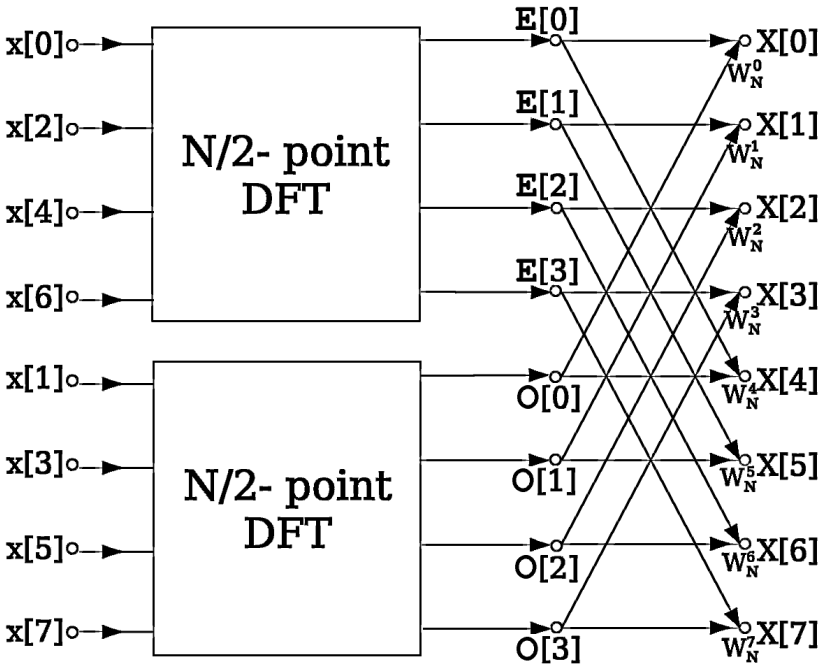
CHALMERS
UNIVERSITY OF TECHNOLOGY



Fast Fourier Transform using compiler auto-vectorization

Master's thesis in Complex Adaptive Systems

DUNDAR GÖC

Department of Mathematical Sciences
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2019

Master's thesis 2019

Fast Fourier Transform using compiler auto-vectorization

DUNDAR GÖC

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Mathematical Sciences
Chalmers University of Technology
Gothenburg, Sweden 2019

Fast Fourier Transform using compiler auto-vectorization
DUNDAR GÖC

Supervisor: Martin Raum, Department of Mathematical Sciences at Chalmers University of Technology
Examiner: Martin Raum, Department of Mathematical Sciences at Chalmers University of Technology

Master's Thesis 2019
Department of Mathematical Sciences
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: An illustration of the FFT algorithm structure.

Typeset in LaTeX
Digital Gothenburg, Sweden 2019

Fast Fourier Transform using compiler auto-vectorization
DUNDAR GÖC
Department of Mathematical Sciences
Chalmers University of Technology

Abstract

The purpose of this thesis is to develop a fast Fourier transformation algorithm written in C with the use of GCC (GNU Compiler Collection) auto-vectorization in the multiplicative group of integers modulo n, where n is a word sized odd integer. The specific Fourier transform algorithm employed is the cache-friendly version of the Truncated Fourier Transform. The algorithm was implemented by modifying an existing library for modular arithmetic written in C called zn_poly. The majority of the thesis work consisted of changing the code in a way to make integration into FLINT possible. FLINT is a versatile library, written in C, aimed at fast arithmetic of different kinds. The results show that auto-vectorization is possible with a potential speedup factor of 3. The performance increase is however entirely dependent on the task at hand and the nature of the computation being made.

Keywords: Fast Fourier Transform, Truncated Fourier Transform, Polynomial multiplication, Auto-vectorization.

Acknowledgements

I'd like to thank Martin Raum for his continued support, guidance and understanding throughout the entire thesis work. This thesis would not have been possible without him.
I'd also like to thank Erik Larsson and Felix Viberg for their opposition and for providing feedback.

Dundar Göc, Gothenburg, August 2019

Contents

List of Tables **xi**

1 Introduction **1**
1.1 Background . . . 1
1.2 Purpose . . . 1
1.3 Scope . . . 2
1.4 Thesis outline . . . 2
1.5 Code Repository . . . 2

2 Theory **5**
2.1 Polynomial multiplication using Fourier transform . . . 5
2.2 Fourier transform . . . 5
2.2.a Discrete Fourier Transform (DFT) . . . 5
2.2.b Fast Fourier Transform (FFT) . . . 6
2.2.b.1 FFT example . . . 7
2.2.c Truncated Fourier Transform (TFT) . . . 9
2.2.c.1 Reason for existence . . . 9
2.2.c.2 TFT . . . 9
2.2.c.3 ITFT . . . 10
2.2.d Number Theoretic Transform (NTT) . . . 11
2.2.e Primitive n–th roots of unity . . . 12
2.2.f Example . . . 12
2.3 Auto-vectorization . . . 14

3 Methods **15**
3.1 Materials . . . 15
3.1.a zn_poly . . . 15
3.1.b FLINT . . . 16
3.1.c Gantenbein . . . 16
3.2 Procedure . . . 16
3.2.a Modifying zn_poly . . . 17
3.2.a.1 Removing unrelated files . . . 17
3.2.a.2 Changing function names/documentation . . . 17
3.2.b Auto-vectorization . . . 18
3.2.b.1 Comparison of assembly output . . . 19

4 Results **21**
4.1 TFT 21
4.2 ITFT 21
4.3 Butterfly operation 22

5 Discussion **25**
5.1 General outcome 25
5.2 Limitations 26
5.3 Future work 26

Bibliography **27**

List of Tables

4.1 Execution time ratio of TFT with and without auto–vectorization. k denotes the number of bits in modulo and n denotes the maximum randomized polynomial length. 22
4.2 Average execution time ratio of TFT with and without auto–vectorization over k, the number of bits of modulo. 22
4.3 Average execution time ratio of TFT with and without auto–vectorization over n, the maximum randomized polynomial length. 23
4.4 Ratio of execution time of ITFT with and without auto–vectorization. k denotes the number of bits in modulo and n denotes the maximum randomized polynomial length. 23
4.5 Average execution time ratio of ITFT with and without auto–vectorization over k, the number of bits of modulo. 23
4.6 Average execution time ratio of ITFT with and without auto–vectorization over n, the maximum randomized polynomial length. 23
4.7 Ratio of execution time of butterfly with and without auto–vectorization. k denotes the number of bits in modulo and n denotes the maximum randomized polynomial length. 24
4.8 Average execution time ratio of butterfly with and without auto–vectorization over k, the number of bits of modulo. 24
4.9 Average execution time ratio of butterfly with and without auto–vectorization over n, the maximum randomized polynomial length. . . 24

1

Introduction

1.1 Background

The Fast Fourier Transform is an important algorithm that is present in many areas of modern life. It was popularized in 1965 and has been heavily used in science, technology and scientific computing ever since [1]. It is therefore an important algorithm to study and improve. It has been extensively developed due to this and there are today countless variations of FFT.

An important development was made in 2004 by Joris van der Hoeven [2] [3]. He developed what he called the Truncated Fourier Transform (TFT). The TFT is a truncated version of the FFT which eliminates the computational "jump" for each sequence length of a power of two. This makes the computational complexity to increase smoothly with increasing sequence length.

A second important development to the TFT, and by extension the FFT, was done in 2008 by David Harvey [4]. He modified the TFT developed by Hoeven to be better suited for large sequences by improving the locality of the transformations. This modification speeds up the algorithm by allowing an entire sequence take advantage of the CPU cache.

1.2 Purpose

The thesis consists of developing a fast computer implementation of FFT as well as researching how compiler auto-vectorization can be used to achieve this. The implementation is to be written in ANSI C and used in conjunction with the Fast Library For Number Theory (FLINT). The purposes of this thesis may be summarized as following:

- Develop a computer program in C in which performs FFT for integers modulo an odd number up to a word size with focus on speed and performance.

- Research if and how compiler auto-vectorization can be used in program and what current limitations exist.
- Designing the program in a way to make a full integration into FLINT. This means the program has to be written in ANSI C and that it must be architecture and compiler independent. The program should also implement FLINT:s functions whenever possible and avoid creating duplicate functions.

1.3 Scope

This thesis will only research FFT in integers modulo n, where n is an odd, word-sized integer. The reason for this is simply because that was the specific request given by the person proposing the thesis. The program is written purely in C, since it has to be designed in such a way that integrating it into FLINT should be possible. The auto-vectorization will only be researched with the GCC compiler. The reason for this choice is that GCC is currently the most popular C compiler and that further insights would benefit a large number of people.

1.4 Thesis outline

The theory of FFT, TFT and auto-vectorization is covered in chapter 2. This is to make sure the reader has the theoretical foundation needed to understand the thesis.

The used methods and chosen solutions to tackle the problems are explained and motivated in chapter 3.

The results are presented in chapter 4 by showing the comparison of performance between the vectorized and non-vectorized TFT.

A full analysis and discussion of the results is provided in chapter 5. The discussion will provide further insights not only into the results themselves but also the reasons behind the results.

1.5 Code Repository

The source code is available in a GitHub repository with the URL https://github.com/DundarGoc/Schonhage-Strassen-algorithm. The benchmarks done in this thesis are available on the branch named ”fourier-transform”. The

main file for benchmarking is benchmarkVectorization.c file. The other files include the algorithms that were benchmarked. The benchmarking was done with GCC version 8.3, FLINT version 2.5.2 and GMP version 6.1.2.

2

Theory

2.1 Polynomial multiplication using Fourier transform

Polynomial multiplication is not the primary focus of this thesis. It is however the primary reason for why achieving fast FFT in integers modulo n is desirable. An explanation of how polynomial multiplication can be achieved by using FFT will be provided in this section.

A polynomial length of n will be defined as a polynomial of degree $n-1$ (since n coefficients are needed to represent it). Suppose there are two polynomials g and h we wish to multiply and that the result is $u = gh$. If the lengths of g and h are z_g and z_h then the resulting polynomial will have the length $n = z_g + z_h - 1$.

Polynomial g and h are first individually transformed i.e. $G = \mathbf{DFT}(g)$ and $H = \mathbf{DFT}(h)$, both with length n. The pointwise product is then $U_i = G_i H_i$ for $0 \leq i < n$, where x_i is the ith coefficient of polynomial. The polynomial U is then transformed back i.e. $u = \mathrm{IDFT}(U)$. [5]

2.2 Fourier transform

2.2.a Discrete Fourier Transform (DFT)

Recall that, given a sequence of complex numbers $x_0, \ldots, x_{N-1}$ of length N, the DFT yields a sequence $X_0, \ldots, X_{N-1}$ defined by

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn}, \quad \text{for} \quad n, k = 0, 1, \ldots, N-1 \text{ [6].} \tag{2.1}$$

Observe that $\omega = e^{-\frac{2\pi i}{n}}$ is a primitive N–th root of unity. We can rephrase (2.1) using matrix notation. Writing $\mathbf{x} = \begin{pmatrix} x_0 & \cdots & x_{N-1} \end{pmatrix}^{\mathrm{T}}$ and $\mathbf{X} = \begin{pmatrix} X_0 & \cdots & X_{N-1} \end{pmatrix}^{\mathrm{T}}$, we have

$$\mathbf{X} = \begin{pmatrix} \omega^{0\cdot 0} & \omega^{0\cdot 1} & \dots & \omega^{0\cdot(N-1)} \\ \omega^{1\cdot 0} & \omega^{1\cdot 1} & \dots & \omega^{1\cdot(N-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \omega^{(N-1)\cdot 0} & \omega^{(N-1)\cdot 1} & \dots & \omega^{(N-1)\cdot(N-1)} \end{pmatrix} \mathbf{x}. \tag{2.2}$$

We write $\mathbf{F}_N$ for the transformation matrix in (2.2), commonly referred to as the DFT-matrix. The time–complexity of a naive matrix multiplication of two matrices of size $n \times m$ and $m \times p$ is $\mathcal{O}(nmp)$, which gives the naive multiplication a time complexity of $\mathcal{O}(N \cdot N \cdot 1) = \mathcal{O}(N^2)$. This causes the computation time to increase quickly as the sequence length N increases. This complication is the rationale for the FFT algorithm, which utilizes the specific structure of $\mathbf{F}_N$ as a Vandermonde matrix associated with power of ω.

The IDFT may be computed in the same way as DFT apart from an additional constant [7] i.e.

$$\mathbf{x} = \frac{1}{N}\mathbf{F}_{\mathrm{N}}\mathbf{X}, \tag{2.3}$$

where each variable is identical to those defined in equation (2.2). This is the reason why it's a common practice to only implement an algorithm for the discrete Fourier transform and reuse the same algorithm to compute the inverse.

2.2.b Fast Fourier Transform (FFT)

The most common FFT algorithm is the radix-2 decimation-in-time FFT where the input sequence is divided into two separate, evenly sized sequences. The DFT of these two sequences are computed separately and the results are then combined to compute the DFT of the main sequence. The same principle can then be applied to each divided sequence in a recursive manner until only two elements are left in a sequence. This reduces the time complexity to $\mathcal{O}(N\log N)$, where N is the sequence length. This requires the input sequence length to be a power of two. If the sequence length isn't a power of two then the end of the sequence is padded with zeroes until it is.

In each recursion the input sequence length $n = 2^l$ can be thought of as a $L_1 \times L_2$ matrix, where $L_1 = 2^{\lfloor \frac{l}{2} \rfloor}$ and $L_2 = 2^{\lceil \frac{l}{2} \rceil}$ and sorted in row-major order. The

function is then recursively called with each column as the input sequence. The same procedure is then applied to all rows as the input sequence. This concludes the algorithm. Pseudocode of this is shown in algorithm 1. The n–th roots of unity is denoted as ω_n, the ith column of the matrix as c_i and the ith row as r_i, starting from 0. The function is initially called as RecursiveFFT$(n, 1, \mathbf{x})$.

Algorithm 1 RecursiveFFT$(n, \zeta, \mathbf{x})$

Base case
1: **if** $n = 2$ **then**
2: $(x_0, x_1) \leftarrow (x_0 + x_1, \zeta\,(x_0 - x_1))$
3: **return**

Recursive case
4: $l \leftarrow log_2\,(n)$
5: $L_1 \leftarrow 2^{\lfloor l/2 \rfloor}$, $L_2 \leftarrow 2^{\lceil l/2 \rceil}$

Column transforms
6: **for** $0 \leq u < L_2$ **do** RecursiveFFT$(L_1, \omega_n^u \zeta, c_u)$

Row transforms
7: **for** $0 \leq u < L_1$ **do** RecursiveFFT(L_2, ζ^{L_1}, r_u)

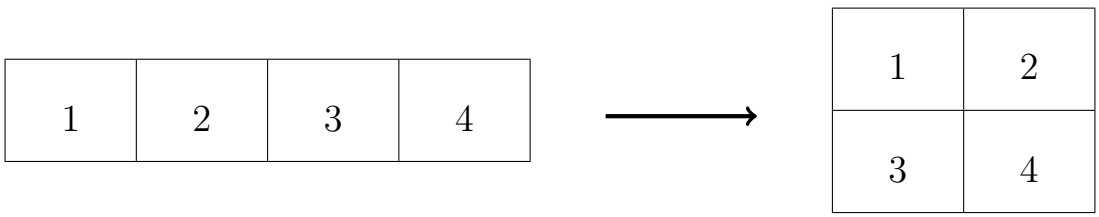
2.2.b.1 FFT example

Suppose we want to transform the sequence $\{1, 2, 3, 4\}$. The function is initially called as RecursiveFFT$(n, 1, \mathbf{x})$, where $n = 4$ and $\mathbf{x} = \begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix}^{\mathrm{T}}$.

We skip the base case since $n \neq 2$. The recursive case in steps 4 and 5 gives

$$l \leftarrow log_2\,(n) = log_2\,(4) = 2, \tag{2.4}$$
$$L_1 \leftarrow 2^{\lfloor l/2 \rfloor} = 2^{\lfloor 2/2 \rfloor} = 2, \tag{2.5}$$
$$L_2 \leftarrow 2^{\lceil l/2 \rceil} = 2^{\lceil 2/2 \rceil} = 2. \tag{2.6}$$

We may now think of the sequence as a row-major order matrix:



We now iterate through the columns of the matrix and recursively call the function for each column as in step 6 of the algorithm.

The primitive 4–th root of unity is $\omega_4 = exp\left(-\frac{2\pi i}{4}\right) = -i$. The reasoning behind this is explained in section 2.2.e and will be omitted in this section for brevity.

We set $u = 0$ and invoke the function as RecursiveFFT$(L_1, \omega_4^0\zeta, c_0)$ = RecursiveFFT$\left(2, 1, \left(\begin{smallmatrix}1 & 3\end{smallmatrix}\right)^{\mathrm{T}}\right)$. Since $n = 2$ in the recursive call then we only need to consider the base case of the algorithm. Our new input sequence becomes $\mathbf{x} = \left(\begin{smallmatrix}x_0 & x_1\end{smallmatrix}\right)^{\mathrm{T}} = \left(\begin{smallmatrix}1 & 3\end{smallmatrix}\right)^{\mathrm{T}}$. Performing the calculation on step 2 gives

$$(x_0, x_1) \leftarrow (1 + 3, 1(1 - 3)) = (4, -2)\,. \tag{2.7}$$

We then return and replace the new values for x_0 and x_1 in the matrix.

1	2
3	4

$\longrightarrow$

4	2
-2	4

We repeat the same procedure but with $u = 1$, which corresponds to the second column. This leaves us with

4	2
-2	4

$\longrightarrow$

4	6
-2	2 i

The only part left now is step 7, the row transformations. The principle is the same as with the column transformations, the only difference are the input values. We set $u = 0$ and call the function as RecursiveFFT$\left(L_2, \zeta^{L_1}, r_0\right)$ = RecursiveFFT$\left(2, 1, \left(\begin{smallmatrix}4 & 6\end{smallmatrix}\right)^{\mathrm{T}}\right)$.

4	6
-2	2 i

$\longrightarrow$

10	-2
-2	2 i

We repeat the same procedure for the second row, which results in

10	-2
-2	2 i

10	-2
-2 + 2 i	-2 -2 i

We now have the FFT of the initial sequence sorted in column major order, i.e. the output sequence is $\{10, -2 + 2i, -2, -2 - 2i\}$. This concludes the example.

2.2.c Truncated Fourier Transform (TFT)

2.2.c.1 Reason for existence

The drawback of the standard radix-2 FFT is as previously mentioned that it needs to have an input sequence length that is a power of two. This causes a jump in computation time each time the input sequence length surpasses a power of two. TFT is designed to counteract this flaw by presenting an algorithm where the execution speed is proportional to the input size without massive jumps in computations. An important clarification to make is that TFT is fundamentally different from FFT in that it gives different output. The results are only correct if TFT is used in conjunction with ITFT. This means that using the conventional inverse fast Fourier transform (IFFT) after using TFT will give wrong results. The reasons and mechanisms for this is explained in this section.

2.2.c.2 TFT

The TFT may be thought of as an extension of the recursive FFT algorithm described in algorithm 1. The difference is that only the relevant calculations are made. TFT is identical to the aforementioned FFT for input sequences of length a power of two. The algorithm is best visualized as with FFT, i.e. by imagining the input sequence as a matrix. Assume there is an input sequence of length n, which isn't necessarily a power of two. This sequence is padded with zeros until its length is a power of two. We'll call this length $L = 2^l$. The sequence is then thought to be transformed into a $L_1 \times L_2$ matrix, where $L_1 = 2^{\lfloor \frac{l}{2} \rfloor}$ and $L_2 = 2^{\lceil \frac{l}{2} \rceil}$ and sorted in row-major order as with the FFT. A pseudocode is shown in algorithm 2 [4].

Algorithm 2 TFT$(L, \zeta, z, n, \mathbf{x})$

$L = 2^l \geq 2$, $1 \leq z < L$, $1 \leq n < L$
$x_i = a_i$ for $0 \leq i < z$

Base case
1: **if** $L = 2$ **then**
2: **if** $n = 2$ and $z = 2$ **then** $(x_0, x_1) \leftarrow (x_0 + x_1, \zeta\,(x_0 - x_1))$
3: **if** $n = 2$ and $z = 1$ **then** $x_1 \leftarrow \zeta x_0$
4: **if** $n = 1$ and $z = 2$ **then** $x_0 \leftarrow x_0 + x_1$
5: **return**

Recursive case
6: $l \leftarrow \lceil log_2\,(n) \rceil$
7: $L_1 \leftarrow 2^{\lfloor l/2 \rfloor}$, $L_2 \leftarrow 2^{\lceil l/2 \rceil}$
8: $n_2 \leftarrow n \bmod L_2$, $n_1 \leftarrow \lfloor n/L_2 \rfloor$, $n_1' \leftarrow \lceil n/L_2 \rceil$
9: $z_2 \leftarrow z \bmod L_2$, $z_1 \leftarrow \lfloor z/L_2 \rfloor$
10: **if** $z_1 > 0$ **then** $z_2' \leftarrow L_2$ **else** $z_2' \leftarrow z_2$

Column transforms
11: **for** $0 \leq u < z_2$ **do** TFT$(L_1, \omega_L^u \zeta, z_1 + 1, n_1', c_u)$
12: **for** $z_2 \leq u < z_2'$ **do** TFT$(L_1, \omega_L^u \zeta, z_1, n_1', c_u)$

Row transforms
13: **for** $0 \leq u < n_1$ **do** TFT$(L_2, \zeta^{L_1}, z_2', L_2, r_u)$
14: **if** $n_2 > 0$ **then** TFT$(L_2, \zeta^{L_1}, z_2', n_2, r_{n_1})$

2.2.c.3 ITFT

The ITFT is similar to TFT but a few modifications are required. These modifications are needed to build up the necessary information that is missing during the TFT by skipping the calculations done in FFT [8]. This procedure may be thought of as "going back in time", in which the correct input for the ITFT is built up reverse-engineering the calculations done in TFT for a specific chosen coefficients [9]. This is done with what may be referred to as a reverse butterfly. A pseudocode is shown in algorithm 3 [4].

Algorithm 3 ITFT$(L, \zeta, z, n, f, \mathbf{x})$

$L = 2^l \geq 2$, $f \in \{0, 1\}$, $1 \leq z < L$
$1 \leq n + f < L$, $x_i = \hat{a}_i$ for $0 \leq i < n$

Base case
if $L = 2$ **then**
  if $n = 2$ **then** $(x_0, x_1) \leftarrow (x_0 + \zeta^{-1}x_1, x_0 - \zeta^{-1}x_1)$
  if $n = 1$ and $f = 1$ and $z = 2$ **then** $(x_0, x_1) \leftarrow (2x_0 - x_1, \zeta(x_0 - x_1))$
  if $n = 1$ and $f = 1$ and $z = 1$ **then** $(x_0, x_1) \leftarrow (2x_0, \zeta x_0)$
  if $n = 1$ and $f = 0$ and $z = 2$ **then** $x_0 \leftarrow 2x_0 - x_1$
  if $n = 1$ and $f = 0$ and $z = 1$ **then** $x_0 \leftarrow 2x_0$
  if $n = 0$ and $z = 2$ **then** $x_0 \leftarrow (x_0 + x_1)/2$
  if $n = 0$ and $z = 1$ **then** $x_0 \leftarrow x_0/2$
  return

Recursive case
$l \leftarrow \lceil log_2(n) \rceil$
$L_1 \leftarrow 2^{\lfloor l/2 \rfloor}$, $L_2 \leftarrow 2^{\lceil l/2 \rceil}$
$n_2 \leftarrow n \bmod L_2$, $n_1 \leftarrow \lfloor n/L_2 \rfloor$
$z_2 \leftarrow z \bmod L_2$, $z_1 \leftarrow \lfloor z/L_2 \rfloor$
if $n_2 + f > 0$ **then** $f' \leftarrow 1$ **else** $f' \leftarrow 0$
if $z_1 > 0$ **then** $z_2' \leftarrow L_2$ **else** $z_2' \leftarrow z_2$
$m \leftarrow \min(n_2, z_2)$, $m' \leftarrow \max(n_2, z_2)$

Row transforms
for $0 \leq u < n_1$ **do** ITFT$(L_2, \zeta^{L_1}, L_2, L_2, 0, r_u)$

Rightmost column transforms
for $n_2 \leq u < m'$ **do** ITFT$(L_1, \omega_L^u\zeta, z_1 + 1, n_1, f', c_u)$
for $m' \leq u < z_2'$ **do** ITFT$(L_1, \omega_L^u\zeta, z_1, n_1, f', c_u)$

Last row transform
if $f' = 1$ **then** ITFT$(L_2, \zeta^{L_1}, z_2', n_2, f, r_{n_1})$

Leftmost column transforms
for $0 \leq u < m$ **do** ITFT$(L_1, \omega_L^u\zeta, z_1 + 1, n_1 + 1, 0, c_u)$
for $m \leq u < n_2$ **do** ITFT$(L_1, \omega_L^u\zeta, z_1, n_1 + 1, 0, c_u)$

2.2.d Number Theoretic Transform (NTT)

FFT may also be generalized to work on the multiplicative group of integers modulo n. This is most commonly referred to as Number Theoretic Transform. NTT is

especially useful for when integer multiplication or polynomial multiplication with positive integer coefficients. It is possible to use FFT over the complex numbers to calculate integer multiplication. This method is however undesirable as it introduces round-off errors. Another advantage of using NTT over FFT is the flexibility of parameter choice, meaning it's possible to choose parameters such that it becomes possible to replace many multiplications with bit shifts and additions [5].

The practical difference is that the roots of unity are in $\mathbb{Z}/n\mathbb{Z}$ instead of $\mathbb{C}$. This change in field causes all arithmetic in the algorithm to be modular. An operation between a and b changes from $a \circ b$ to $(a \circ b) \bmod n$, where $\circ$ is an arithmetic operator such as addition, subtraction, multiplication or division. The biggest difference however is the change in the primitive n–th roots of unity and how it is calculated.

2.2.e Primitive n–th roots of unity

The n–th roots of unity in a complex field are the complex numbers that are the solutions to the equation $x^n = 1$, where n is a positive integer. The n–th roots of unity are simply $exp(-2\pi ki/n)$ for $k = 0, 1, \ldots, n-1$.

A primitive n–th root of unity is a root of unity that satisfies the relation $x^k \neq 1$, $k < n$. In a complex field this equates the roots of unity where k and n are coprime, meaning their greatest common denominator is 1. Since 1 is coprime to all integers then a possible primitive n–th root of unity regardless of n is $exp(-2\pi i/n)$. This procedure makes choosing a primitive n-th root of unity a straightforward task when performing FFT in complex fields.

The concept of roots of unity is not limited to complex fields and can be defined in any field. The type of field that is relevant is the ring of integers modulo m. The difficulty of finding a primitive n–th root of unity modulo m is a significantly harder task.

A n–th root of unity modulo m, where m is a positive integer, is an integer that satisfies the equation $x^n \equiv 1 \pmod{m}$. In a similar manner, a primitive n–th root of unity modulo m is a n–th root of unity modulo m that also satisfies $x^k \not\equiv 1 \pmod{\text{m}}$ for $k < n$.

2.2.f Example

Suppose we're searching for the primitive 6–th root of unity modulo 9. This corresponds to $n = 6$ and $m = 9$ in the previous section. A simple, although not necessarily the most efficient, method is to simply iterate over the values $2, 3, \ldots, p-1$ one by one and test if it satisfies the mentioned constraints. The

first step is to check if it is a root of unity, and if so test if it is a primitive roots of unity. If the number isn't a root of unity then there's no need to check if it's a primitive one since it's a necessary condition.

Let's test if 3 is a primitive 6–th root of unity modulo 9. We first need to test if it's a regular root by computing

$$3^6 = 729 \equiv 0 \pmod 9. \tag{2.8}$$

Since the computation didn't yield a 1 we know 3 isn't a root of unity and can stop testing it.

Next let's test 4:

$$4^6 = 4069 \equiv 1 \pmod 9. \tag{2.9}$$

This shows that 4 is a root of unity since the result is equal to one. Next we need to determine if it's a primitive root of unity:

$$4^1 = 4 \equiv 4 \pmod 9 \tag{2.10}$$
$$4^2 = 16 \equiv 7 \pmod 9 \tag{2.11}$$
$$4^3 = 64 \equiv 1 \pmod 9 \tag{2.12}$$
$$4^4 = 256 \equiv 4 \pmod 9 \tag{2.13}$$
$$4^5 = 1024 \equiv 7 \pmod 9. \tag{2.14}$$

4 is not a primitive 6–th root of unity modulo 9 since $4^3 \equiv 1 \pmod 9$ which breaks the second constraint that no power less than 6 should result in 1.

Lastly, let's test the number 2:

$$2^6 = 64 \equiv 1 \pmod 9. \tag{2.15}$$

The test confirms that 2 is a root of unity. We then test if it's a primitive one:

$$2^1 = 2 \equiv 2 \pmod 9 \tag{2.16}$$
$$2^2 = 4 \equiv 4 \pmod 9 \tag{2.17}$$
$$2^3 = 8 \equiv 8 \pmod 9 \tag{2.18}$$
$$2^4 = 16 \equiv 7 \pmod 9 \tag{2.19}$$
$$2^5 = 32 \equiv 5 \pmod 9. \tag{2.20}$$

Since none of the answer equal 1 then we know for sure that 2 is a primitive 6–th root of unity modulo 9. This concludes the example.

2.3 Auto-vectorization

Parallel computing and vectorization plays a central role in high performance computing. There are different types of parallelism but the one that is relevant for this thesis is the Single Instruction, Multiple Data (SIMD) class. A SIMD instruction performs the same operation on multiple data points at the same time [10].

It is possible to manually vectorize a loop or an instruction with the help of intrinsic functions or intrinsics. An intrinsic function is a built-in function [11] defined by the compiler itself which allows for increased performance due to compiler specific optimizations. The downside of manually implementing intrinsics is that it is entirely architecture dependent. Different computer architectures supports different SIMD instruction sets. In other words, using intrinsics to increase the performance is only possible if that specific hardware supports it. Multiple implementations of the same function has to be defined in order to guarantee that intrinsics are used in all computer architectures. This makes intrinsics difficult, error-prone and time-consuming [12] to implement and requires in-depth knowledge of compilers, computer architecture and SIMD instruction sets.

One possible solution to circumvent this problem is to use automatic vectorization, or auto-vectorization. Many modern C compilers have built-in auto-vectorization that can be taken advantage of if the program is compiled with the compiler flag ftree-vectorize. The compiler will then attempt to vectorize loops if possible by using the available intrinsics the hardware supports. The main benefit of this, aside from the obvious performance gains, is that the user is able to take advantage of the performance gains of using intrinsics without needing learn complicated intrinsic functions which differ wildly from each other in both function and form depending on architecture. The user can instead focus on designing a program code without architecture dependence in mind.

3

Methods

3.1 Materials

The two main software programs used are the two C libraries zn_poly and FLINT. Other software used were mainly dependencies for these libraries to function properly. This section will aim to give a thorough description the two libraries and their dependencies.

3.1.a zn_poly

zn_poly is a C library developed by David Harvey, Associate Professor at University of New South Wales. He himself describes it as ..."a C library for polynomial arithmetic in $\mathcal{Z}/n\mathcal{Z}[x]$, where n is any modulus that fits into an unsigned long." The main dependency of this library is the "The GNU Multiple Precision Arithmetic Library" (GMP), a C library for arbitrary precision arithmetic designed to be as fast as possible.

The developed TFT program in this thesis is a rework of the zn_poly library. Although the employed algorithms are fundamentally unchanged the entire library had to essentially rewritten in order to make full integration into FLINT possible. More on this procedure is described in section 3.2.a.

The are a few reasons why zn_poly was chosen to be main inspiration for the developed code for this thesis. The first and foremost reason was that it's a developed library for arithmetic in $\mathcal{Z}/n\mathcal{Z}[x]$, including fast TFT, which is precisely what is sought after in this thesis. It's also inspired parts of both the NTL and FLINT library, both of which are widely used arithmetic libraries in C and C++ which adds further credibility to the quality of the code.

3.1.b FLINT

This thesis is aimed at developing a fast TFT algorithm for FLINT. The most relevant sections of FLINT is the nmod_vec and nmod_poly sections, which are the sections of the library used to make arithmetic operations of vectors and polynomials over $\mathcal{Z}/n\mathcal{Z}$ where n is a word-sized modulo meaning that it fits in an unsigned long as with the zn_poly library.

FLINT requires either GMP or Multiple Precision Integers and Rationals (MPIR). This thesis will use the MPIR library for benchmarking. The only reason for this is for convenience since it already was installed and configured on Gantenbein, the system where the benchmarking is performed.

The other dependency FLINT has is the GNU MPFR library. On their website they describe it as "... a C library for multiple-precision floating-point computations with *correct rounding*.".

3.1.c Gantenbein

The benchmarking is done on a computer system called Gantenbein. It is a super computer with 120 cores and is designed with high performance computing in mind. The main reasons for choosing to test performance on it was partly because it's designed to perform heavy calculations which is fits with the theme of this thesis perfectly. Another reason was also that it's a system used by many people. This means an improvement to FLINT in Gantenbein will ensure that as many as possible will be able to benefit from the improvements done in this thesis.

3.2 Procedure

The thesis work has can be roughly sectioned into the following parts.

- Using the existing library zn_poly and use it to construct a fast TFT algorithm that can be fully integrated into FLINT.

- Investigating the auto-vectorization capabilities of the GCC compiler in order to boost the performance of the TFT algorithm.

3.2.a Modifying zn_poly

zn_poly is an extensive library with a many functions that interact with each other in a complex manner. Configuring the code to suit the needs of the thesis consisted of many parts.

The zn_poly library is made up of various separate program files, header files and test files to name a few. All programs related to testing and tuning as well as files related to configuration, demonstration of the program, separate documentation, profiling and other administrative texts were removed. Essentially, any file that wasn't directly part of a algorithm was removed.

3.2.a.1 Removing unrelated files

After this procedure there the task of removing any functions that wasn't used by the TFT algorithm remained. The zn_poly is a fairly extensive library capable of performing a variety of arithmetic operations Fourier transform. This required learning the library structure and how the various functions interacted with each other. This was done manually in a naive fashion by inspecting each function and checking all possible program outcomes and removing the functions that were never called. This was in hindsight not a efficient move since this procedure is already automated by numerous programs, one notable example being doxygen, that does this in an instant.

Many functions in the zn_poly library had similar or identical ones in FLINT which meant they had to be replaced. This helped reduce the complexity of the code since the functions in FLINT could be used, which meant that troubleshooting the code became much easier since the opportunities for mistakes lessened. This doesn't guarantee that mistakes are ruled out in those replaced functions since the FLINT library itself might be flawed. The risk is however mitigated since it's been tested and used by several users.

3.2.a.2 Changing function names/documentation

The later part of the thesis work entailed in writing quality documentation and rewriting the code to ensure that any potential user in the future will be able to use and understand the program as easily as possible. This documentation includes a quick summary of each function and its role in the overall algorithm. Each function and variable was named to eliminate any potential ambiguity and misunderstandings. This also entails adhering to a consistent coding standard. Each function name is a verb e.g. "GetFudgeFactorFromFFT" to explicitly tell the user what a function does in a single sentence. Each variable is explicitly named

unless in clear obvious cases. All iteration constants starts with the letter ”i”, ”j” and ”k” to signify its intent. There isn't a de facto or a universally recognized coding standard. Instead each standard is chosen depending on programming language, purpose and preference. This particular standard was chosen to make the code as understandable as possible and to maximize maintainability of the code. The intent is to make a new user to be able to quickly understand the program structure and function interaction.

3.2.b Auto-vectorization

The auto-vectorization was primarily tested with the GNU Compiler Collection (GCC). The reason for this choice was due to its widespread popularity which would ensure auto-vectorization for most users. The auto-vectorization in C language is enabled by compiling the program with the compiler flag -ftree-vectorize.

GCC has in theory a vectorization report, enabled with the flag fopt-info-vec, which shows explicit information about which sections the compiler succeeded and failed to vectorize. The excessive amount of information and the ambiguous messaging made this a near impossible task in practice.

Another way to study the auto vectorization is to look at the assembly output by using the compiler flag -S. The advantage of this is that there is no ambiguity involved as with the vectorization report. The disadvantage is that the assembly language is a low-level programming language that requires extensive knowledge about computer architecture as well as the myriad of instructions available. The amount of code also dramatically increases which makes it more difficult to gain an understanding of the program structure.

This method was used in the beginning of the thesis work to investigate the vectorization. The vectorization was investigated in a ”trial-and-error” fashion. The assembly output of two nearly identical versions of the same program was compared to each other. The goal was to study how each minute change affected the vectorization. Predicting the compiler behavior beforehand wasn't a viable option due to its complex and often times volatile nature, where seemingly insignificant changes had significant results.

Studying the assembly code yielded moderate success. The primary limitation was the mentioned disadvantages and by extension the author's lack of expertise in assembly programming. A complementary solution was to simply benchmark the vectorization. Two identical versions of a code was compared with each other, with one version having vectorization enabled. This made it possible to study to what extent the compiler vectorized and how effective it was in practice.

Three primary code sections was chosen for benchmarking: the TFT algorithm,

the ITFT and the butterfly operation. TFT and ITFT were obvious choices since they're a core part of this thesis. The butterfly operation was chosen because it's the most time-consuming, and therefore the most interesting, part of both algorithms.

3.2.b.1 Comparison of assembly output

It is possible to confirm that vectorization works as intended by comparing the assembly output of regular C code with vectorization enabled against the assembly output of intrinsic functions. In this example we will perform vector subtraction between two vectors of legnth 8. In regular C code the vector subtraction function would look like the following:

```
void sub(int c[], int const a[], int const b[])
{
  for(int i=0; i<8; ++i){
    c[i]=a[i]-b[i];
  }
}
```

Performig vector subtraction with intrinsics would instead take the following form:

```
#include "immintrin.h"
void sub_intrinsics(int c[], int const a[], int const b[])
{
  __m256i ar = _mm256_loadu_si256((__m256i const*)a);
  __m256i br = _mm256_loadu_si256((__m256i const*)b);
  __m256i cr = _mm256_sub_epi32(ar, br);
  _mm256_storeu_si256((__m256i *)c, cr);
}
```

Converting both codes to assembly yields the same core vector operations.

```
vmovdqu ymm0, YMMWORD PTR [rsi]
vpsubd  ymm0, ymm0, YMMWORD PTR [rdx]
vmovdqu YMMWORD PTR [rdi], ymm0
vzeroupper
```

The regular C code has some additional assembly instructions it performs that is absent from the intrinsic version. The additional code is simply a non-vectorized version of the subtraction the compiler can choose to use instead of the vectorized version. The important part, that is the vectorization, is identical which shows that auto-vectorization was successful.

4

Results

The results of the benchmarks between auto-vectorized compared to non auto-vectorized code is presented in this section. In the following tables in this section, k indicates the number of bits the modulus has and n indicates the maximum possible polynomial length that is randomized. The numbers shown are ratios between the time it takes to execute each implementation, that is

$$\frac{\text{non-vectorized execution time}}{\text{vectorized execution time}}. \tag{4.1}$$

Ratios greater than 1 means that vectorization is faster and less than 1 means non-vectorized implementation is faster. The ratios are color-coded into three colors: green if ratio is above 1.5, red if ratio is below 1/1.5 and black if it's in between.

4.1 TFT

The ratio of execution time between vectorized and non-vectorized TFT is presented in table 4.1. The average ratio of execution time for each value of k is presented in table 4.2 and the average execution time for each value of n is presented in table 4.3. The average performance increase with auto-vectorization over all parameters is 2.23.

4.2 ITFT

The ratio of execution time between vectorized and non-vectorized ITFT is presented in table 4.4. The average ratio of execution time for each value of k is presented in table 4.5 and the average execution time for each value of n is presented in table 4.6. The average performance increase with auto-vectorization over all parameters is 1.64.

Table 4.1: Execution time ratio of TFT with and without auto–vectorization. k denotes the number of bits in modulo and n denotes the maximum randomized polynomial length.

k	$n/1024$												
	1		2		4		8		16		32		64
5	1.63	2.04	2.08	2.06	2.03	2.27	2.27	2.35	2.45	2.15	2.38	2.09	2.24
10	2.05	2.03	2.09	2.14	2.22	2.26	2.33	2.31	2.51	2.35	2.38	2.30	2.34
15	2.07	2.06	2.12	2.12	2.22	2.23	2.30	2.40	2.34	2.23	2.41	2.24	2.43
20	2.06	2.04	2.09	2.15	2.20	2.33	2.33	2.33	2.25	2.38	2.24	2.24	2.16
25	2.07	2.04	2.13	2.13	2.21	2.26	2.32	2.20	2.40	2.33	2.33	2.25	2.20
30	2.05	2.03	2.11	2.12	2.23	2.23	2.31	2.34	2.39	2.33	2.30	2.27	2.27
35	2.05	2.06	2.10	2.11	2.20	2.24	2.34	2.30	2.43	2.31	2.33	2.30	2.40
40	2.06	2.05	2.09	2.17	2.21	2.24	2.34	2.36	2.36	2.24	2.44	2.28	2.36
45	2.07	2.06	2.11	2.12	2.21	2.27	2.33	2.28	2.37	2.26	2.46	2.27	2.19
50	2.07	2.06	2.11	2.13	2.22	2.29	2.28	2.30	2.29	2.30	2.41	2.21	2.30
55	2.06	2.04	2.08	2.15	2.23	2.30	2.30	2.26	2.38	2.34	2.16	2.50	2.33
60	2.06	2.03	2.07	2.15	2.23	2.27	2.31	2.31	2.40	2.30	2.47	2.20	2.24

Table 4.2: Average execution time ratio of TFT with and without auto–vectorization over k, the number of bits of modulo.

k												
5	10	15	20	25	30	35	40	45	50	55	60
2.16	2.25	2.24	2.22	2.22	2.23	2.24	2.25	2.23	2.23	2.24	2.23

4.3 Butterfly operation

The ratio of execution time between vectorized and non-vectorized butterfly is presented in table 4.7. The average ratio of execution time for each value of k is presented in table 4.8 and the average execution time for each value of n is presented in table 4.9. The average performance increase with auto-vectorization over all parameters is 2.88.

Table 4.3: Average execution time ratio of TFT with and without auto–vectorization over n, the maximum randomized polynomial length.

$n/1024$												
1		2		4		8		16		32		64
2.03	2.04	2.10	2.13	2.20	2.27	2.31	2.31	2.38	2.29	2.36	2.26	2.29

Table 4.4: Ratio of execution time of ITFT with and without auto–vectorization. k denotes the number of bits in modulo and n denotes the maximum randomized polynomial length.

k	$n/1024$												
	1		2		4		8		16		32		64
5	1.44	1.40	1.46	1.46	1.55	1.65	1.72	1.76	1.78	1.68	1.80	1.85	1.89
10	1.39	1.41	1.49	1.51	1.59	1.58	1.67	1.70	1.82	1.65	1.84	1.62	1.85
15	1.40	1.40	1.47	1.50	1.59	1.62	1.70	1.68	1.79	1.71	1.78	1.76	1.81
20	1.40	1.40	1.44	1.50	1.62	1.62	1.69	1.67	1.74	1.77	1.84	1.68	1.79
25	1.41	1.40	1.47	1.50	1.61	1.65	1.70	1.70	1.80	1.69	1.74	1.79	1.83
30	1.41	1.42	1.47	1.49	1.59	1.62	1.72	1.70	1.79	1.70	1.81	1.92	1.93
35	1.41	1.43	1.49	1.51	1.61	1.61	1.72	1.70	1.75	1.76	1.73	1.81	1.94
40	1.41	1.40	1.47	1.52	1.63	1.61	1.69	1.66	1.75	1.68	1.78	1.72	1.92
45	1.41	1.43	1.47	1.53	1.61	1.57	1.68	1.69	1.75	1.70	1.84	1.84	1.81
50	1.43	1.43	1.48	1.49	1.62	1.64	1.70	1.73	1.81	1.70	1.86	1.76	1.88
55	1.41	1.41	1.48	1.51	1.59	1.63	1.72	1.70	1.79	1.70	1.74	1.74	1.72
60	1.41	1.43	1.46	1.52	1.61	1.61	1.70	1.63	1.77	1.81	1.81	1.75	1.81

Table 4.5: Average execution time ratio of ITFT with and without auto–vectorization over k, the number of bits of modulo.

k												
5	10	15	20	25	30	35	40	45	50	55	60
1.65	1.63	1.63	1.63	1.64	1.66	1.65	1.63	1.64	1.65	1.63	1.64

Table 4.6: Average execution time ratio of ITFT with and without auto–vectorization over n, the maximum randomized polynomial length.

$n/1024$												
1		2		4		8		16		32		64
1.41	1.41	1.47	1.51	1.60	1.62	1.70	1.69	1.78	1.71	1.80	1.77	1.85

Table 4.7: Ratio of execution time of butterfly with and without auto–vectorization. k denotes the number of bits in modulo and n denotes the maximum randomized polynomial length.

k	$n/1024$												
	1		2		4		8		16		32		64
5	2.72	2.93	3.01	3.04	2.93	2.87	2.83	2.84	2.80	2.87	2.85	2.86	2.79
10	2.72	2.86	2.96	2.98	2.96	2.87	2.91	2.92	2.84	2.90	2.86	2.93	2.85
15	2.74	2.87	2.95	2.98	2.91	2.87	2.85	2.84	2.87	2.78	2.94	2.86	2.91
20	2.74	2.87	2.94	2.94	2.93	2.89	2.84	2.84	2.86	2.88	2.86	2.93	2.87
25	2.73	2.86	2.96	2.96	2.92	2.87	2.86	2.92	2.88	2.85	2.96	2.84	2.94
30	2.73	2.87	2.94	2.99	2.94	2.87	2.83	2.82	2.86	2.74	2.83	2.95	2.83
35	2.71	2.85	2.94	2.99	2.98	2.89	2.90	2.85	2.91	2.91	2.85	2.85	2.77
40	2.73	2.88	2.95	3.02	2.96	2.87	2.90	2.88	2.84	2.84	2.88	2.88	2.86
45	2.75	2.84	2.96	2.99	2.95	2.87	2.86	2.87	2.91	2.86	2.84	3.04	2.76
50	2.75	2.85	2.95	2.96	2.98	2.85	2.88	2.85	2.83	2.94	2.80	2.82	2.85
55	2.71	2.87	2.95	2.99	2.97	2.87	2.85	2.84	2.90	2.87	2.89	2.86	2.82
60	2.72	2.85	2.95	2.95	2.96	2.92	2.90	2.87	2.86	2.87	2.86	2.86	2.75

Table 4.8: Average execution time ratio of butterfly with and without auto–vectorization over k, the number of bits of modulo.

k												
5	10	15	20	25	30	35	40	45	50	55	60
2.87	2.89	2.88	2.88	2.89	2.86	2.88	2.88	2.89	2.87	2.88	2.87

Table 4.9: Average execution time ratio of butterfly with and without auto–vectorization over n, the maximum randomized polynomial length.

$n/1024$												
1		2		4		8		16		32		64
2.73	2.87	2.96	2.98	2.95	2.88	2.87	2.86	2.86	2.86	2.87	2.89	2.83

5

Discussion

The results presented in section 4 as well as how they relate to the theory is discussed and analyzed in this section.

5.1 General outcome

The results show that auto-vectorization increased performance not only on average but for all tested parameter ranges for both the number of bits of modulo as well as polynomial length. It's important to note that this does not automatically imply parallelization by default is more performant than serialization. This is due to the overhead costs of associated with parallelization such as splitting calculations and combining results. The GCC compiler performs pre–calculations before attempting to parallelize to determine if it's efficient to do so. Determining if the increased performance is the result of the chosen algorithms always benefiting parallelization or the quality of the pre–calculations, or both and if so to what degree, is a difficult if not impossible question to answer given the current data.

The best average performance increase was the butterfly benchmark with a factor of 2.88, followed by the benchmark of TFT with a factor of 2.23 and lastly benchmark of ITFT with a factor of 1.64. The butterfly benchmark having the best performance increase is not surprising. Iterating over a sequence and performing butterfly operations on two pairwise data points is a perfect candidate for parallelization since the iterations are independent of each other. This essentially means that loop order does not matter and the task can be divided into multiple smaller tasks without affecting the result.

Why the TFT had worse performance improvements compared to butterfly could depend on multiple factors. One reason could be the presence of control flow, meaning the multiple if statements in TFT as can be seen in algorithm 2. Control flow hinders parallelization since the compiler has to perform a so-called if-conversion to convert the if statement to a vectorized instruction. Such conversions may be difficult or even impossible which may be the reason why the

improvements to TFT are poorer than for butterfly.

The ITFT is likely suffers from the same limitations to parallelization due to the similarity of the algorithms. The performance increase is however markedly worse compared to TFT. It's likely that this difference is due to the quantitative rather then the qualitative difference between TFT and ITFT since they're very similar algorithms. In essence, there are more if statements and recursive operations in ITFT compared to TFT which could be the reason for the slowdown.

5.2 Limitations

A big limitation of this study is that it was only implemented on a single computer architecture. This is important since auto-vectorization is entirely dependent on which architecture it's run on. It's very possible that the results in this thesis is only relevant for that on specific computer system and that auto-vectorization isn't beneficial in others.

Another drawback of the thesis is the lack of sophisticated statistical analysis. It measures only the average performance increase and says nothing about its reliability. Data such as the standard deviation over many measurements would help solidify the results further.

5.3 Future work

There are multitude of ways the work in this thesis can be improved and built upon.

Researching how different architectures affect auto-vectorization is an important aspect to consider in future studies. It may however be a difficult task to study in a consistent manner since it requires resources to acquire the necessary hardware as well as in-depth computer architecture knowledge.

Another possible course of action is to study how different compilers such as GCC, clang and ICC utilizes auto-vectorization and compare their strengths and weaknesses.

Bibliography

[1] Michael Heideman, Don Johnson, and Charles Burrus. Gauss and the history of the fast fourier transform. *IEEE ASSP Magazine*, 1(4):14–21, 1984.

[2] Joris van der Hoeven. The truncated fourier transform and applications. In *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*, ISSAC '04, pages 290–296, New York, NY, USA, 2004. ACM.

[3] Joris van der Hoeven. Notes on the truncated fourier transform. 2012.

[4] David Harvey. A cache-friendly truncated fft. *Theoretical Computer Science*, 410(27):2649 – 2658, 2009.

[5] Sonja Benz. Fast multiplication of multiple-precision integers. Master's thesis, Rochester Institute of Technology, 1991.

[6] Kamisetty Ramamohan Rao and Pat Yip, editors. *The Transform and Data Compression Handbook*. CRC Press, Inc., Boca Raton, FL, USA, 2000.

[7] Sakar Pudasaini. Computation of idft through forward dft. 01 2017.

[8] Paul Vrbik. An Illustrated Introduction to the Truncated Fourier Transform. *arXiv e-prints*, page arXiv:1602.04562, Feb 2016.

[9] Li Zhang. Implementation techniques for the truncated fourier transform. Master's thesis, The University of Western Ontario, 2015.

[10] Dorit Naishlos. Autovectorization in gcc. In *Proceedings of the 2004 GCC Developers Summit*, pages 105–118, 2004.

[11] Alen Stojanov, Ivaylo Toskov, Tiark Rompf, and Markus Püschel. Simd intrinsics on managed language runtimes. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, pages 2–15, New York, NY, USA, 2018. ACM.

[12] Saeed Maleki, Yaoqing Gao, María Garzarán, Tommy Wong, and David Padua. An evaluation of vectorizing compilers. pages 372–382, 10 2011.