it/documentation/gpu/`

Test	Program	Input Struct	Input Type	Input sizes
`genfft_td1`	`genfft`	Radix 2	[N],[N]`f32`	$2^{13} \leq N \leq 2^{24}$
`genfft_td2`	`genfft`	Radix 2, 3 and 4 comb.	[N],[N]`f32`	$9216 \leq N \leq 17915904$
`genfft_td3`	`genfft`	Radix 2, 3, 4 and prime comb.	[N],[N]`f32`	$7680 \leq N \leq 16920576$
`genfft_td4`	`genfft`	Prime sized	[N],[N]`f32`	$8191 \leq N \leq 16777213$
`genfft_td5`	`genfft`	Comb. of two primes	[N],[N]`f32`	$8051 \leq N \leq 16777207$
`genfftmulti_td1`	`genfft`	Radix 2	[N],[N]`f32`	$2^{13} \leq N \leq 2^{24}$
`fft_td1`	`/futlib/fft`	Radix 2	[N],[N]`f32`	$2^{13} \leq N \leq 2^{24}$
`fftmulti_td1`	`/futlib/fft`	Radix 2	[N],[N]`f32`	$2^{13} \leq N \leq 2^{24}$
`gf2_td1`	`r2fft`	Radix 2	[N],[N]`f32`	$2^{13} \leq N \leq 2^{24}$
`gf3_td6`	`r3fft`	Radix 3	[N],[N]`f32`	$3^{8} \leq N \leq 3^{15}$
`gf4_td1`	`r4fft`	Radix 2	[N],[N]`f32`	$2^{13} \leq N \leq 2^{24}$
`gfRd_td6`	`rdfft`	Radix 3	[N],[N]`f32`	$3^{8} \leq N \leq 3^{15}$
`planner_td1`	`planner`	Radix 2	`i32`	$2^{13} \leq N \leq 2^{24}$
`planner_td2`	`planner`	Radix 2, 3 and 4 comb.	`i32`	$9216 \leq N \leq 17915904$
`planner_td3`	`planner`	Radix 2, 3, 4 and prime comb.	`i32`	$7680 \leq N \leq 16920576$
`planner_td4`	`planner`	Prime number	`i32`	$8191 \leq N \leq 16777213$
`planner_td5`	`planner`	Comb. of two primes	`i32`	$8051 \leq N \leq 16777207$
`planner_td6`	`planner`	Radix 3	`i32`	$3^{8} \leq N \leq 3^{15}$

Table 1: Benchmarking Test Scenarios

4.3 Results

All benchmarked run-times can be found in the `.time`-files in the package `benchmarks.zip`. Figures 1, 2, 3 and 4 show graphical representations of some of the benchmarks for `/futlib/fft` and `genfft` programs. Results for the `genfft_td4`, `genfft_td5` and `gfRd_td6` have not been included in the figures because of very slow run-times. In addition, `genfft_td4` and `gfRd_td6` were only capable of computing the DFT on three of the given datasets, and `genfft_td5` was only capable of computing the first nine datasets. The remaining datasets for these three programs were not computable because of lack of memory.

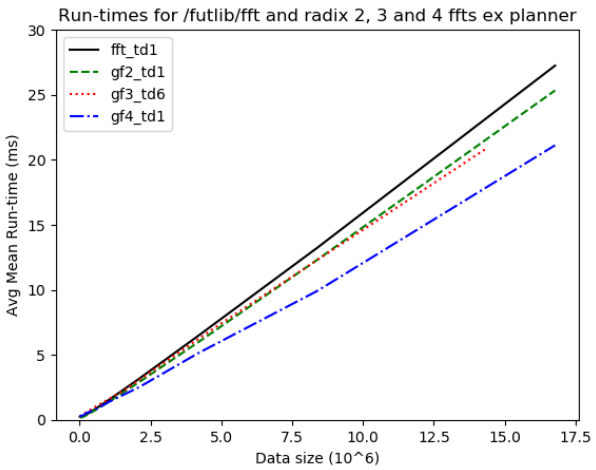


Figure 1: Benchmark results for FFT implementations. As singular programs the new FFT implementations all appear faster than the existing `/futlib/fft`, which may not be surprising since the existing FFT includes additional functionality compared to the new ones.

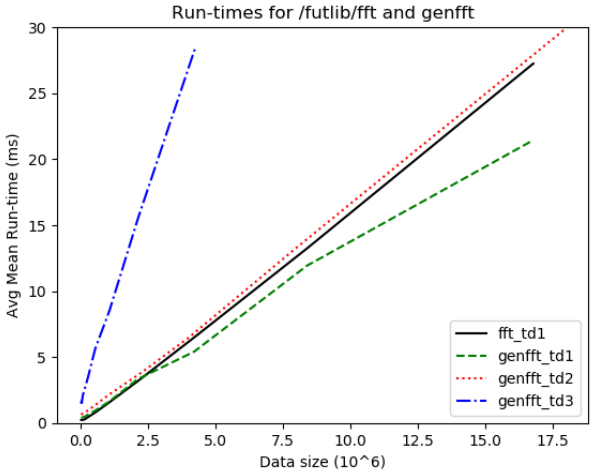


Figure 2: Benchmark results for `genfft`. Using fft_td1 as a baseline, `genfft` is only faster when applied to the same radix-2 sized datasets, `td1`. When applied to datasets that include prime factors the execution time is very slow.

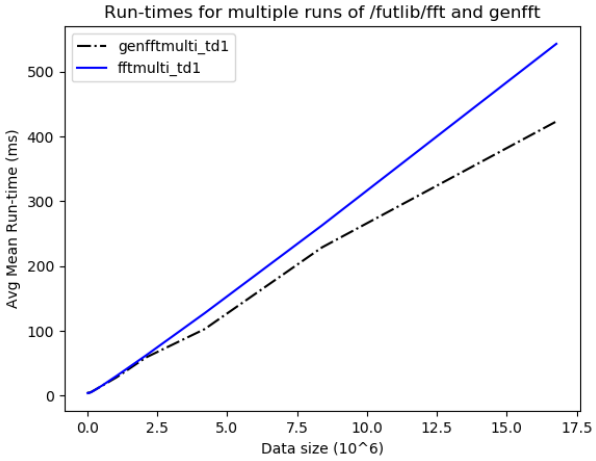


Figure 3: Benchmark results for multiple runs on same datasets. In according with the other results, it is not surprising to see that reusing plans allow an increased speedup of `genfft` compared to `/futlib/fft`.

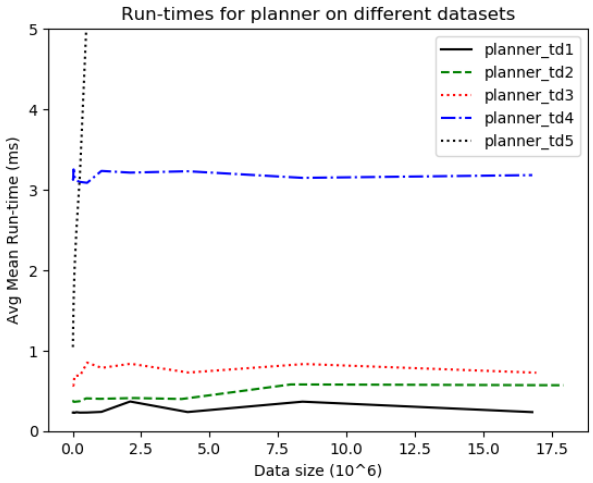


Figure 4: Benchmark results. Most noticeable is the amount of time spend on creating a plan, when N is the product of two primes (`planner_td5`). To decompose $N = 2099501$ into two primes, `planner.fut` spends around 13.6ms, which is more than `genfft` uses to compute the DFT of a complex input sized $N = 2^{23}$. Also, planning on prime sized input (`td3`) appears problematic. Perhaps less noticeable, is the fact that `planner.fut` spends at around the same amount of time creating a plan for input sized $N < 2^{17}$ as `/futlib/fft` uses to compute the DFT.

4.4 Data Validation

To validate run-times of the benchmark test, the mean, median and standard deviation σ are computed for each of the 18 benchmark tests. All computed values can be found in the package `benchmarks.zip`. Validation of test data are placed in the file `benchmarks_val.ods`.

Mean and Median

For convenience, Table 2 have been added, and contains a compilation of the mean and median values for the largest data sets in each test. In all tests the values show very small deviations when compared to the mean. The largest differences are found in the benchmarks for the `planner`, but all are the results of low run-times, most of them below one millisecond.

Test	**Dataset** X	**Mean** $\bar{X}$ μs	**Median** $\tilde{X}$ μs	$\tilde{X}/\bar{X}$	σ μs	$\sigma/\bar{X}$
`genfft_td1`	$N = 2^{24}$	21441	21398	0.998	176	0.008
`genfft_td2`	$N = 17915904$	29801	29784	0.999	67	0.002
`genfft_td3`	$N = 4230144$	28386	28200	0.993	372	0.013
`genfft_td4`	$N = 65537$	4266797	4267269	1.000	9623	0.002
`genfft_td5`	$N = 2099501$	516776	516154	0.999	1740	0.003
`genfftmulti_td1`	$N = 2^{24}$	423116	423121	1.000	96	0.000
`fft_td1`	$N = 2^{24}$	27245	27208	0.999	132	0.005
`fftmulti_td1`	$N = 2^{24}$	542961	542962	1.000	39	0.000
`gf2_td1`	$N = 2^{24}$	25333	25261	0.997	172	0.007
`gf3_td6`	$N = 3^{15}$	20863	20861	1.000	36	0.002
`gf4_td1`	$N = 2^{24}$	21117	21113	1.000	43	0.002
`gfRd_td6`	$N = 3^{11}$	4263581	4262477	1.000	9017	0.002
`planner_td1`	$N = 2^{24}$	235	233	0.991	12	0.051
`planner_td2`	$N = 17915904$	569	557	0.979	21	0.037
`planner_td3`	$N = 16920576$	725	714	0.985	25	0.035
`planner_td4`	$N = 16777213$	3181	3173	0.997	3173	0.016
`planner_td5`	$N = 16777207$	135505	135081	0.997	1174	0.009
`planner_td6`	$N = 3^{15}$	305	302	0.991	16	0.051

Table 2: Mean, Median and Standard Deviation

Standard Deviation

Table 2 also contains the standard deviation σ for each of the selected data sets. No σ values are larger than 4 percent compared to the mean for all selected data sets except in tests `planner_td1` and `planner_td6`, both caused by low run-times around 300 microseconds. Note, that although the shown σ values are fair representations of deviations in the tests, they are not necessarily the largest σ value of each of the tests. However, no σ values are large enough to cause concern, and no outliers are deviating enough to cause a major impact on the mean. The thirty results for each dataset ensures that outliers only have limited impact on the mean, and as a result, no outliers were removed from the final test results.

Conclusion

As shown, the mean, median and standard deviation show very little deviation in the overall test results. Based on this we argue that the results are a fair representation of each of the benchmark tests and therefore valid for comparing.

5 Discussion

The results of the benchmark tests proved some of our expectations (see Section 4.2) but not all.

The expectations regarding run-times on larger datasets (E1) were proven to some extent by `genfft` being faster than `/futlib/fft` on dataset sizes of $N > 3.98 \cdot 10^6$ involving the radix-2, 3 and 4 FFTs. When the datasets involved prime factors `genfft` showed to be slower than `/futlib/fft`, which goes against our expectations. In some cases where prime factors were involved the DFT was not even computable because of too high memory usage by `genfft`.

Looking at the results in Figure 2 also confirms the expectation that `/futlib/fft` would be fastest on the smaller sized datasets (E2). Although not visible in the Figure 2 the results show that `/futlib/fft` is faster than every benchmark result of `genfft` on dataset sizes of $N < 3.98 \cdot 10^6$. As mentioned earlier, this is not surprising since so much time is spend on planning the FFTs.

When looking at the results in Figure 1 and 2 it is evident, that the overhead in `genfft` spent on planning slows down the computation (E3). But since the time spent on planning the FFT for non-prime sized N is almost constant no matter how large, the impact of the overhead diminishes as the size of N increases.

All in all, the benchmark tests show certain aspects of the prototype `genfft`. The results in Figure 1 supports the use of the Stockham FFT algorithm as reference for the radix-2, 3 and 4 FFTs, although both the radix-3 and 4 FFTs could possibly be optimised for parallelisation even further. It is also evident that the current support for prime numbers is not sufficient. The results in Figure 4 suggest looking at the chosen strategy in `planner.fut` and the handling of prime numbers in `primes.fut`. On the smallest datasets in the test ($N < 10000$) the run-times t of `planner.fut` are in the range of $229\mu s \leq t \leq 3135\mu s$, and though it may prove difficult to improve the minimum, it should be possible to lower the maximum.

It is evident that the prototype of `planner.fut` is not efficient enough to compete with the run-times of `/futlib/fft`, and even though the new FFTs – not counting Rader – have shown to be fast, they still have a hard time outrunning `/futlib/fft`.

6 Conclusion

To investigate how techniques from FFTW can be used in a FFT library in Futhark, a prototype program, `genfft`, capable of making and executing a plan for computing the 1-D DFT of complex input of arbitrary size N was implemented. The program includes implementations of radix 2, 3 and 4 parallel FFTs, as well as an implementation of Rader's FFT algorithm for prime sized N. The correctness of all implemented FFTs were validated using expected outputs and tests of linearity.

The new `genfft` was expected to outperform the existing radix-2 FFT in Futhark, `/futlib/fft`, on larger datasets, which was the case except for datasets involving prime factors. We also expected `/futlib/fft` to outperform `genfft` on the smallest datasets, which proved to be the case. Finally, we expected that the overhead used on planning the FFT would slow down `genfft`, which proved to be the case.

Although certain aspects of the current `genfft` has proven insufficient, the results conclude that it is possible to employ FFTW techniques to implement an efficient library in Futhark. The current version implements Rader's FFT and is therefore theoretically capable of computing the DFT for all $N > 1$. But because it lacks efficiency when prime factors are involved, there are several obvious steps to take in order to improve the current version. First, it would be ideally to implement a radix-5 and possibly a radix-7 FFT to increase the number of composite N computable using Cooley-Tukey FFTs. In addition, the handling of primes needs to be improved and the strategy needs to be rethought to potentially reduce planning time without increasing execution time. This could involve only searching for a specific amount of prime numbers, which has shown to be very time consuming, and would require implementing, for example, Bluestein's FFT for arbitrary N. Another improvement could be to include predefined plans for certain sizes, which may help reduce overhead on highly composite sizes. This could eventually develop into a feature similar to FFTW's *wisdom files* [6], which contain saved information about how to optimally compute DFTs of various sizes.

References

[1] Eleanor Chu and Alan George. *Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms*. CRC Press, 1999.

[2] James Cooley and John Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[3] NVIDIA Corporation. cuFFT — CUDA Toolkit Documentation, 2018. Online: [`https://docs.nvidia.com/cuda/cufft/index.html`], accessed 17-June-2018.

[4] 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.

[5] Matteo Frigo and Steven G. Johnson. The fastest fourier transform in the west. Technical Report MIT/LCS/TR-728, Massachusetts Institute of Technology Laboratory for Computer Science, 1997.

[6] Matteo Frigo and Steven G. Johnson. FFTW WISDOM manual, 2003. Online: [`http://www.fftw.org/fftw-wisdom.1.html`], accessed 22-June-2018.

[7] Matteo Frigo and Steven G. Johnson. FFTW 3.3.8 — Documentation, 2018. Online: [`http://www.fftw.org/fftw3_doc/`], accessed 18-June-2018.

[8] 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.

[9] Kasper Abildtrup Hansen. Github repository: fut_genfft. Online: `https://github.com/diku-dk/fut_genfft`.

[10] Jeffrey Hoffstein, Jill Pipher, and J.H. Silverman. *An Introduction to Mathematical Cryptography*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[11] J. Löfgren and P. Nilsson. On hardware implementation of radix 3 and radix 5 fft kernels for lte systems. In *2011 NORCHIP*, pages 1–4, Nov 2011.

[12] C. M. Rader. Discrete fourier transforms when the number of data samples is prime. *Proceedings of the IEEE*, 56(6):1107–1108, June 1968.

[13] Charles Wu. Implementing the Radix-4 Decimation in Frequency (DIF) Fast Fourier Transform (FFT) Algorithm Using a TMS320C80 DSP. Technical report, January 1998. Online: `http://www.ti.com/lit/an/spra152/spra152.pdf`, accessed 22-June-2018.