

IMPLEMENTATION TECHNIQUES FOR THE TRUNCATED  
FOURIER TRANSFORM

Li Zhang  
lzhan494@uwo.ca

Supervised by Dr. Marc Moreno-Maza

September 14, 2015

Department of Computer Science  
The University of Western Ontario  
London, Ontario, Canada

© Li Zhang 2015

THE UNIVERSITY OF WESTERN ONTARIO  
School of Graduate and Postdoctoral Studies

**CERTIFICATE OF EXAMINATION**

Examiners:

Supervisor:

.....  
Dr. Arash Reyhani-Masoleh

.....  
Dr. Marc Moreno-Maza

.....  
Dr. Eric Schost

.....  
Dr. Robert Webber

The thesis by

**Li Zhang**

entitled:

**Implementation Techniques for the Truncated Fourier Transform**

is accepted in partial fulfillment of the  
requirements for the degree of  
Master of Science

.....  
Date

.....  
Chair of the Thesis Examination Board

## Abstract

We study various algorithms for the Truncated Fourier Transform (TFT) which is a variation of the Discrete Fourier Transform (DFT) that allows one to work with an input vector of arbitrary size without zero padding.

After a review of the original algorithms for the forward and inverse TFT introduced by J. van der Hoeven, we consider the variation of D. Harvey as well as that of J. Johnson and L.C. Meng. Both variations are based on Cooley-Tukey like formulas. The former is called *strict general radix* as it strictly follows the specifications proposed by J. van der Hoeven, while the latter is called *relaxed general radix* as it requires some zero padding so as to improve data flow which supports full vectorization and parallelization.

In this thesis, we report on an implementation of the relaxed general radix forward TFT and a strict general radix inverse TFT. We have three objectives. First, obtaining a software tool generating optimized code forward and inverse TFT, extending the previous work of S. Covanov dedicated to FFT code generation. Second, comparing the practical efficiency of the strict and relaxed general radix schemes. Third, investigating the parallelization of one-dimensional TFT algorithms.

Our experimental results show that, in practice, the relaxed general radix forward TFT can reach similar performance (in terms of running time, clock cycles and cache misses) as the optimized FFT code of the BPAS library (on input vectors on which both codes apply without zero padding). Moreover, for an input vector whose size ranges between two consecutive values for which FFT does not require zero padding, our relaxed TFT generated code provides an effective implementation. Unfortunately, the same satisfactory observation does not hold for the strict radix scheme when comparing the inverse TFT and FFT. As for parallelization, here again the relaxed general radix scheme is satisfactory while the strict general radix is not. For instance, w.r.t. to the FFT code, the parallel forward TFT code has a speedup factor of 5.31 and 6.78 for an input vector of size  $2^{23}$  and  $2^{26}$  respectively.

**Keywords.** Parallel Algorithms, High Performance Computing, TFT, Inverse TFT, Computer Algebra.

## Acknowledgments

First and foremost I would like to offer my sincerest gratitude to my supervisor, Dr Marc Moreno Maza, who has supported me throughout my thesis with his patience and knowledge. I attribute the level of my Masters degree to his encouragement and effort, and without him, this thesis would not have been completed or written.

Secondly, I would like to thank my academic brothers and sisters Ning Xie, Xiaohui Chen, Javad Doliskani, Parisa Alvandi and Dr. Paul Vrbik for working along with me and helping me complete this research work successfully. Special thanks to Svyatoslav Covanov and Andrew Arnold for helping me with Montgomery tricks and theory of the TFT. In addition, thanks to Shaun Li for reading this thesis and his useful comments.

Thirdly, all my sincere thanks and appreciation go to all the members from our Ontario Research Centre for Computer Algebra (ORCCA) lab in the Department of Computer Science for their invaluable support and assistance, and all the members of my thesis examination committee.

Finally, I would like to thank all of my friends and family members for their consistent encouragement and continued support.

I dedicate this thesis to my parents for their unconditional love and support throughout my life.

# Contents

<b>List of Algorithms</b>	<b>vi</b>
<b>List of Tables</b>	<b>vii</b>
<b>List of Figures</b>	<b>ix</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Literature review . . . . .	1
1.2 Contributions of this thesis . . . . .	2
<b>2 Background</b>	<b>4</b>
2.1 Rings and fields . . . . .	4
2.2 Montgomery arithmetic . . . . .	6
2.3 Primitive roots of unity . . . . .	7
2.4 Discrete Fourier transform (DFT) . . . . .	7
2.5 Fast Fourier transform (FFT) . . . . .	8
2.6 Montgomery arithmetic in practice . . . . .	8
2.7 Tensor algebra . . . . .	11
2.8 Cooley Tukey factorization formula . . . . .	13
2.9 Multi-core architectures . . . . .	14
2.10 The fork-join concurrency model . . . . .	15
2.11 The CilkPlus programming language . . . . .	17
2.12 The ideal cache model . . . . .	17
2.13 Cache complexity of data transposition . . . . .	20
2.14 Cache complexity of Cooley-Tukey algorithm . . . . .	21
2.15 Blocking strategy for FFT . . . . .	22
<b>3 Forward and Inverse Truncated Fourier Transform</b>	<b>25</b>
3.1 FFT: review and complement . . . . .	25
3.2 The truncated Fourier transform . . . . .	28

3.3	Forward TFT: pseudo-code with an illustrative example . . . . .	30
3.4	The inverse truncated Fourier transform . . . . .	31
3.5	Inverse TFT: an algorithm . . . . .	32
3.6	Illustration of the inverse TFT algorithm . . . . .	34
<b>4</b>	<b>The Relaxed General Radix TFT and Strict General Radix Inverse TFT</b>	<b>41</b>
4.1	Introduction . . . . .	41
4.2	A relaxed general-radix TFT algorithm . . . . .	42
4.3	A cache-friendly inverse TFT (ITFT) . . . . .	43
<b>5</b>	<b>Python Code Generator for TFT and Inverse TFT in C++/CilkPlus</b>	<b>45</b>
5.1	C++ code generation in Python . . . . .	45
5.2	The basic polynomial algebra subprograms . . . . .	47
5.2.1	Design and specification . . . . .	47
5.2.2	User interface . . . . .	48
5.2.3	BPAS's DFT code generator . . . . .	48
5.2.4	The use of the BPAS library . . . . .	49
5.3	Code generation for TFT and ITFT . . . . .	50
5.3.1	Details of the Python code generator . . . . .	51
5.3.2	The structure of the template file . . . . .	52
5.4	Optimization techniques . . . . .	53
5.4.1	The use of machine code . . . . .	53
5.4.2	Hard-coded constants . . . . .	54
5.4.3	Unrolling loops . . . . .	54
5.4.4	Work space . . . . .	54
5.4.5	Montgomery arithmetic . . . . .	54
5.4.6	Cache-efficient transpose . . . . .	55
5.4.7	Parallel code generation . . . . .	55
<b>6</b>	<b>Experimentation of Serial and Inverse TFT (ITFT)</b>	<b>61</b>
6.1	Experimental setup . . . . .	61
6.2	Comparison of serial code . . . . .	62
6.3	Results for serial TFT between two consecutive powers of two . . . . .	63
6.4	Results for TFT and ITFT parallel code . . . . .	63
<b>7</b>	<b>Conclusion</b>	<b>70</b>



# List of Algorithms

1	<code>transpose(sfixn *A, int lda, sfixn *B, int ldb, int i0, int i1, int j0, int j1)</code> . . .	18
2	<code>FFT<sub>radix K</sub>(<math>\alpha, \omega, n = J \cdot K</math>)</code> . . . . .	24
3	<code>FFT(<math>\alpha, \omega</math>)</code> . . . . .	24
4	<code>TFT(<math>X, \omega, p</math>)</code> . . . . .	30
5	<code>InvTFT(<math>\mathbf{x}, \text{head}, \text{tail}, \text{last}, s</math>)</code> . . . . .	33
6	<code>CACHEFRIENDLYTFT(<math>L, \zeta, z, n, f; (x_0, \dots, x_{L-1})</math>)</code> . . . . .	44
7	<code>DFT<sub>eff</sub>(<math>n, A, \Omega, H</math>)</code> . . . . .	49
8	<code>Shuffle(<math>n, A</math>)</code> . . . . .	49
9	<code>DFT<sub>rec</sub>(<math>n, A, \Omega, H</math>)</code> . . . . .	50
10	<code>TFT_8POINT(sfixn *A, sfixn *W)</code> . . . . .	53
11	<code>TFT_Core(<math>invec, \omega, p, n, \ell, m, basecase, invectmp</math>)</code> . . . . .	56
12	<code>MontMulModSpe_OPT3_AS_GENE_INLINE(sfixn a, sfixn b)</code> . . . . .	57
13	<code>unrolledSpe8MontMul(sfixn* input1, sfixn* input2, MONTP_OPT2_AS_GENE * pPtr)</code> . . . . .	57
14	<code>AddModSpe(sfixn a, sfixn b)</code> . . . . .	58
15	<code>SubModSpe(sfixn a, sfixn b)</code> . . . . .	58
16	<code>Prod_Inv(<math>x, y, z, p</math>)</code> . . . . .	58
17	<code>Prod_Inv_Mont(<math>x, y, z, p</math>)</code> . . . . .	58
18	<code>transpose_serial(sfixn *A, int lda, sfixn *B, int ldb, int i0, int i1, int j0, int j1)</code> . . . . .	59
19	<code>FFT_8POINT(sfixn *A, sfixn *W)</code> . . . . .	59
20	<code>DFT<sub>iter</sub>(<math>n, A, \Omega</math>)</code> . . . . .	60
21	<code>FFT_2POINT(sfixn *A, sfixn *W)</code> . . . . .	60
22	<code>FFT_4POINT(sfixn *A, sfixn *W)</code> . . . . .	60



# List of Tables

6.1	Clock cycles for serial FFT, TFT and ITFT with input size $n$ . . . . .	62
6.2	Cache misses for serial FFT, TFT and ITFT with input size $n$ . . . . .	64
6.3	Cilkview analysis of parallel TFT on input size $N$ , where $work$ , and $span$ rows are the number of instructions, and $parallelism$ is the ratio of $Work/Span$ . . . . .	67
6.4	Cilkview analysis of parallel ITFT on input size $N$ , where $work$ , and $span$ rows are the number of instructions, and $parallelism$ is the ratio of $Work/Span$ . . . . .	67
6.5	Running time (secs) for serial FFT, serial TFT and parallel TFT with grain size of 1024 on 12 cores) and the speedup between serial FFT and parallel TFT and between serial TFT and parallel TFT. . . . .	69

# List of Figures

2.1	The ideal-cache model. . . . .	19
2.2	Scanning an array of $n = N$ elements, with $L = B$ words per cache line. . .	19
2.3	Algorithm 3 strategy. . . . .	22
2.4	Optimal FFT using blocking. . . . .	23
3.1	Butterfly. . . . .	27
3.2	Butterflies. Schematic representation of Equation (3.1). The black dots correspond to the $x_{s,i}$ . The top row corresponding to $s = 0$ . In this case $n = 16 = 2^4$ . . . . .	27
3.3	The Fast Fourier Transform for $n = 16$ . The top row, corresponding to $s = 0$ , represents the values of $\mathbf{x}_0$ . The bottom row, corresponding to $s = 4$ is some permutation of $\hat{\mathbf{a}}$ (the result of the FFT on $\mathbf{a}$ ). . . . .	28
3.4	The FFT with “artificial” zero points (green). . . . .	29
3.5	Removing all unnecessary computations from Figure 3.4 gives the schematic representation of the TFT. . . . .	29
3.6	Example of TFT where $n = 16, \ell = 9$ , prime number is 17, and $\omega = 3$ . . . . .	35
3.7	The relation for no butterfly. . . . .	35
3.8	tail $\geq$ LeftMiddle (i.e. at least half the values are at $x = p$ ). . . . .	36
3.9	tail $<$ LeftMiddle (i.e. less than half the values are at $x = p$ ). . . . .	37
3.10	Schematic representation of the recursive computation of the Inverse TFT for $n = 16$ and $\ell = 11$ . . . . .	38
3.11	The first part of ITFT example. . . . .	39
3.12	The second part of ITFT example. . . . .	40
4.1	An example of factoring $\text{TFT}_{32,17,17}$ with the relaxed general-radix TFT algorithm. . . . .	43
5.1	A snapshot of BPAS algebraic data structures. . . . .	48
6.1	Running time (secs) of serial FFT, TFT and ITFT. . . . .	63
6.2	TFT and ITFT results on a range between $2^{22}$ and $2^{23}$ on a 12 cores node. . . . .	65

6.3	TFT and ITFT results on a range between $2^{23}$ and $2^{24}$ on a 12 cores node.	65
6.4	TFT speedup on 4 cores and 12 cores. . . . .	66
6.5	ITFT speedup on 4 cores and 12 cores. . . . .	66
6.6	Parallel TFT with different grain sizes. . . . .	68
6.7	Parallel ITFT with different grain sizes. . . . .	68
A.1	Python code. . . . .	74

# Chapter 1

## Introduction

The discrete Fourier transform (DFT) plays a fundamental role in *digital signal processing* and *computer algebra*. In the latter case, coefficients<sup>1</sup> are in a *finite field* and  $K$ -way Cooley-Tukey *fast Fourier transform* (FFT) is commonly used, while in the former case, coefficients are usually complex numbers and other schemes like mixed-radix are preferred.

Over finite fields,  $K$ -way Cooley-Tukey FFTs can be implemented efficiently, for well-chosen  $K$ . However, when the input vector has a size varying between two consecutive powers of  $K$ , say between  $K^e + 1$  and  $K^{e+1}$ , a  $K$ -way FFT has the same cost (that at  $K^{e+1}$ ) in terms of arithmetic operations.

*Truncated Fourier transforms* TFT deal with this challenge but the complex data flow of those algorithms make them hard to implement efficiently. This thesis compares experimentally different schemes for implementing TFT both serially and in parallel.

### 1.1 Literature review

The original TFT algorithms of Joris van der Hoeven [26] has stimulated a significant research activity. It was integrated in various software libraries, like the `modpn` library [17] where it was used in a building block, in particular for multi-dimensional FFT-like transforms [19] and their application to dense multivariate polynomial arithmetic [20].

For the one-dimensional case, improvements to the algorithms of Joris van der Hoeven were proposed by David Harvey [12] and by Lingchuan Meng and Jeremy R. Johnson [21]. In the former case, these enhancements are in terms of cache complexity, even though the paper does not phrase things in such terms; in the latter case, data flow is simplified (to the expense of slightly increasing the algebraic complexity) so as to offer more

---

<sup>1</sup>Often, we have  $K = 2$ .

opportunities for concurrent computations to take place.

Both the variation of David Harvey and that of Lingchuan Meng and Jeremy R. Johnson are based on a Cooley-Tukey like formula. The former is called *strict general radix* as it strictly follows the specifications proposed by J. van der Hoeven, while the latter is called *relaxed general radix* as it requires some zeroes padding to improve data flow that supports full vectorization and parallelization.

## 1.2 Contributions of this thesis

L.C. Meng and J. R. Johnson have exhibited Cooley-Tukey-like formulas (called *relaxed*, *strict*) for TFFT (forward and inverse) but do not provide pseudo-code nor publicly available code (as of August 2015 when this thesis was written). We propose pseudo-code for their *relaxed* Cooley-Tukey-like formula and a Python code generator integrated into the *Basic Polynomial Algebra Subprograms* (BPAS)<sup>2</sup> for both forward and inverse FFT. Our generated code can be serial (C++) or parallel (CilkPlus).

Our second contribution is experimental. Thanks to Svyatoslav Covanov [4], BPAS has a serial-FFT Python generator which produces highly optimized and competitive code. For appropriate input vectors, we compare the serial-FFT and serial-TFFT (both forward and inverse) codes produced by the BPAS code generators (the one of S. Covanov and ours). The forward serial-TFFT (which uses the relaxed formula) is competitive while the inverse serial-TFFT (which uses the strict formula) suffers, as expected, from a more complex data flow. Our generated parallel forward TFFT code provides interesting speedup factors, beneficial to the BPAS library. For instance, w.r.t. to the FFT code, the parallel forward TFFT code has a speedup factor of 5.31 and 6.78 for an input vector of size  $2^{23}$  and  $2^{26}$  respectively.

This thesis is organized as follows. In Chapter 2, we briefly review finite field arithmetic and DFT computations over such fields, the fork-join concurrency model, and CilkPlus programming language, the ideal cache model and cache complexity results for FFT algorithms.

In Chapter 3, we review the original algorithms for TFFT and its inverse, as they were proposed by J. van der Hoeven. In Chapter 4, we describe the variation of D. Harvey as well as that of J. Johnson and L.C. Meng. We stress the fact that David Harvey in [12] proposed conceptually simpler ways of computing TFFTs compared to J. van der Hoeven and this inspired the work of J. Johnson and L.C. Meng [21] which has brought a practically efficient forward TFFT algorithm.

---

<sup>2</sup>This library is available in source at [www.bpaslib.org](http://www.bpaslib.org).

In Chapter 5, we rely on the `BPAS` library to fulfil the implementation of our Python code generator. We take advantage of the Python code generator framework designed by Svyatoslav Covanov for FFT. Our experimental results are collected in Chapter 6. It includes the comparisons of running times, clock cycles, cache misses as well as `Cilkview` analysis results such as speedup factors, work and burdened span.

# Chapter 2

## Background

In this chapter, we review basic concepts related to high-performance implementation of the truncated Fourier transform (TFT) and the inverse TFT (ITFT). We start with the definition of rings and fields in Section 2.1. We continue with the Montgomery arithmetic, described in Section 2.2, which plays an important role in our algorithms. We introduce the definition of primitive roots of unity in Section 2.3. The algorithm of the fast Fourier transform (FFT) is summarized in Section 3.1. We describe the implementation of Montgomery arithmetic in practice in Section 2.6. Further, we review basic notions of tensor algebra 2.7 which is used as a particular factorization of the  $DFT_n$  in the FFT algorithm, follow the PhD thesis of Wei Pan <http://www.csd.uwo.ca/~moreno/Publications/Wei.Pan-Thesis-UWO.pdf>. The Cooley Tukey factorization formula is summarized in Section 2.14.

We continue with an introduction of multi-core architectures in Section 2.9 and the fork-join concurrency model in Section 2.10, follow the Master thesis of Farnam Mansouri [18]. We give a brief description of `CilkPlus` programming language in Section 2.11. The theory behind the ideal cache model can be found in Section 2.12. Then, we describe the cache complexity of data transposition in Section 2.13. Cache complexity of Cooley-Tukey algorithm is analyzed in Section 2.14. The blocking strategy for FFT can be found in Section 2.15.

### 2.1 Rings and fields

In algebra, a *ring* is a (non-empty) set  $\mathcal{R}$  endowed with two binary operations, denoted additively and multiplicatively. Both are required to be associative and have a neutral element (denoted 0 and 1, respectively). Moreover, the addition must be commutative

and each  $x \in \mathcal{R}$  must admit an opposite element, denoted by  $-x$ , such that  $x + (-x) = 0$  holds. Finally, the multiplication must be distributive w.r.t. the addition. For more details, see [https://en.wikipedia.org/wiki/Ring\\_%28mathematics%29](https://en.wikipedia.org/wiki/Ring_%28mathematics%29).

Examples of rings are: (1) the set  $\mathbb{Z}$  of (positive and negative) integers, (2) the set of square matrices of order  $n$ , for a given positive integer  $n$ , with coefficients in  $\mathbb{Z}$ , (3) the set of univariate polynomials with coefficients in  $\mathbb{Z}$  and (4) the set  $\mathbb{Z}/m\mathbb{Z}$  of integers modulo  $m$ , where  $m$  is a given positive integer.

When the multiplication itself is commutative, the ring  $\mathcal{R}$  is called commutative. If each non-zero  $x \in \mathcal{R}$  also admits an inverse, denoted by  $x^{-1}$  or  $1/x$ , such that  $x \times x^{-1} = 1$  holds, then the commutative ring  $\mathcal{R}$  is said to be a *field*.

Examples of fields are: (1) the set  $\mathbb{Q}$  of rational numbers, (2) the set  $\mathbb{R}$  of real numbers, (3) the set  $\mathbb{C}$  of complex numbers, and (4) the set  $\mathbb{F}_p := \mathbb{Z}/p\mathbb{Z}$  where  $p$  is a prime number.

Fields of the form  $\mathbb{F}_p$  play a fundamental role in algebra and are called *prime fields*. Elements of  $\mathbb{F}_p$  are the residue classes of the equivalence relation on  $\mathbb{Z} \times \mathbb{Z}$  defined by

$$a \equiv b \pmod{p} \iff p \text{ divide } (a - b).$$

Let  $\bar{a}, \bar{b} \in \mathbb{F}_p$ , be represented by  $a, b \in \mathbb{Z}$  respectively. The sum  $\bar{a} + \bar{b}$  and the product  $\bar{a} \times \bar{b}$  are given by  $\bar{r}$  and  $\bar{s}$ , where  $r$  (resp.  $s$ ) is the remainder in the Euclidean division of  $a + b$  (resp.  $a \times b$ ) by  $p$ .

Consider now the implementation of  $\mathbb{F}_p$  on computers. Let  $w_s$  be the size in bits of a machine word, which is assumed to be even. Assume that elements of  $\mathbb{F}_p$  are encoded by the non-negative integers  $0, 1, \dots, p-1$ . We focus here on the case where  $p$  is a prime number such that

$$2(p-1) \leq 2^{w_s} - 1$$

holds (for a reason that will become clear shortly) thus implying the inequality

$$\lceil \log_2(p) \rceil + 1 \leq w_s,$$

that is, all integers in the range  $0, 1, \dots, p-1$  can be written on a single machine word. Clearly, the addition  $(a, b) \mapsto \bar{a} + \bar{b}$  is easily implemented using machine word operations. Here's a C function illustrating that fact and which is correct thanks to our assumption  $2(p-1) \leq 2^{w_s} - 1$ :

```

sfixn AddMod(sfixn a, sfixn b, sfixn p){
    sfixn r = a + b;
    r -= p;
}

```



```

    r += (r >> BASE_1) & p;
  return r;
}

```

where `sfixn` is the type of a machine word and `BASE_1` is  $w_s - 1$ .

Implementing the multiplication  $(a, b) \mapsto \bar{a} \times \bar{b}$  with machine word operations is a more delicate task, unless  $(p - 1)^2 \leq 2^{w_s} - 1$  holds. The next section presents an efficient solution.

## 2.2 Montgomery arithmetic

Let  $x, p$  be integers such that  $p > 2$  is a prime. We shall compute  $x \bmod p$  in an *indirect way*, following an idea proposed by Peter Montgomery in [22]. Consider a positive integer  $R \geq p$  such that  $\gcd(R, p) = 1$ . Hence there exists integers  $R^{-1}, p'$  such that

$$RR^{-1} - pp' = 1 \quad \text{and} \quad 0 < p' < R.$$

Consider the following two Euclidean divisions <sup>1</sup>

$$x \left| \begin{array}{l} R \\ c \end{array} \right. \quad \text{and} \quad dp' \left| \begin{array}{l} R \\ e \end{array} \right.$$

Hence we have:

$$x + fp = cR + d + (dp' - eR)p = cR + d(1 + pp') - epR.$$

Therefore  $x + fp$  writes  $qR$  and thus  $\frac{x}{R} \equiv q \pmod{p}$ . Suppose  $R$  is a power of 2. Then we have obtained a procedure computing  $\frac{x}{R} \bmod p$  for any  $0 \leq x < p^2$ , amounting to 2 multiplications, 2 additions and 3 shifts. Recall the three divisions (actually shifts):

$$x \left| \begin{array}{l} R \\ c \end{array} \right. \quad \text{and} \quad dp' \left| \begin{array}{l} R \\ e \end{array} \right. \quad \text{and} \quad x + fp \left| \begin{array}{l} R \\ 0 \\ q \end{array} \right.$$

The result is  $q$  or  $q - p$  since  $\frac{x}{R} \equiv q \pmod{p}$  and we have:

$$0 \leq x < p^2 \quad \Rightarrow \quad 0 \leq q < 2p.$$

---

<sup>1</sup>For non-negative integers  $a, b, q, r$ , with  $b > 0$ , we write  $a \left| \begin{array}{l} b \\ q \end{array} \right.$  whenever  $a = bq + r$  and  $0 \leq r < b$  both hold. See [https://en.wikipedia.org/wiki/Euclidean\\_division](https://en.wikipedia.org/wiki/Euclidean_division) for details.

It follows that to compute in  $\mathbb{Z}/p\mathbb{Z}$ , we map each  $a \in \mathbb{Z}/p\mathbb{Z}$  to  $\underline{a} := aR \in \mathbb{Z}/p\mathbb{Z}$ . Then the above procedure gives us  $\frac{aRbR}{R} \pmod{p}$ , that is,  $\underline{ab}$  the image of  $ab$  in this new representation. We call *Montgomery multiplication* the map  $(\underline{a}, \underline{b}) \in \mathbb{F}_p \times \mathbb{F}_p \mapsto \underline{ab}$ . Note that we have  $\underline{a+b} \equiv \underline{a} + \underline{b} \pmod{p}$ .

In summary, although the map  $a \in \mathbb{Z}/p\mathbb{Z} \mapsto \underline{a} \in \mathbb{Z}/p\mathbb{Z}$  is not a ring homomorphism, one can think of it as it were. To be precise, if an algorithm performs a sequence of additions and multiplications in  $\mathbb{Z}/p\mathbb{Z}$ , one can replace each residue class  $\bar{a}$  by  $\underline{a}$  provided that the products are computed by Montgomery multiplication. Section 2.6 contains C code for this procedure. Before that we shall review the discrete and fast Fourier transforms.

## 2.3 Primitive roots of unity

Let  $\mathcal{R}$  be a commutative ring. Let  $n > 1$  be an integer. An element  $\omega \in \mathcal{R}$  is a *primitive*  $n$ -th root of unity if for  $1 < k \leq n$  we have:

$$\omega^k = 1 \iff k = n.$$

The element  $\omega \in \mathcal{R}$  is a *principal*  $n$ -th root of unity if  $\omega^n = 1$  and for all  $1 \leq k < n$  we have

$$\sum_{j=0}^{n-1} \omega^{jk} = 0. \quad (2.1)$$

In particular, if  $n$  is a power of 2 and  $\omega^{n/2} = -1$ , then  $\omega$  is a principal  $n$ -th root of unity. When  $\mathcal{R}$  is a field, every primitive root of unity of  $\mathcal{R}$  is also a principal root of unity in  $\mathcal{R}$ .

## 2.4 Discrete Fourier transform (DFT)

Let  $\omega \in \mathcal{R}$  be a principal  $n$ -th root of unity. The  $n$ -point DFT at  $\omega$  is the linear function, mapping the vector  $a := (a_0, \dots, a_{n-1}) \in \mathcal{R}^n$  to the vector  $\hat{a} = (\hat{a}_0, \dots, \hat{a}_{n-1}) \in \mathcal{R}^n$  with

$$\hat{a}_i = \sum_{j=0}^{n-1} a_j \omega^{ij}.$$

If  $n$  admits an inverse in  $\mathcal{R}$ , then the  $n$ -point DFT at  $\omega$  has an inverse map which is  $1/n$  times the  $n$ -point DFT at  $\omega^{-1} = \omega^{n-1}$ .

Alternatively we can see the vector  $a$  as the coefficient array of a polynomial  $A$  from  $\mathcal{R}[x]$  (with degree less than  $n$ ) and interpret the  $n$ -point DFT at  $\omega$  as the mapping which

takes  $A = a_0 + a_1x + \dots + a_{n-1}x^{n-1}$  to the vector  $(A(\omega^0), \dots, A(\omega^{n-1}))$ . It is convenient to denote this by:

$$\text{DFT}_\omega(a_0, \dots, a_{n-1}) = (A(\omega^0), \dots, A(\omega^{n-1})).$$

The DFT has major applications in signal processing and computer algebra. In the former case, the ring  $\mathcal{R}$  is often the field  $\mathbb{C}$  of complex numbers whereas in the latter case, it is generally a prime field.

A *fast Fourier Transform* is an asymptotically fast algorithm for computing the  $n$ -point DFT of a vector over  $\mathcal{R}$ .

## 2.5 Fast Fourier transform (FFT)

From now on, we assume that  $n = 2^e$  for some positive integer  $e$ . Then, the DFT can be computed using binary splitting. This method requires that we evaluate the polynomial  $A$  only at  $\omega^{2^i}$  for  $i \in (0, \dots, e-1)$ , rather than at all powers  $\omega^0, \dots, \omega^{n-1}$ . To compute the DFT of  $a$  at  $\omega$  we write:

$$(a_0, \dots, a_{n-1}) = (b_0, c_0, \dots, b_{n/2-1}, c_{n/2-1})$$

and recursively compute the DFT of  $(b_0, \dots, b_{n/2-1})$  and  $(c_0, \dots, c_{n/2-1})$  w.r.t  $\omega^2$ :

$$\begin{aligned} \text{DFT}_{\omega^2}(b_0, \dots, b_{n/2-1}) &= (\hat{b}_0, \dots, \hat{b}_{n/2-1}); \\ \text{DFT}_{\omega^2}(c_0, \dots, c_{n/2-1}) &= (\hat{c}_0, \dots, \hat{c}_{n/2-1}); \end{aligned} \tag{2.2}$$

Finally we construct  $\hat{a}$  according to:

$$\text{DFT}_\omega(a_0, \dots, a_{n-1}) = (\hat{b}_0 + \hat{c}_0, \dots, \hat{b}_{n/2-1} + \hat{c}_{n/2-1}\omega^{n/2-1}, \hat{b}_0 - \hat{c}_0, \dots, \hat{b}_{n/2-1} - \hat{c}_{n/2-1}\omega^{n/2-1}).$$

This leads to a 2-way divide-and-conquer, with recursive calls on half of the input and a merging phase whose work is proportional to the input data size. Therefore, its running is in  $\Theta(n \log(n))$  operations on coefficients. Since its running time is, up to a log factor, proportional to the input data size, this method, due to Cooley & Tukey [5], is considered as asymptotically fast. More generally, any algorithm computing  $\text{DFT}_\omega(a_0, \dots, a_{n-1})$  in that time is called a *fast Fourier transform*.

## 2.6 Montgomery arithmetic in practice

As in Section 2.2, suppose that  $p > 2$  is a prime. Moreover, suppose that it is a *Fourier prime*, that is, a prime number such that  $p - 1 = c2^n$  and  $\ell \leq 2n$  hold, where  $\ell :=$

$\lfloor \log_2(p) \rfloor + 1 \leq w$  on  $w$ -bit machine words. Fourier primes are clearly interesting in view of DFT computations since they support the 2-way DFT computation of large vectors, namely vectors of size  $2^n$ . Let  $R := 2^\ell$  and  $0 \leq x \leq (p-1)^2$ . We obtain  $\frac{x}{R} \pmod p$  by:

$$\frac{x}{r_1} \left| \frac{R}{q_1} \right. \quad \text{and} \quad \frac{c2^n r_1}{r_2} \left| \frac{R}{q_2} \right. \quad \text{and} \quad \frac{c2^n r_2}{0} \left| \frac{R}{q_3} \right.$$

Using  $c2^n \equiv -1 \pmod p$  we have:

$$\frac{x}{R} \equiv q_1 + \frac{r_1}{R} \equiv q_1 - q_2 - \frac{r_2}{R} \equiv q_1 - q_2 + q_3 \pmod p.$$

The last equality requires a proof. We have:

$$r_2 = c2^n r_1 - q_2 R = c2^n r_1 - q_2 2^\ell.$$

Hence  $2^n \mid r_2$  thus  $2^{2n} \mid c2^n r_2$  and  $R \mid c2^n r_2$ . Moreover we have:

$$-(p-1) < q_1 - q_2 + q_3 < 2(p-1).$$

Hence the desired output is either  $(q_1 - q_2 + q_3) + p$ , or  $q_1 - q_2 + q_3$  or  $(q_1 - q_2 + q_3) - p$ . Indeed  $0 \leq x \leq (p-1)^2$  and  $p \leq R$  imply

$$q_1 = x \text{ quo } R \leq (p-1)^2 / R < p-1.$$

Next, we have:  $q_2 = c2^n r_1 \text{ quo } R \leq c2^n = p-1$ , since  $r_1 < R$ . Similarly, we have  $q_3 < p-1$ .

We now describe the C implementation for 32-bit machine integers, assuming we have at hand the following function:

```
/**
 * Input : The addresses of two unsigned machine integers a, b
 * Output : Store (a * b) quo 2^32 into a, and
            store (a * b) mod 2^32 into b
 **/
inline void MulHiLoUnsigned (uint32_t *a, uint32_t *b) {

    uint64_t prod;
    prod = (uint64_t)(*a) * (uint64_t)(*b);

    *a = (uint32_t) (prod >> 32);
    *b = (uint32_t) prod;
```

}

Then, Montgomery multiplication can be computed as follows.

1. Let  $a, b$  be non-negative 32-bit machine integers less than  $p$ . We state how to compute  $\frac{ab}{R} \bmod p$ .
2.  $q_1, 2^{32-\ell}r_1 := \text{MulHiLoUnsigned}(a, 2^{32-\ell}b)$
3.  $q_2, 2^{32-\ell}r_2 := \text{MulHiLoUnsigned}(2^{32-\ell}r_1, 2^n c)$
4.  $q_3 := c \frac{r_2}{2^{\ell-n}}$ . The division  $\frac{r_2}{2^{\ell-n}}$  is exact and the multiplication  $c \frac{r_2}{2^{\ell-n}}$  is correct on 32 bits.
5. Let  $A := q_1 - q_2 + q_3$ . Then we execute the following code:

```
A += (A >> 31) & p;
A -= p;
A += (A >> 31) & p;
```

6. Finally we have performed 6 shifts, 5 additions, 2 64-bit multiplications and 1 32-bit multiplication.

Here is a numerical example:

- Consider  $p = 257 = 1 + 2^8$ . Hence  $c = 1$ ,  $n = 8$ ,  $\ell = 9$  and  $R = 2^9$ .
- Take  $a = 131$  and  $b = 187$ .
- Compute  $2^{32-\ell}b = 1568669696$ .
- Compute  $q_1 = 47$  and  $2^{32-\ell}r_1 = 3632267264$ .
- Compute  $q_2 = 216$  and  $2^{32-\ell}r_2 = 2147483648$ .
- Compute  $q_3 = c \frac{r_2}{2^{\ell-n}} = 128$ .
- Compute  $A = q_1 - q_2 + q_3 = -41$ .
- Adjust to get  $\frac{ab}{R} \equiv 216 \pmod p$ .

## 2.7 Tensor algebra

Each FFT algorithm can be interpreted as a particular factorization of the  $\text{DFT}_n$  through tensor algebra. We review basic notions of the latter.

Let  $n, m, q, s$  be positive integers and let  $A, B$  be two matrices over  $\mathbb{K}$  with respective dimensions  $m \times n$  and  $q \times s$ . The tensor (or Kronecker) product of  $A$  by  $B$  is an  $mq \times ns$  matrix over  $\mathbb{K}$  denoted by  $A \otimes B$  and defined by

$$A \otimes B = [a_{k\ell}B]_{k,\ell} \quad \text{with} \quad A = [a_{k\ell}]_{k,\ell} \quad (2.3)$$

For example, let

$$A = \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}. \quad (2.4)$$

Then their tensor products are

$$A \otimes B = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 2 & 2 & 3 & 3 \\ 2 & 2 & 3 & 3 \end{bmatrix} \quad \text{and} \quad B \otimes A = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 2 & 3 & 2 & 3 \\ 0 & 1 & 0 & 1 \\ 2 & 3 & 2 & 3 \end{bmatrix}. \quad (2.5)$$

Denoting by  $I_n$  the identity matrix of order  $n$ , we emphasize two particular types of tensor products,  $I_n \otimes A_m$  and  $A_n \otimes I_m$ , where  $A_m$  (resp.  $A_n$ ) is a square matrix of order  $m$  (resp.  $n$ ) over  $\mathbb{K}$  that plays an important role in matrix factorization. A few examples follow:

$$I_4 \otimes \text{DFT}_2 = \begin{bmatrix} 1 & 1 & & & & & & \\ 1 & -1 & & & & & & \\ & & 1 & 1 & & & & \\ & & 1 & -1 & & & & \\ & & & & 1 & 1 & & \\ & & & & 1 & -1 & & \\ & & & & & & 1 & 1 \\ & & & & & & 1 & -1 \end{bmatrix}$$



We have

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} = \begin{bmatrix} x_0 \\ x_2 \\ x_4 \\ x_6 \\ x_1 \\ x_3 \\ x_5 \\ x_7 \end{bmatrix} \quad (2.11)$$

which shows that this matrix is as desired.

## 2.8 Cooley Tukey factorization formula

The well-known Cooley-Tukey Fast Fourier Transform (FFT) [6] in its recursive form is a procedure for computing  $\text{DFT}_n \mathbf{x}$  based on the following factorization of the matrix  $\text{DFT}_n$ , for any integers  $q, s$  such that  $n = qs$  holds:

$$\text{DFT}_{qs} = (\text{DFT}_q \otimes I_s) D_{q,s} (I_q \otimes \text{DFT}_s) L_q^{qs}, \quad (2.12)$$

where  $D_{q,s}$  is the diagonal twiddle matrix defined as

$$D_{q,s} = \bigoplus_{j=0}^{q-1} \text{diag}(1, \omega^j, \dots, \omega^{j(s-1)}), \quad (2.13)$$

Formula (2.14) illustrates Formula (2.12) with  $\text{DFT}_4$ :

$$\begin{aligned} \text{DFT}_4 &= (\text{DFT}_2 \otimes I_2) D_{2,2} (I_2 \otimes \text{DFT}_2) L_2^2 \\ &= \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \omega \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & -1 & -\omega \\ 1 & -1 & 1 & -1 \\ 1 & -\omega & -1 & \omega \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix}. \end{aligned} \quad (2.14)$$



Assume that  $n$  is a power of 2, e.g.,  $n = 2^k$ . Formula (2.12) can be unrolled so as to reduce  $\text{DFT}_n$  to  $\text{DFT}_2$  (or a base case  $\text{DFT}_m$ , where  $m$  divides  $n$ ) together with the appropriate diagonal twiddle matrices and stride permutation matrices. This unrolling can be done in various ways. Before presenting one of them, we introduce a notation. For integers  $i, j, h \geq 1$ , we define

$$\Delta(i, j, h) = (I_i \otimes \text{DFT}_j \otimes I_h) \quad (2.15)$$

which is a square matrix of size  $ijh$ . For  $m = 2^\ell$  with  $1 \leq \ell < k$ , the following formula holds:

$$\text{DFT}_{2^k} = \left( \prod_{i=1}^{k-\ell} \Delta(2^{i-1}, 2, 2^{k-i}) (I_{2^{i-1}} \otimes D_{2,2^{k-i}}) \right) \Delta(2^{k-\ell}, m, 1) \left( \prod_{i=k-\ell}^1 (I_{2^{i-1}} \otimes L_2^{2^{k-i+1}}) \right). \quad (2.16)$$

Therefore, Formula (2.16) reduces the computation of  $\text{DFT}_{2^k}$  to composing  $\text{DFT}_2$ ,  $\text{DFT}_{2^\ell}$ , diagonal twiddle endomorphisms and stride permutations. Another recursive factorization of the matrix  $\text{DFT}_{2^k}$  is

$$\text{DFT}_{2^k} = (\text{DFT}_2 \otimes I_{2^{k-1}}) D_{2,2^{k-1}} L_2^{2^k} (\text{DFT}_{2^{k-1}} \otimes I_2), \quad (2.17)$$

from which one can derive the Stockham FFT [25] as follows

$$\text{DFT}_{2^k} = \prod_{i=0}^{k-1} (\text{DFT}_2 \otimes I_{2^{k-1-i}}) (D_{2,2^{k-i-1}} \otimes I_{2^i}) (L_2^{2^{k-i}} \otimes I_{2^i}). \quad (2.18)$$

This is a basic routine that is implemented in our library (CUMODP <sup>2</sup>) as the FFT over a finite field (prime) targeted GPUs [23].

## 2.9 Multi-core architectures

A multi-core processor is an integrated circuit consisting of two or more processors. Having multiple processors would enhance the performance by giving the opportunity of executing tasks simultaneously. Ideally, the performance of a multi-core machine with  $n$  processors, is  $n$  times that of a single processor (considering that they have the same frequency).

In recent years, this family of processors has become popular and widely being used due to their performance and power consumption compared to single-core processors. In

---

<sup>2</sup><http://cumodp.org/>

addition, because of the physical limitations of increasing the frequency of processors, or designing more complex integrated circuits, most of the recent improvements have been in designing multi-core systems.

In different topologies for multi-core systems, the cores may share the main memory, cache, bus, etc. Plus, heterogeneous multi-cores may have different cores, however in most cases the cores are similar to each other.

In a multi-core system, we may have multi-level cache memories that can have a huge impact on performance. Having cache memories on each of the processors, gives the programmers an opportunity of designing extremely fast memory access procedures. Implementing a program that can take benefits from the cache hierarchy, with low cache misses rates, is known to be challenging.

There are numerous parallel programming languages for multi-core architectures. Well-known examples of these concurrency platforms are CilkPlus <sup>3</sup>, OpenMP <sup>4</sup>, MPI <sup>5</sup>.

## 2.10 The fork-join concurrency model

The *Fork-Join Parallelism Model* is a multi-threading model for parallel computing. In this model, execution of threaded programs is represented by *DAG* (*directed acyclic graph*) in which the vertexes correspond to threads, and edges (*strands*) correspond to relations between threads (forked or joined). *Fork* stands for ending one strand, and starting a couple of new strands; whereas, *join* is the opposite operation in which a couple of strands end and one new strand begins.

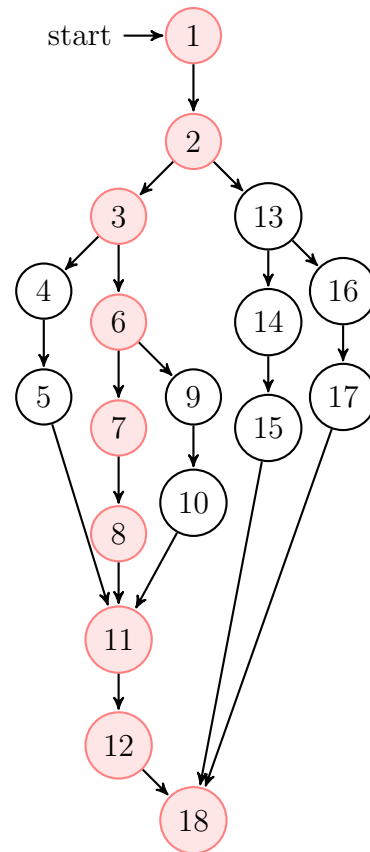
---

<sup>3</sup><http://www.cilkplus.org/>

<sup>4</sup><http://openmp.org/wp/>

<sup>5</sup>[http://en.wikipedia.org/wiki/Message\\_Passing\\_Interface](http://en.wikipedia.org/wiki/Message_Passing_Interface)

In the following diagram, a sample *DAG* is shown in which the program starts with the thread 1. Later, the thread 2 will be forked into two threads 3 and 13. Following the division of the program, the threads 15, 17 and 12 will be joined to 18.



For analyzing the parallelism in the fork-join model, we measure  $T_1$  and  $T_\infty$  which are defined as the following:

**Work ( $T_1$ ):** the total amount of time required to process all of the instructions of a given program on a single-core machine.

**Span ( $T_\infty$ ):** the total amount of time required to process all of the instructions of a given program on a multi-core machine with an infinite number of processors. This is also called the *critical path*.

**Work/Span Law:** the total amount of time required to process all of the instructions of a given program using a multi-core machine with  $p$  processors (called  $T_p$ ) bounded as the following:

$$T_p \geq T_\infty \quad , \quad T_p \geq \frac{T_1}{p}$$

**Parallelism:** the ratio of *work* to *span* ( $T_1/T_\infty$ ).

In the above *DAG*, the work, span, and the parallelism are 18, 9, and 2 respectively. (The *critical path* is highlighted.)

**Greedy Scheduler** A scheduler is *greedy* if it attempts to do as much work as possible at every step. In any greedy scheduler, there are two types of steps: **complete steps** in which there are at least  $p$  strands that are ready to run (then the greedy scheduler selects any  $p$  of them and runs them), and **incomplete step** in which there are strictly fewer than  $p$  threads that are ready to run (then the greedy scheduler runs them all).

**Graham-Brent Theorem** For any greedy scheduler, we have:  $T_p \leq T_1/p + T_\infty$ .

## 2.11 The CilkPlus programming language

CilkPlus is a C++ based concurrency platform providing an implementation of the fork-join concurrency model [16, 10, 7]. The CilkPlus runtime system offers a dynamic scheduler using the randomized *work-stealing* scheduling [3] in which every processor has a stack of pending tasks, and all of the processors can steal tasks from others' stacks when they are idle.

In CilkPlus, one can use the keywords `cilk_spawn` to spawn a function call, and `cilk_sync` as a synchronization point for concurrent threads. Algorithm 1 is an illustrative Cilkplus program which transposes a given rectangular matrix A into a matrix B:

In this implementation, we divide the problem into two sub problems based on the input sizes. If the dimension sizes of the sub problems are large enough, then the sub problems are solved recursively and the corresponding recursive calls are spawned, otherwise a serial code performs the transposition using the naive transposition algorithm. Note that the constant THRESHOLD is determined by consideration like the size of L1 cache.

## 2.12 The ideal cache model

The *cache complexity* of an algorithm aims at measuring the (negative) impact of memory traffic between the cache and the main memory of a processor executing that algorithm. Cache complexity is based on the *ideal-cache model* shown in Figure 2.1 which is taken from [10]. This idea was first introduced by Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran in 1999 [8]. In this model, there is a computer with a two-level memory hierarchy consisting of an ideal (data) cache of  $Z$  words and an arbitrarily large main memory. The cache is partitioned into  $Z/L$  *cache lines* where  $L$  is the length of each cache line representing the amount of consecutive words that are always moved in a group between the cache and the main memory. In order to achieve

---

**Algorithm 1:** transpose(sfixn \*A, int lda, sfixn \*B, int ldb, int i0, int i1, int j0, int j1)

---

**Input:**  $A, B$  matrix represented in array,  $lda$  number of columns,  $ldb$  number of rows,  $i0, i1$  index of rows,  $j0, j1$  index of columns.

**Output:** Array  $A$ .

/\* parallel version

\*/

tail:

int  $di = i1 - i0, dj = j1 - j0$ ;

if  $di \geq dj \&\& di > TRANSPOSETHRESHOLD$  then

    int  $im = (i0 + i1)/2$ ;

    cilk\_spawn transpose( $A, lda, B, ldb, i0, im, j0, j1$ );

$i0 = im$ ; goto tail;

else if  $dj > TRANSPOSETHRESHOLD$  then

    int  $jm = (j0 + j1)/2$ ;

    cilk\_spawn transpose( $A, lda, B, ldb, i0, i1, j0, jm$ );

$j0 = jm$ ; goto tail;

else

    for  $i$  from  $i0$  to  $i1$  do

        for  $j$  from  $j0$  to  $j1$  do

$B[j * ldb + i] = A[i * lda + j]$ ;

*spatial locality*, cache designers usually use  $L > 1$  which eventually mitigates the overhead of moving the cache line from the main memory to the cache. As a result, it is generally assumed that the cache is *tall* and practically that we have

$$Z = \Omega(L^2).$$

In the sequel of this thesis, the above relation is referred to as the *tall cache assumption*.

In the ideal-cache model, the processor can only refer to words that reside in the cache. If the referenced line of a word is found in cache, then that word is delivered to the processor for further processing. This situation is literally called a *cache hit*. Otherwise, a *cache miss* occurs and the line is first fetched into anywhere in the cache before transferring it to the processor; this mapping from memory to cache is called *full associativity*. If the cache is full, a cache line must be evicted. The ideal cache uses the optimal off-line cache replacement policy to perfectly exploit *temporal locality*. In this policy, the cache line whose next access is furthest in the future is replaced [2].

Cache complexity analyzes algorithms in terms of two types of measurements. The first one is the *work complexity*,  $W(n)$ , where  $n$  is the input data size of the algorithm.

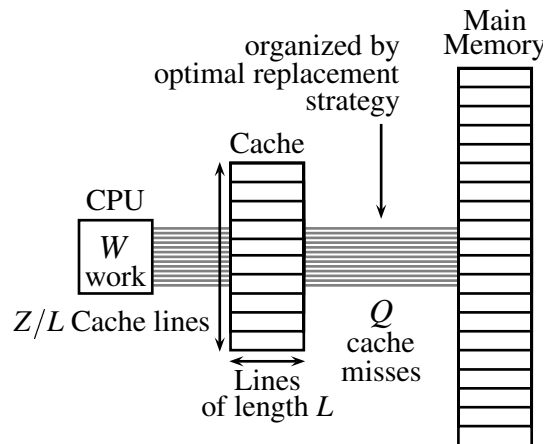


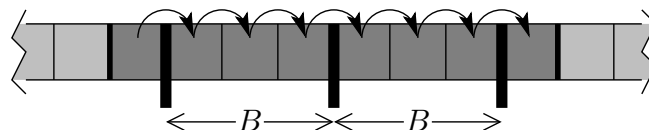
Figure 2.1: The ideal-cache model.

This complexity estimate is actually the conventional running time in a RAM model [1]. The second measurement is its *cache complexity*,  $Q(n; Z, L)$ , representing the number of cache misses the algorithm incurs as a function of:

- the input data size  $n$ ,
- the cache size  $Z$ , and
- the cache line length  $L$  of the ideal cache.

When  $Z$  and  $L$  are clear from the context, the cache complexity can be denoted simply by  $Q(n)$ .

An algorithm whose cache parameters can be tuned, either at compile-time or at run-time, to optimize its cache complexity, is called *cache aware*; while other algorithms whose performance does not depend on cache parameters are called *cache oblivious*. The performance of a cache-aware algorithm is often satisfactory. However, there are many approaches which can be applied to design optimal cache oblivious algorithms to run on any machine without fine tuning their parameters.

Figure 2.2: Scanning an array of  $n = N$  elements, with  $L = B$  words per cache line.

Although cache oblivious algorithms do not depend on cache parameters, their analysis naturally depends on the alignment of data block in memory. For instance, due to a specific type of alignment issue based on the size of block and data elements 2.2 (See Proposition 1 and its proof), the cache-oblivious bound is an additive 1 away from the

external-memory bound [14]. However, such type of error is reasonable as our main goal is to match bounds within multiplicative constant factors.

**Proposition 1** *Scanning  $n$  elements stored in a contiguous segment of memory with cache line size  $L$  costs at most  $\lceil n/L \rceil + 1$  cache misses.*

PROOF. The main ingredient of the proof is based on the alignment of data elements in memory. We make the following observations.

- Let  $(q, r)$  be the quotient and remainder in the integer division of  $n$  by  $L$ . Let  $u$  (resp.  $w_{un}$ ) be the total number of words in fully (not fully) used cache lines. Thus, we have  $n = u + w_{un}$ .
- If  $w_{un} = 0$  then  $(q, r) = (\lfloor n/L \rfloor, 0)$  and the scanning costs exactly  $q$ ; thus the conclusion is clear since  $\lceil n/L \rceil = \lfloor n/L \rfloor$  in this case.
- If  $0 < w_{un} < L$  then  $(q, r) = (\lfloor n/L \rfloor, w_{un})$  and the scanning costs exactly  $q + 2$ ; the conclusion is clear since  $\lceil n/L \rceil = \lfloor n/L \rfloor + 1$  in this case.
- If  $L \leq w_{un} < 2L$  then  $(q, r) = (\lfloor n/L \rfloor, w_{un} - L)$  and the scanning costs exactly  $q + 1$ ; the conclusion is clear again.

## 2.13 Cache complexity of data transposition

We consider the following problem, which plays a fundamental role in implementing multi-dimensional FFTs [24] and TFTs [19]. Given an  $m \times n$  matrix  $A$  stored in a row-major layout, compute and store the transposed matrix  $A^T$  into an  $n \times m$  matrix  $B$  also stored in a row-major layout. We shall describe a recursive cache-oblivious algorithm which uses  $\Theta(mn)$  work and incurs  $\Theta(1 + mn/L)$  cache misses, which is optimal. The straightforward algorithm employing doubly nested loops incurs  $\Theta(mn)$  cache misses on one of the matrices when  $m \gg Z/L$  and  $n \gg Z/L$ .

This recursive algorithm due to Leiserson et al. [9] works as follows:

- If  $n \geq m$ , the Rec-Transpose algorithm partitions

$$A = (A_1 \ A_2) , \quad B = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}$$

and recursively executes  $\text{Rec-Transpose}(A_1, B_1)$  and  $\text{Rec-Transpose}(A_2, B_2)$ .

- If  $m > n$ , the Rec-Transpose algorithm partitions

$$A = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} , \quad B = (B_1 \ B_2)$$

and recursively executes  $\text{Rec - Transpose}(A_1, B_1)$  and  $\text{Rec - Transpose}(A_2, B_2)$ .

The Cilkplus implementation of this algorithm is shown in Section 2.11

## 2.14 Cache complexity of Cooley-Tukey algorithm

We analyze the cache complexity of the (radix 2) Cooley-Tukey algorithm stated in Section 3.1. for an ideal cache with  $Z$  words and  $L$  words per cache line. We assume that each coefficient of the input vector fits within a machine word and that the array storing the coefficients consist of consecutive memory words. If  $Q(n)$  denotes the number of cache misses incurred by the algorithm of Section 3.1. then, neglecting misalignment, we have for some  $0 < \alpha < 1$ ,

$$Q(n) = \begin{cases} n/L & \text{if } n < \alpha Z \quad (\text{base case}) \\ 2Q(n/2) + n/L & \text{if } n \geq \alpha Z \quad (\text{recurrence}) \end{cases} \quad (2.19)$$

Unfolding  $k$  times the recurrence relation (2.19) yields

$$Q(n) = 2^k Q(n/2^k) + kn/L.$$

Assuming  $n \geq \alpha Z$  and choosing  $k$  such that  $n/2^k \simeq \alpha Z$ , that is,  $2^k \simeq \frac{n}{\alpha Z}$ , or equivalently  $n/L \simeq 2^k \alpha Z/L$ , we obtain

$$\begin{aligned} Q(n) &\leq 2^k \alpha Z/L + kn/L \\ &= n/L + kn/L \\ &= (k+1)n/L \\ &\leq (\log_2(\frac{n}{\alpha Z}) + 1)n/L. \end{aligned}$$

Therefore we have  $Q(n) \in O(n/L (\log_2(n) - \log_2(\alpha Z)))$ . This result is known to be non-optimal, following the work of Hong Jia-Wei and H.T. Kung in their landmark paper *I/O complexity: The red-blue pebble game* in the proceedings of STOC'81 [14].

Usually, this (non-optimal) radix 2 FFT is implemented as follows:

- If the input vector does not fit in cache, a recursive algorithm is applied
- Once the vector fits in cache, an iterative algorithm (not requiring shuffling) takes over.

This strategy is illustrated by Figure 2.3 [15] and Algorithm 3 below.



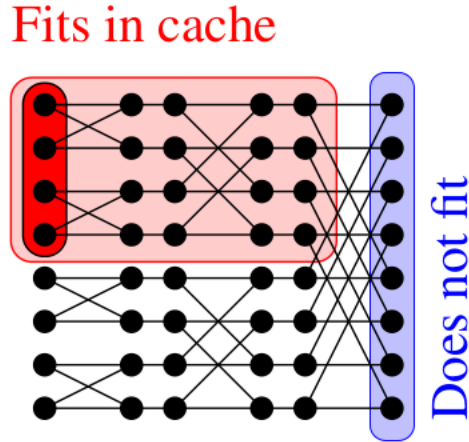


Figure 2.3: Algorithm 3 strategy.

## 2.15 Blocking strategy for FFT

To obtain an optimal FFT in terms of cache-complexity, one should proceed as follows

- Instead of processing row-by-row, one computes as deep as possible while staying in cache (resp. registers): this yields a **blocking strategy**.
- On the left picture, assuming  $Z = 4$ , on the first (resp, last) two rows, we successively compute the **red**, **green**, **blue**, **orange** 4-point blocks.
- On an ideal cache of  $Z$  words with  $L$  words per cache line the cache complexity drops to  $O(n/L(\log_2(n)/\log_2(Z)))$  which is **optimal**.

This strategy is illustrated by the picture and pseudo-code in Figure 2.4 and is reported in [4]. Figure 2.4 is taken from the Master thesis of Svyatoslav Covanov [www.csd.uwo.ca/~moreno//Publications/Svyatoslav-Covanov-Rapport-de-Stage-Recherche-2014.pdf](http://www.csd.uwo.ca/~moreno//Publications/Svyatoslav-Covanov-Rapport-de-Stage-Recherche-2014.pdf). The strategy is used by the BPAS library [www.bpaslib.org](http://www.bpaslib.org) for its FFT code generator. The work reported in this thesis extends this tool to TFT computations. Our TFT code generator also follows this blocking strategy.

Let us estimate now the cache complexity of the above algorithm for an ideal cache with  $Z$  words and  $L$  words per cache line. As before, we assume that each coefficient fits within a machine word. If  $Q(n)$  denotes the number of cache misses incurred by Algorithm 2, then, neglecting misalignment, we have for some  $0 < \alpha < 1$ ,

$$Q(n) = \begin{cases} n/L & \text{if } n < \alpha Z \quad (\text{base case}) \\ KQ(n/K) + n/L + n/KQ(K) & \text{if } n \geq \alpha Z \quad (\text{recurrence}) \end{cases} \quad (2.20)$$

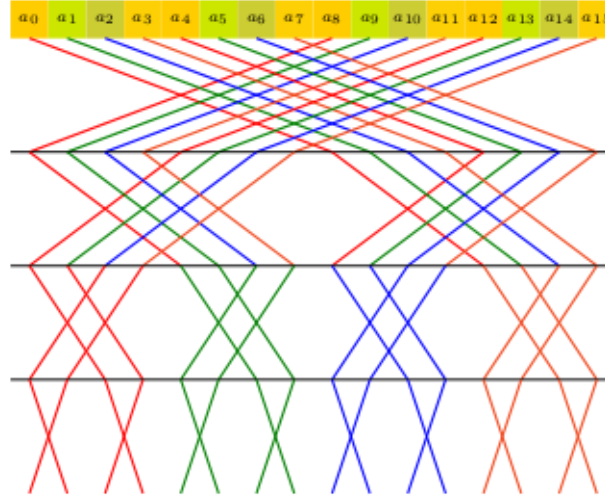


Figure 2.4: Optimal FFT using blocking.

We shall assume that  $K < \alpha Z$  holds. Hence, we have  $Q(K) \leq K/L$ . Thus, for  $n \geq \alpha Z$ , Relation (2.20) leads to:

$$\begin{aligned}
 Q(n) &= KQ(n/K) + 2n/L \\
 &\leq K^e Q(n/K^e) + 2en/L \\
 &\leq K^e \frac{\alpha Z}{L} + 2en/L \\
 &= n/L (1 + 2e) \\
 &\leq n/L 3e.
 \end{aligned} \tag{2.21}$$

where  $e$  is chosen such that  $n/K^e = \alpha Z$ , that is,  $K^e = \frac{n}{\alpha Z}$  or equivalently  $n/L = K^e \alpha Z/L$ . Therefore, we have  $Q(n) \in O(n/L (\log_K(n) - \log_K(\alpha Z)))$ . In particular, for  $K \simeq \alpha Z$  and since we have

$$Q(n) \in O(n/L \log_{\alpha Z}(n)). \tag{2.22}$$

According to the paper *I/O complexity: The red-blue pebble game*, this bound would be optimal for  $\alpha = 1$ . In practice  $\alpha$  is likely to 1/8 or 1/16 and  $Z$  is likely to be between 1024 and 8192 for an L1 cache. Hence, the above estimate of  $Q(n)$  suggests to choose  $K$  between 64 and 1024. In fact, in practice, we have experimented  $K$  between 8 and 16. The reason is that optimizing register usage (minimizing register spilling) is also another factor of performance and, to some sense, registers can be seen another level cache. As an example, the X86-64 processors that we have been using have 16 GPRs/data+address registers and 16/32 FP registers.

**Algorithm 2:**  $\text{FFT}_{\text{radix } K}(\alpha, \omega, n = J \cdot K)$ 


---

**Input:**  $\alpha = [a_0, a_1, \dots, a_{n-1}]$  is the coefficient array of the input polynomial,  $\omega$  is a primitive  $n$ -th root of unity,  $n = J \cdot K$  denotes  $n$  be split into  $K$  parts of size  $J$ .

**Output:** Array  $\alpha$ .

```

for  $0 \leq j < J$  do
  | /* Data transposition */
  | for  $0 \leq k < K$  do
  | |  $\gamma[j][k] := \alpha_{kJ+j}$ ;
for  $0 \leq j < J$  do
  | /* Base case FFTs */
  | |  $c[j] := \text{FFT}_{\text{base-case}}(\gamma[j], \omega^J, K)$ ;
for  $0 \leq k < K$  do
  | /* Twiddle factor multiplication */
  | | for  $0 \leq j < J$  do
  | | |  $\delta[k][j] := c[j][k] * \omega^{jk}$ ;
for  $0 \leq k < K$  do
  | /* Recursive calls */
  | |  $\zeta[k] = \text{FFT}_{\text{radix } K}(\zeta[k], \omega^K, J)$ ;
for  $0 \leq k < K$  do
  | /* Data transposition */
  | | for  $0 \leq j < J$  do
  | | |  $\alpha[jK + k] := \zeta[k][j]$ ;
return  $(\alpha_0, \alpha_1, \dots, \alpha_{n-1})$ ;

```

---

**Algorithm 3:**  $\text{FFT}(\alpha, \omega)$ 


---

**Input:**  $\alpha = [a_0, a_1, \dots, a_{n-1}]$  is the coefficient array of the input polynomial,  $\omega$  a primitive  $n$ -th root of unity.

**Output:** The output array  $\alpha$  becomes

$$[\alpha_0 + \alpha_{n/2}, \alpha_1 + \omega \cdot \alpha_{n/2+1}, \dots, \alpha_{n/2-1} - \omega^{n/2-1} \cdot \alpha_{n-1}].$$

**if**  $n \leq \text{HTHRESHOLD}$  **then**

```

  | ArrayBitReversal( $[\alpha_0, \alpha_1, \dots, \alpha_{n-1}]$ );
  | return  $\text{FFT\_iterative\_in\_cache}([\alpha_0, \alpha_1, \dots, \alpha_{n-1}], \omega)$ ;

```

Shuffle( $[\alpha_0, \alpha_1, \dots, \alpha_{n-1}]$ );

$$[\alpha_0, \alpha_1, \dots, \alpha_{n/2-1}] = \text{FFT}([\alpha_0, \alpha_1, \dots, \alpha_{n/2-1}], \omega^2);$$

$$[\alpha_{n/2}, \alpha_{n/2+1}, \dots, \alpha_{n-1}] = \text{FFT}([\alpha_{n/2}, \alpha_{n/2+1}, \dots, \alpha_{n-1}], \omega^2);$$

**return**  $[\alpha_0 + \alpha_{n/2}, \alpha_1 + \omega \cdot \alpha_{n/2+1}, \dots, \alpha_{n/2-1} - \omega^{n/2-1} \cdot \alpha_{n-1}]$ ;

---

# Chapter 3

## Forward and Inverse Truncated Fourier Transform

We review the notion of truncated Fourier transform (TFT) as introduced by Joris van der Hoeven in [26], together with detailed pseudo-code and examples for the forward and inverse TFT, follow Paul Vrbik’s tech report about TFT <https://carma.newcastle.edu.au/paulvrbik/pdfs/TFT.pdf>. We stress the fact those algorithms have the same specifications as those of David Harvey in [12]. However, the formulation in this latter paper opened the door to conceptually simpler ways of computing TFTs. In fact, David Harvey’s paper inspired the work of Jeremy Johnson and LingChuan Meng [21] which has brought a practically efficient forward TFT algorithm.

In Section 3.1, we review the 2-way divide-and-conquer FFT algorithm presented in Section 2.4. Then, we slightly modify its presentation in order to better introduce the concept of truncated Fourier transform (TFT) in Section 3.2. From there, computing the forward TFT is deduced from the 2-way divide-and-conquer TFT algorithm in a very natural manner: we do this in Section 3.3. Unfortunately, and unlike FFT, the inverse map of TFT is very different from the forward process and, in fact, harder to understand in details. Sections 3.4, 3.5 and 3.6 attempt to deal with this challenge.

### 3.1 FFT: review and complement

Let  $\mathcal{R}$ ,  $n$ , and  $\omega$  be given as in Section 2.4. The DFT — with respect to  $\omega$  — of an  $n$ -tuple  $\mathbf{a} = (a_0, \dots, a_{n-1}) \in \mathbb{R}^n$  is the  $n$ -tuple  $\hat{\mathbf{a}} = (\hat{a}_0, \dots, \hat{a}_{n-1}) \in \mathbb{R}^n$  with

$$\hat{a}_i = \sum_{j=0}^{n-1} a_j \omega^{ij}.$$

The  $n$ -tuples can alternatively be represented as coefficients of polynomials in  $\mathbb{R}[x]$  and the FFT can be defined as the mapping from  $A = a_0 + a_1x + \dots + a_{n-1}x^{n-1}$  to the tuple  $(A(\omega^0), \dots, A(\omega^{n-1}))$ . Binary splitting is used to perform the FFT efficiently by evaluating only at  $\omega^{2^i}$  for  $i \in \{0, \dots, e-1\}$ , rather than all  $\omega^0, \dots, \omega^{n-1}$ . To compute the FFT of  $\mathbf{a}$  with respect to  $\omega$  we write

$$a_0, \dots, a_{n-1} = (b_0, c_0, \dots, b_{n/2-1}, c_{n/2-1})$$

and compute recursively the Fourier transform of  $(b_0, \dots, b_{n/2-1})$  and  $(c_0, \dots, c_{n/2-1})$  at  $\omega^2$ :

$$\begin{aligned} \text{FFT}_{\omega^2}(b_0, \dots, b_{n/2-1}) &= (\hat{b}_0, \dots, \hat{b}_{n/2-1}); \\ \text{FFT}_{\omega^2}(c_0, \dots, c_{n/2-1}) &= (\hat{c}_0, \dots, \hat{c}_{n/2-1}). \end{aligned}$$

Finally we construct  $\hat{\mathbf{a}}$  according to

$$\begin{aligned} \text{FFT}_{\omega}(a_0, \dots, a_{n-1}) &= (\hat{b}_0 + \hat{c}_0, \dots, \hat{b}_{n/2-1} + \hat{c}_{n/2-1}\omega^{n/2-1} \\ &\quad \hat{b}_0 - \hat{c}_0, \dots, \hat{b}_{n/2-1} - \hat{c}_{n/2-1}\omega^{n/2-1}). \end{aligned}$$

The equivalent polynomial interpretation divides  $A$  into even and odd parts, evaluates them at  $\omega^2$ , and then reconstructs to obtain  $\hat{A}$ . Although this can be implemented as a recursive algorithm, it is faster to use avoid the overhead of recursive stacks via an in-place algorithm.

The 2-way divide-and-conquer TFFT recalled above can be executed in-place. Let us explain how since this way of presenting Cooley-Tukey algorithm is a good introduction to TFFT. We need the following definition.

**Definition** We denote by  $[i]_e$  the bit wise reverse<sup>1</sup> of  $i$  at length  $e$ . Suppose  $i = i_02^0 + \dots + i_{e-1}2^{e-1}$  and  $j = j_02^0 + \dots + j_{e-1}2^{e-1}$  then

$$[i]_e = j \iff i_k = j_{e-k-1} \text{ for } k \in \{0, \dots, e-1\}.$$

**Example**  $[3]_5 = 24$  because  $3 = 00011_2$  whose reverse is  $11000_2 = 24$ .

$[11]_5 = 26$  because  $11 = 01011_2$  whose reverse is  $11010_2 = 26$ .

---

<sup>1</sup>In [26] the word "mirror" instead of reverse is used, which may lead to some ambiguity.

We begin at step zero with the vector

$$\mathbf{x}_0 = (x_{0,0}, \dots, x_{0,n-1}) = (a_0, \dots, a_{n-1})$$

and update this vector at step  $s \in \{1, \dots, e\}$  by the rule

$$\begin{bmatrix} x_{s,im_s+j} \\ x_{s,(i+1)m_s+j} \end{bmatrix} = \begin{bmatrix} 1 & \omega^{[i]_s m_s} \\ 1 & -\omega^{[i]_s m_s} \end{bmatrix} \begin{bmatrix} x_{s-1,im_s+j} \\ x_{s-1,(i+1)m_s+j} \end{bmatrix} \quad (3.1)$$

for all  $i \in \{0, 2, \dots, n/m_s - 2\}$  and  $j \in \{0, \dots, m_s - 1\}$ , where  $m_s = 2^{e-s}$ .

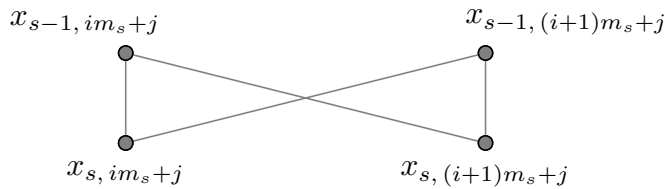


Figure 3.1: Butterfly.

Figure 3.1, known as a butterfly because of its form, is an illustration of Equation (3.1) as a relation among four values at steps  $s$  and  $s - 1$ . The butterfly's width is determined by  $m_s$ , which decreases as  $s$  increases. Note that two additions and *one* multiplication

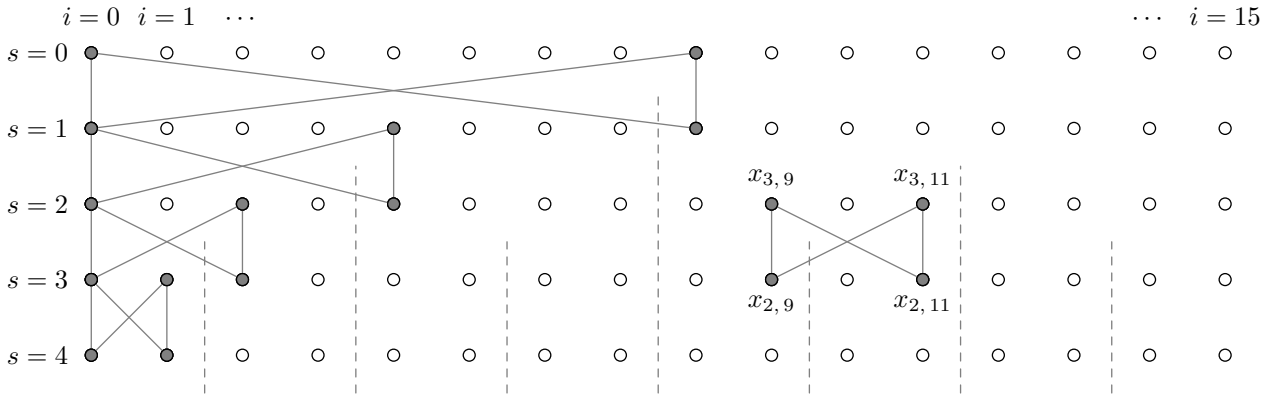


Figure 3.2: Butterflies. Schematic representation of Equation (3.1). The black dots correspond to the  $x_{s,i}$ . The top row corresponding to  $s = 0$ . In this case  $n = 16 = 2^4$ .

are done in Equation (3.1) as one product is merely the negation of the other. Using induction over  $s$ , it can be easily shown [26] that

$$x_{s,im_s+j} = (\text{DFT}_{\omega^{m_s}}(a_j, a_{m_s+j}, \dots, a_{n-m_s+j}))_{[i]_s},$$

for all  $i \in \{0, \dots, n/m_s - 1\}$  and  $j \in \{0, \dots, m_s - 1\}$ . In particular, when  $s = e$  and  $j = 0$  we have

$$\begin{aligned} x_{e,i} &= \hat{a}_{[i]_e} \\ \hat{a}_i &= x_{e,[i]_e} \end{aligned}$$

for all  $i \in \{0, \dots, n - 1\}$ . That is,  $\hat{\mathbf{a}}$  is a permutation of  $\mathbf{x}_e$  as illustrated in Figure 3.3

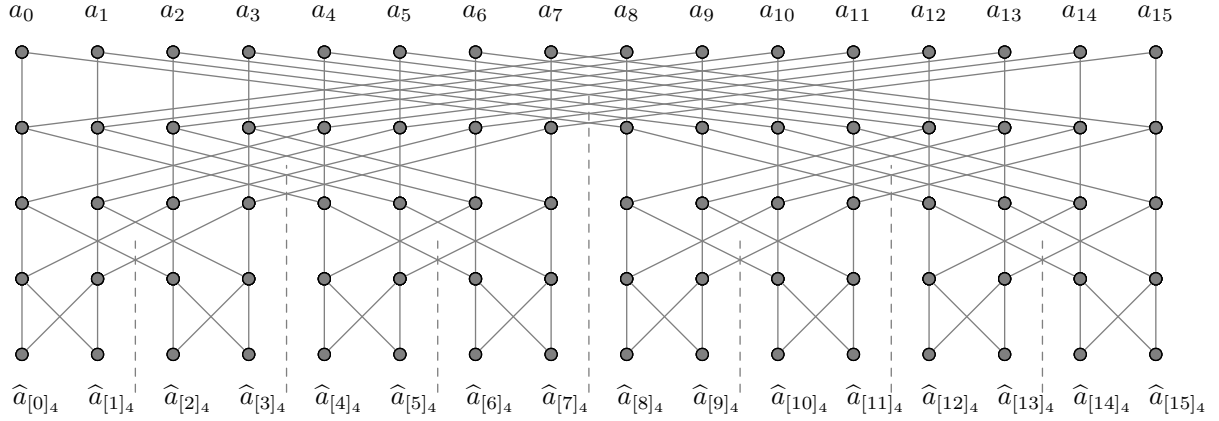


Figure 3.3: The Fast Fourier Transform for  $n = 16$ . The top row, corresponding to  $s = 0$ , represents the values of  $\mathbf{x}_0$ . The bottom row, corresponding to  $s = 4$  is some permutation of  $\hat{\mathbf{a}}$  (the result of the FFT on  $\mathbf{a}$ ).

One nice feature of the FFT is that it is straightforward to recover  $\mathbf{a}$  from  $\hat{\mathbf{a}}$

$$\text{DFT}_{\omega^{-1}}(\hat{\mathbf{a}})_i = \text{DFT}_{\omega^{-1}}(\text{DFT}_{\omega}(\mathbf{a}))_i = \sum_{k=0}^{n-1} \sum_{j=0}^{n-1} a_j \omega^{(i-k)j} = n a_i \tag{3.2}$$

since

$$\sum_{j=0}^{n-1} \omega^{(i-k)j} = 0$$

whenever  $i \neq k$ . This yields a polynomial multiplication algorithm of time complexity  $O(n \log n)$  in  $\mathbb{R}[x]$ .

### 3.2 The truncated Fourier transform

When the length of  $\mathbf{a}$  (the input) is not equal to a power of two, the  $\ell$ -tuple  $\mathbf{a} = (a_0, \dots, a_{\ell-1})$  is completed by setting  $a_i = 0$  when  $i \geq \ell$  to artificially extend the length of  $\mathbf{a}$  to the nearest power of two so that FFT can be performed.

As illustrated in Figures 3.4 and 3.5, FFT will calculate all of  $\hat{\mathbf{a}}$ , even if only  $\ell$  components of  $\hat{\mathbf{a}}$  are needed. These unnecessary computations occur when FFT is used to multiply polynomials, as the degree of the product is rarely a power of two.

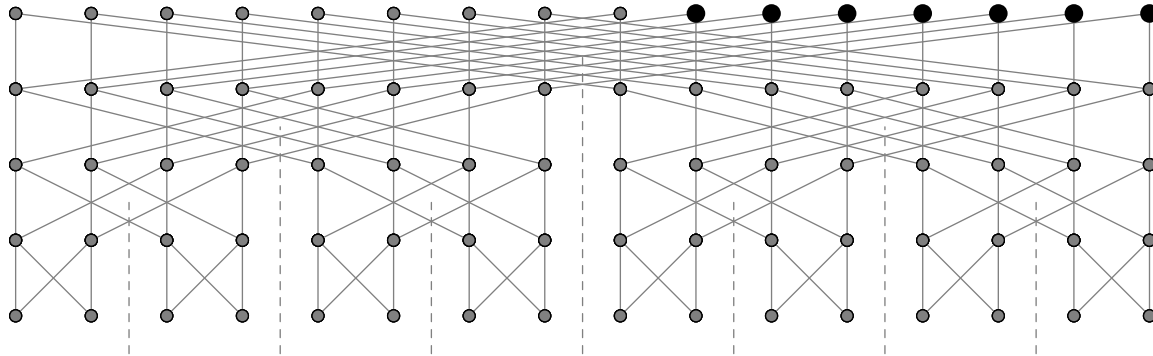


Figure 3.4: The FFT with “artificial” zero points (green).

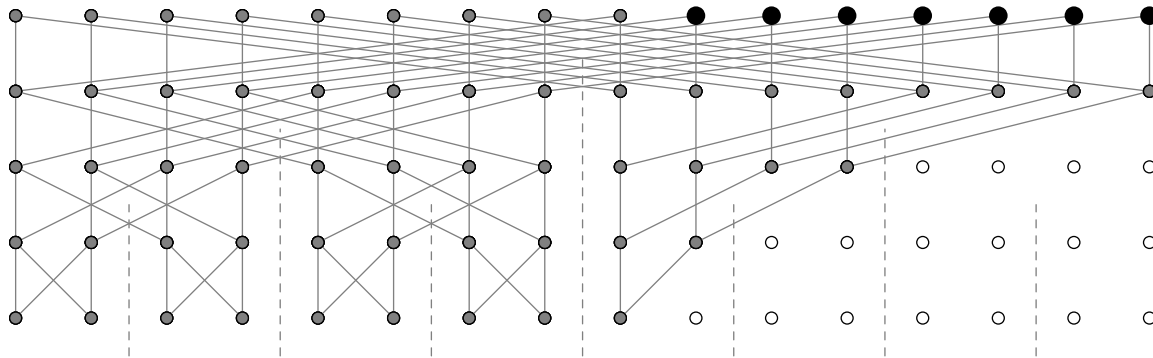


Figure 3.5: Removing all unnecessary computations from Figure 3.4 gives the schematic representation of the TFT.

With the exception that the lengths of the input and output vectors ( $\mathbf{a}$  resp.  $\hat{\mathbf{a}}$ ) are not necessarily powers of two, the TFT is similar to the FFT. More precisely the TFT of an  $\ell$ -tuple  $(a_0, \dots, a_{\ell-1}) \in \mathbb{R}^\ell$  is the  $\ell$ -tuple

$$(A(\omega_e^{[0]}), \dots, A(\omega_e^{[\ell-1]e})) \in \mathbb{R}^\ell.$$

where  $n = 2^e$ ,  $\ell < n$  (usually  $\ell \geq n/2$ ) and  $\omega$  a  $n$ -th root of unity.

**Remark** A more general description of the TFT, in which one can choose an initial vector  $(x_{0,i_0}, \dots, x_{0,i_n})$  and target vector  $(x_{e,j_0}, \dots, x_{e,j_n})$ , is given by van der Hoeven. The TFT may be performed by considering the full FFT and removing computations



unnecessary for the desired output if each  $i_k$  is distinct. However, as we are ignorant to a sufficiently fast method to find this sub graph, this discussion is restricted to the scenario in which the input and output are the same initial segments, as depicted in Figure 3.5.

The in-place algorithm in the previous section can be easily modified to perform the TFT. At stage  $s$  it suffices to compute

$$(x_{s,0}, \dots, x_{s,j}) \text{ with } j = (\lfloor (\ell - 1)/m_s \rfloor + 1)m_s - 1$$

where  $m_s = 2^{e-s}$ .<sup>2</sup>

### 3.3 Forward TFT: pseudo-code with an illustrative example

Denote  $X$  as a vector over  $\mathbb{Z}/p\mathbb{Z}$ ,  $\omega \in \mathbb{Z}/p\mathbb{Z}$  is a primitive  $n$ -th root of unity,  $p$  is a prime number,  $\ell$  is the length of  $X$ ,  $e := \min\{k \mid \ell \leq 2^k\}$  and  $n = 2^e$ . Initially, we pad the vector  $X$  (at its end) with zeroes (s. t. its size becomes  $n$ ) and call it  $X_0$ . The value of  $X$  at the end of the  $s$ -th iteration is denoted by  $X_s$  for  $1 \leq s \leq \log_2(n)$ . We write  $x_{s,i}$  for  $X_s[i]$  with  $0 \leq i \leq n - 1$ .

---

#### Algorithm 4: TFT( $X, \omega, p$ )

---

**Input:**  $X$  is the coefficient array of the input polynomial,  $\omega$  is a primitive  $n$ -th root of unity,  $p$  is a prime number

**Output:** Array  $X$ .

**for**  $s$  from 1 to  $\log_2 n$  **do**

$m_s = n/2^s$

**for**  $i$  from 0 by 2 to  $(n/m_s - 2)$  **do**

Let  $i_s$  be the bit wise reverse of  $i$  in the form of a decimal number

**for**  $j$  from 0 to  $(m_s - 1)$  **do**

**if**  $(i + 1)m_s + j < \lfloor \frac{\ell}{m_s} \rfloor m_s$  **then**

$\begin{bmatrix} x_{s,im_s+j} \\ x_{s,(i+1)m_s+j} \end{bmatrix} = \begin{bmatrix} 1 & \omega^{i_s m_s} \\ 1 & -\omega^{i_s m_s} \end{bmatrix} \begin{bmatrix} x_{s-1,im_s+j} \\ x_{s-1,(i+1)m_s+j} \end{bmatrix}$

---

The following is an example of serial forward TFT w.r.t. prime number is 17,  $\omega$  is 3,  $\ell$  is 9 and  $n$  is 16 which is defined before. The initial input is an vector  $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ .

<sup>2</sup>This is a correction to the bound given in [12].

The size of input  $\ell$  is 9 and the smallest number which larger than  $\ell$  and satisfied some power of two is 16. Totally, we need  $\log_2 n = \log_2 16 = 4$  steps to achieve the final output. With zero padding in the end of input to make it a vector  $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 0, 0, 0, 0, 0, 0\}$  as showing in figure 3.5(b). Using equation 3.1, the second line can be calculated from the original input which is an vector  $\{10, 2, 3, 4, 5, 6, 7, 8, 9, 2, 3, 4, 5, 6, 7, 8\}$ .

Applied equation 3.1 to calculate the third line from the second line as before. As showing in the algorithm 2, only  $\lceil \frac{\ell}{m_s} \rceil m_s$  items are calculated which is 12 in this step. After calculation, the output is vector  $\{15, 8, 10, 12, 5, 13, 13, 13, 6, 12, 9, 6\}$ . Keep applying equation 3.1 to the last two steps and the final output is vector  $\{11, 5, 4, 6, 10, 15, 12, 0, 13\}$ .

### 3.4 The inverse truncated Fourier transform

Unlike the FFT, the TFT cannot be inverted simply by performing another TFT with  $1/\omega$  and adjusting by a constant factor. There is information missing that must be taken into account.

**Example** Let  $\mathbb{R} = \mathbb{Z}/17\mathbb{Z}$ ,  $n = 2^2 = 4$ , with  $\omega = 4$  a  $n$ -th primitive root of unity. The TFT of  $\mathbf{a} = (a_0, a_1, a_2)$  is

$$\begin{bmatrix} A(\omega^0) \\ A(\omega^2) \\ A(\omega^1) \end{bmatrix} = \begin{bmatrix} A(1) \\ A(-1) \\ A(3) \end{bmatrix} = \begin{bmatrix} a_0 + a_1 + a_2 \\ a_0 - a_1 + a_2 \\ a_0 + 3a_1 + 9a_2 \end{bmatrix}$$

Now to show that the TFT of this w.r.t.  $1/\omega$  is *not*  $\mathbf{a}$  define

$$\mathbf{b} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} a_0 + a_1 + a_2 \\ a_0 - a_1 + a_2 \\ a_0 + 3a_1 + 9a_2 \end{bmatrix}$$

The TFT of  $\mathbf{b}$  w.r.t  $1/\omega = -4$  is

$$\begin{bmatrix} B(\omega^0) \\ B(\omega^{-2}) \\ B(\omega^{-1}) \end{bmatrix} = \begin{bmatrix} B(1) \\ B(-1) \\ B(-4) \end{bmatrix} = \begin{bmatrix} b_0 + b_1 + b_2 \\ b_0 - b_1 + b_2 \\ b_0 - 4b_1 - b_2 \end{bmatrix} = \begin{bmatrix} 3a_0 + 3a_1 + 11a_2 \\ a_0 + 5a_1 + 9a_2 \\ -4a_0 + 2a_1 + 5a_2 \end{bmatrix}$$

which is not some constant multiple of  $\text{TFT}_\omega(\mathbf{a})$ . The completion of  $\mathbf{b}$  to  $(b_0, b_1, b_2, 0)$  results in this discrepancy. Instead, to match the FFT of  $\mathbf{a}$  w.r.t  $\omega$ ,  $b$  should be completed to  $(b_0, b_1, b_2, a_0 - 3a_1 + 9a_2)$ .

We follow the paths from  $\mathbf{x}_e$  back to  $\mathbf{x}_0$  to invert the TFT. If one value of

$$x_{s,im_s+j}, x_{s-1,im_s+j}$$

and one value of

$$x_{s,(i+1)m_s+j}, x_{s-1,(i+1)m_s+j}$$

are known, the other values may be found. In other words, if two values of a butterfly are known, then Equation (3.1) can be used to find the other two values, as the relevant matrix can be inverted. Also, this situation is ideal for implementation, as these relations only involve shifting (multiplication and division by two), additions, subtractions and multiplications by roots of unity.

Given  $x_{e,0}, \dots, x_{e,2^k-1}, x_{e-k,0}, \dots, x_{e-k,2^k-1}$  can be found. As depicted in Figure 3.7, no butterfly relations necessary to move up in this manner require  $x_{s,2^k+j}$  for any  $s \in \{e-k, \dots, e\}, j > 0$ . In general,

$$x_{e,2^j+2^k}, \dots, x_{e,2^j+2^k-1}$$

is enough to calculate

$$x_{e-k,2^j}, \dots, x_{e-k,2^j+2^k-1}$$

for  $0 < k \leq j < e$ .

### 3.5 Inverse TFT: an algorithm

For our restricted case (all padding zeroes packed at the end), a simple recursive description of the inverse TFT algorithm is presented. The algorithm operates in a length  $n$  array  $\mathbf{x} = (\mathbf{x}_0, \dots, \mathbf{x}_{n-1})$  for which we assume access; here  $n = 2^e$  corresponds to  $\omega$ , a  $n$ th primitive root of unity.

Initially, the content of the array is

$$\mathbf{x} := (x_{e,0}, \dots, x_{e,\ell-1}, 0, \dots, 0)$$

where  $(x_{e,0}, \dots, x_{e,\ell-1})$  is the result of the TFT on  $(x_{0,0}, \dots, x_{0,\ell-1}, 0, \dots, 0)$ .

As in our illustrations, we use pictures, like Figure 3.7, to indicate which values are known (solid dots  $\bullet$ ) and which are to be calculated (empty dot  $\circ$ ). For instance, “push down  $\tilde{x}_k$  with Figure 3.7, represents: use  $\mathbf{x}_k = x_{s-1,im_s+j}$  and  $\mathbf{x}_{k+m_s+j} = x_{s-1,(i+1)m_s+j}$  to

find  $x_{s, im_s+j}$ . An arrow emphasizes that this new value should also overwrite the one at  $\mathbf{x}_k$ . With the caveat that  $i$  and  $j$  are not explicitly known values, this calculation is easily accomplished using (3.1). As  $s$  is known, so are  $m_s$  and an array position  $k$ . Note  $i$  is recovered by  $i = k \text{ quo } m_s$ , the quotient of  $k/m_s$ .

A full description of the inverse TFT follows in Algorithm 5. Note that the initial call is **InvTFT**(0,  $\ell - 1$ ,  $n - 1$ , 1). Figure 3.10 illustrates Algorithm 5.

---

**Algorithm 5:** InvTFT( $\mathbf{x}$ , head, tail, last,  $s$ )

---

**Input:**  $\mathbf{x}$  is an input array, head, tail, last is the indexing of the input array,  
 $s = \log_2 \text{last}$ .

**Output:** Array  $\mathbf{x} = [x_0, x_1, \dots, x_\ell]$ .

middle  $\leftarrow \frac{\text{last} - \text{head}}{2} + \text{head}$ ;

LeftMiddle  $\leftarrow \lfloor \text{middle} \rfloor$ ;

RightMiddle  $\leftarrow \text{LeftMiddle} + 1$ ;


**if** head > tail **then**

    Base case—do nothing;

**return** null;

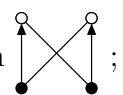
**else if** tail  $\geq$  LeftMiddle **then**

    Push up the self-contained region  $\mathbf{x}_{\text{head}}$  to  $\mathbf{x}_{\text{LeftMiddle}}$ ;


    Push down  $\mathbf{x}_{\text{tail}+1}$  to  $\mathbf{x}_{\text{last}}$  with  ;

    InvTFT( $\mathbf{x}$ , RightMiddle, tail, last,  $s + 1$ );


$s \leftarrow e - \log_2(\text{LeftMiddle} - \text{head} + 1)$ ;

    Push up (in pairs)  $(\mathbf{x}_{\text{head}}, \mathbf{x}_{\text{head}+m_s})$  to  $(\mathbf{x}_{\text{LeftMiddle}}, \mathbf{x}_{\text{LeftMiddle}+m_s})$  with  ;

**else if** tail < LeftMiddle **then**

    Push down  $\mathbf{x}_{\text{tail}+1}$  to  $\mathbf{x}_{\text{LeftMiddle}}$  with  ;

    InvTFT( $\mathbf{x}$ , head, tail, LeftMiddle,  $s + 1$ );

    Push up  $\mathbf{x}_{\text{head}}$  to  $\mathbf{x}_{\text{LeftMiddle}}$  with  ;

---

### 3.6 Illustration of the inverse TFT algorithm

Figure 3.11 and Figure 3.12 show an example of the inverse TFT algorithm w.r.t prime number is 17,  $\omega$  is 3,  $\ell$  is 11 and  $n$  is 16. The initial input is an vector  $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$  in the first step and zero padding in the final step. The algorithm operates on a length- $n$  array  $X = (X[0], \dots, X[n-1])$  with  $n = 2^e$ ,  $1 \leq \ell < n$  (the last  $n - \ell$  coefficients being zeroes) and  $\omega$  is an  $n$ -th primitive root of unity). Thus, initially, the contents of the array are

$$X = (x_{e,0}, \dots, x_{e,\ell-1}, 0, \dots, 0)$$

where  $(x_{e,0}, \dots, x_{e,\ell-1})$  is the result of the TFT on  $(x_{0,0}, \dots, x_{0,\ell-1}, 0, \dots, 0)$ , ultimately, the output of the computation.

According to Algorithm 4,  $\text{tail} \geq \text{LeftMiddle}$  ( $\text{tail} = 10$ ,  $\text{LeftMiddle} = 7$ ) self-contained push up is used to calculate  $x_{1,0}, \dots, x_{1,7}$  from  $x_{4,0}, \dots, x_{4,7}$  using Equation 3.1. Then push down  $x_{\text{tail}+1}$  to  $x_{\text{last}}$  which is  $x_{4,11}$  to  $x_{4,15}$  here. After calculation,  $x_{4,11}$  to  $x_{4,15}$  is equal to  $\{14, 15, 7, 10, 8\}$ .

Recursive call on right half.  $\text{Tail} \leq \text{LeftMiddle}$  ( $\text{tail} = 2$ ,  $\text{LeftMiddle} = 3$ ), push down to calculate  $x_{3,11} = 16$ . Recursive call on left half.  $\text{Tail} \geq \text{LeftMiddle}$  ( $\text{tail} = 2$ ,  $\text{LeftMiddle} = 1$ ), push up the contained (dashed) region then push down. After calculation,  $x_{2,8} = 1$ ,  $x_{2,9} = 14$  and  $x_{2,11} = 15$ . Recursive call on right half to obtain  $x_{2,10} = 15$ .

Following Algorithm 5, do pushing up to calculate  $x_{3,8}$  to  $x_{3,11}$  which is  $\{11, 6, 14, 16\}$ . Self push up to calculate  $x_{4,8}$  to  $x_{4,11}$  which is  $\{3, 0, 3, 14\}$ .

For now all items in the second line are available. According to Equation 3.1, the final step can be achieved by pushing up which is  $\{8, 15, 13, 14, 15, 7, 10, 8, 5, 15, 10\}$  here.

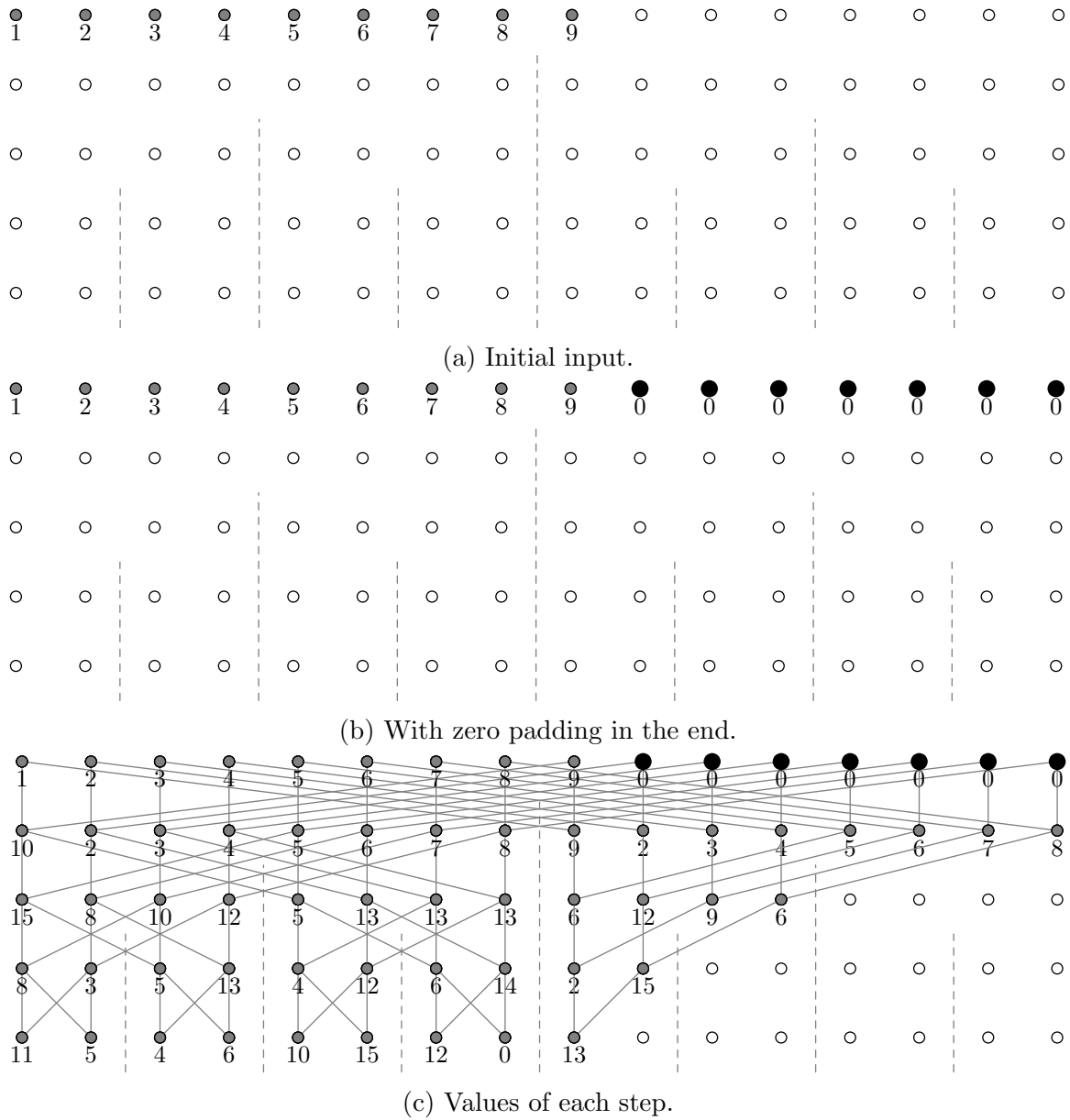


Figure 3.6: Example of TFT where  $n = 16, \ell = 9$ , prime number is 17, and  $\omega = 3$ .

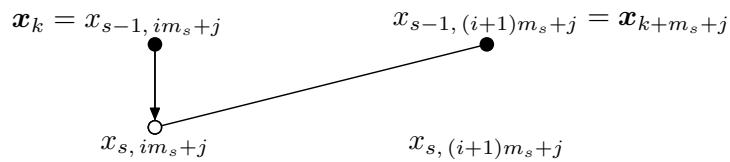
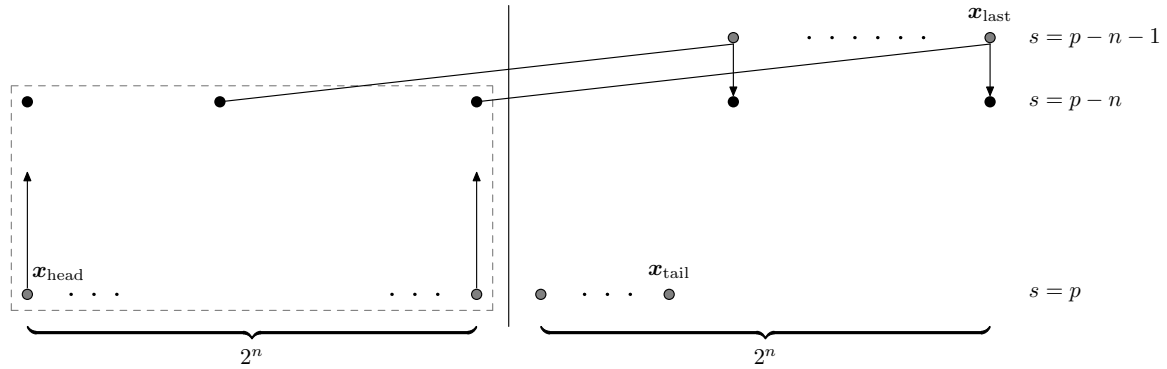
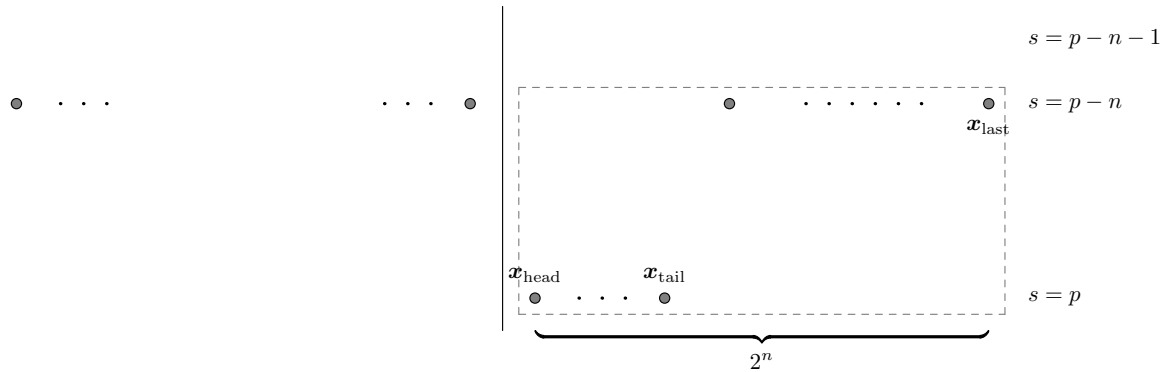


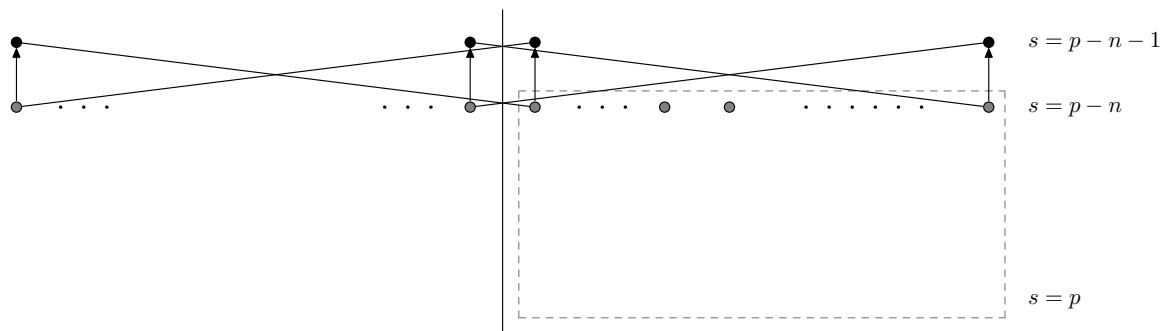
Figure 3.7: The relation for no butterfly.



(a) Line (8): push up the self contained (dashed) region. This yields values sufficient to push down at line (9).

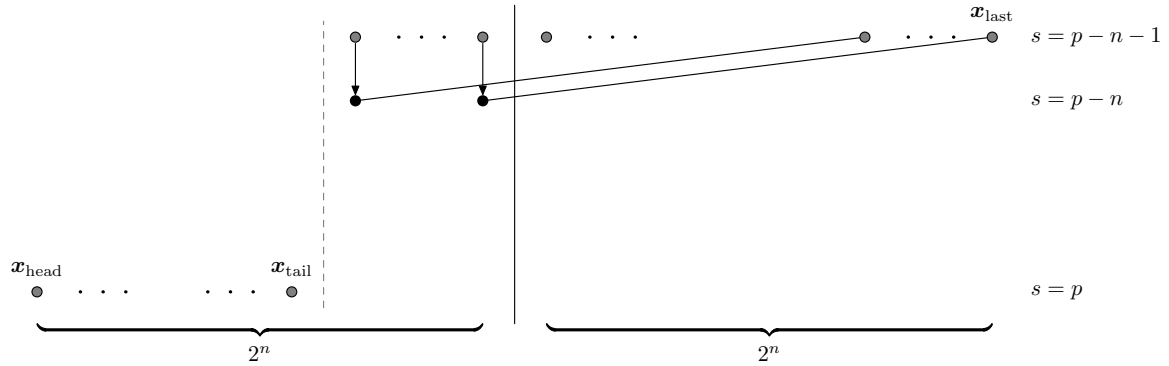


(b) This enables us to make a recursive call on the dashed region (line (12)). By our induction hypothesis this brings all points at  $s = p$  to  $s = p - n$ .

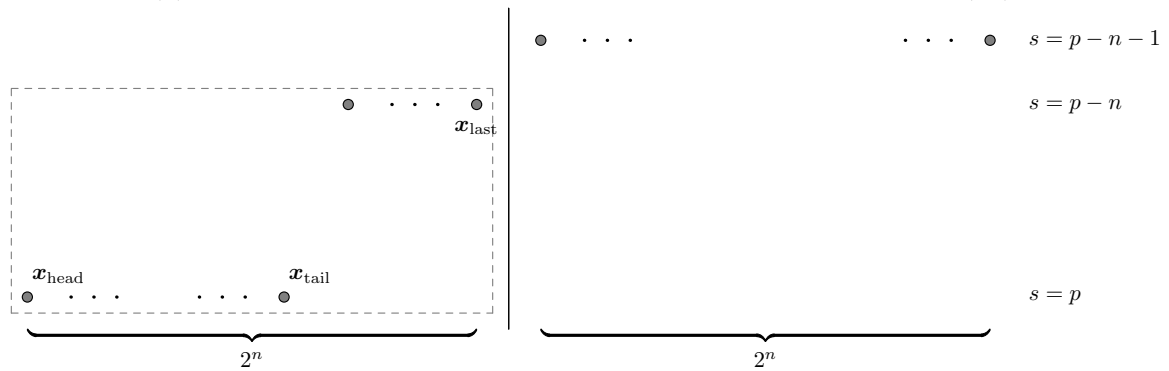


(c) Sufficient points at  $s = p - n$  are known to move to  $s = p - n - 1$  at line (13).

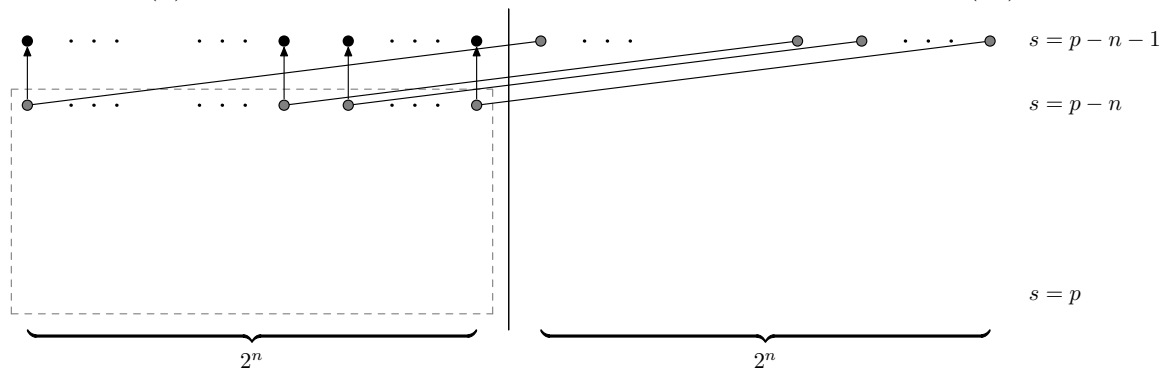
Figure 3.8:  $\text{tail} \geq \text{LeftMiddle}$  (i.e. at least half the values are at  $x = p$ ).



(a) Initially there is sufficient information to push down at line (14).



(b) This enables us to make the prescribed recursive call at line (15).



(c) By the induction hypothesis this brings the values in the dashed region to  $s = p - n$ , leaving enough information to move up at line (16).

Figure 3.9:  $\text{tail} < \text{LeftMiddle}$  (i.e. less than half the values are at  $x = p$ ).



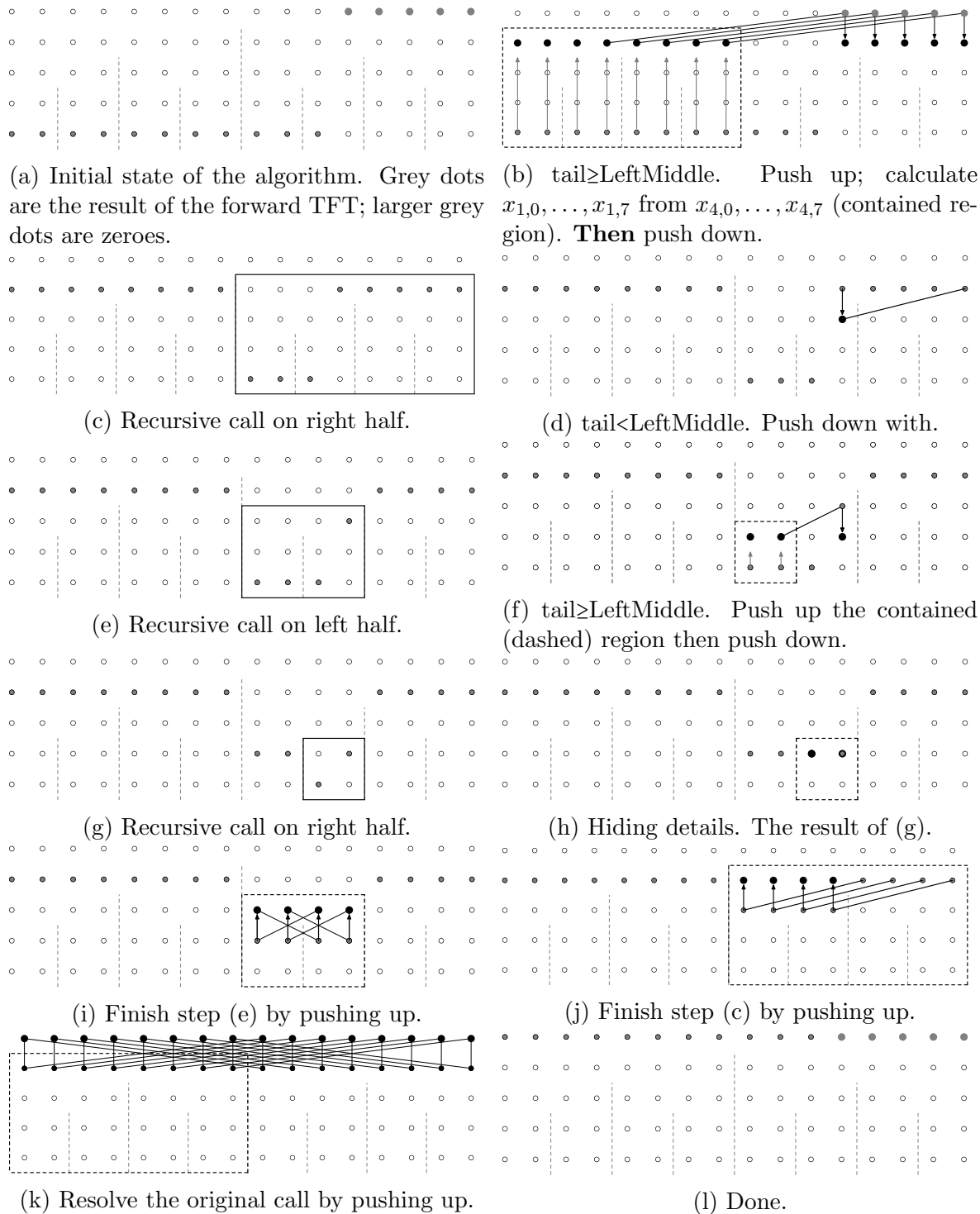
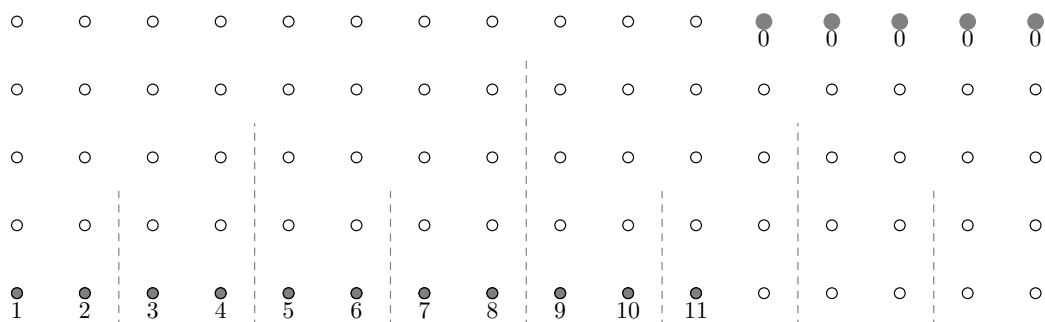
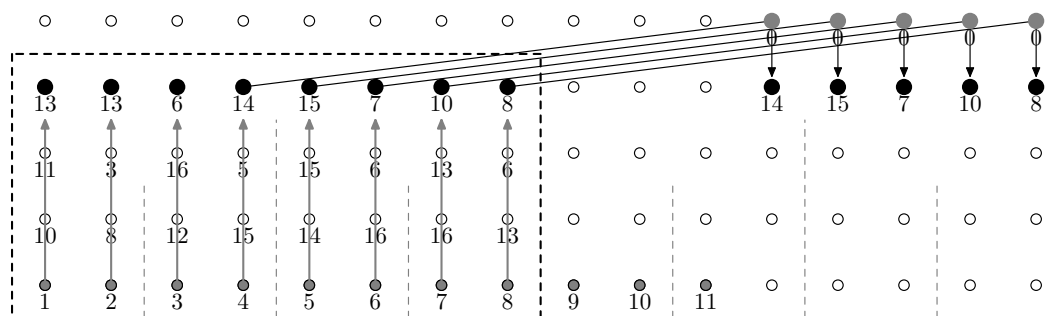


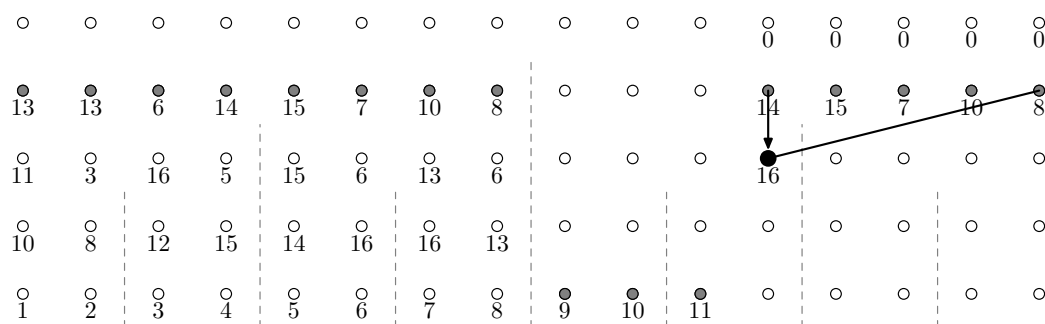
Figure 3.10: Schematic representation of the recursive computation of the Inverse TFT for  $n = 16$  and  $\ell = 11$ .



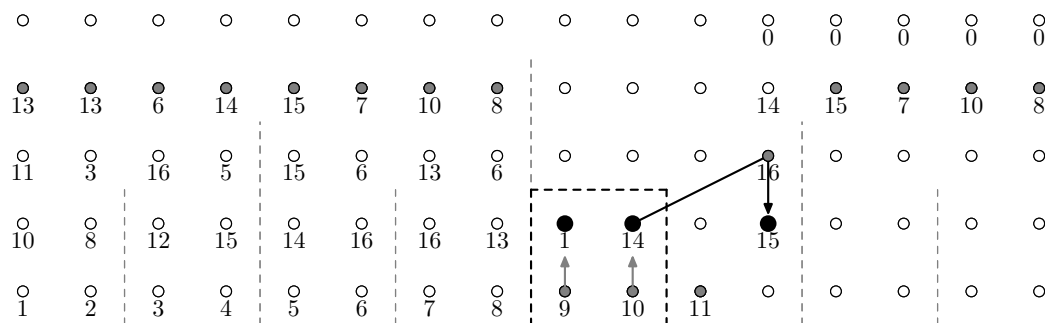
(a) Initial state of the algorithm with zero padding in the end.



(b)  $\text{tail} \geq \text{LeftMiddle}$ . Push up. calculate  $x_{1,0}, \dots, x_{1,7}$  from  $x_{4,0}, \dots, x_{4,7}$  (contained region).

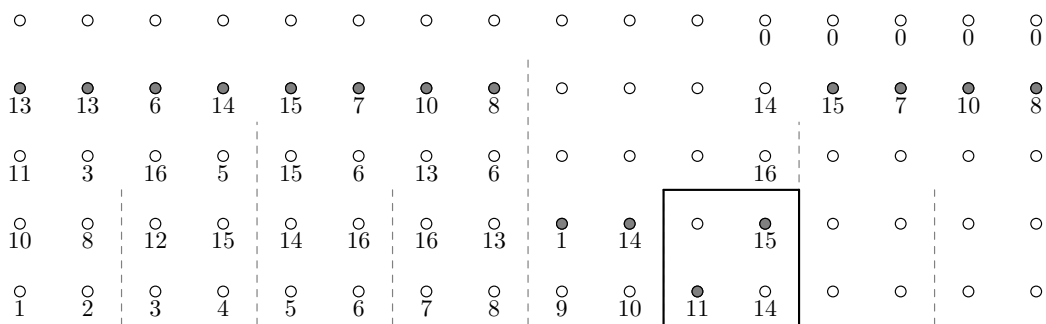


(c)  $\text{tail} < \text{LeftMiddle}$ . Push down.

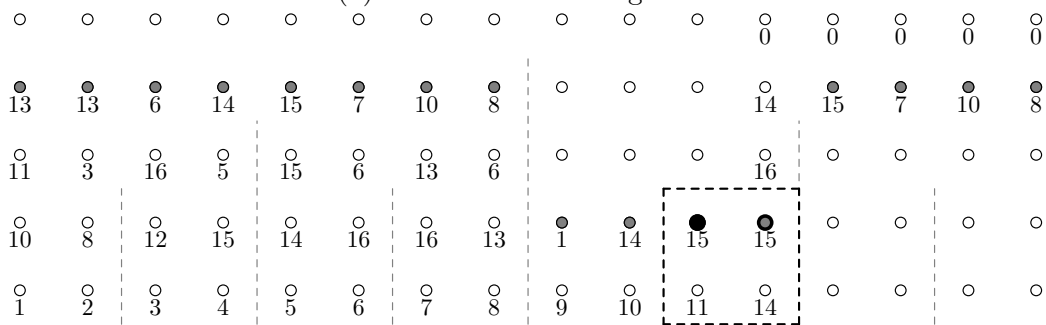


(d)  $\text{tail} \geq \text{LeftMiddle}$ . Push up the contained (dashed) region then push down.

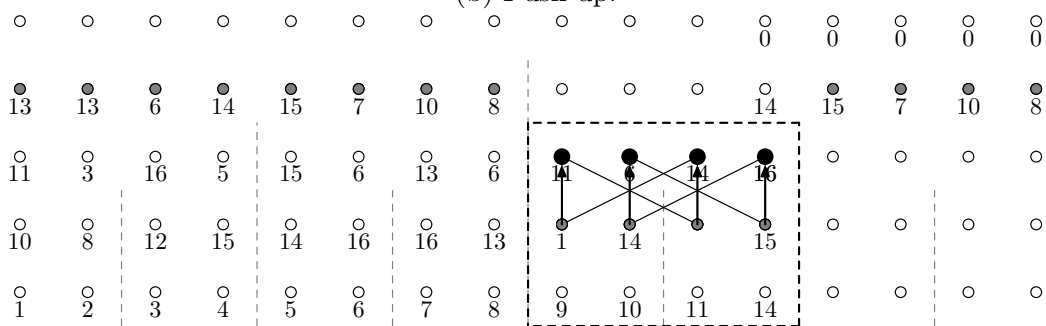
Figure 3.11: The first part of ITFT example.



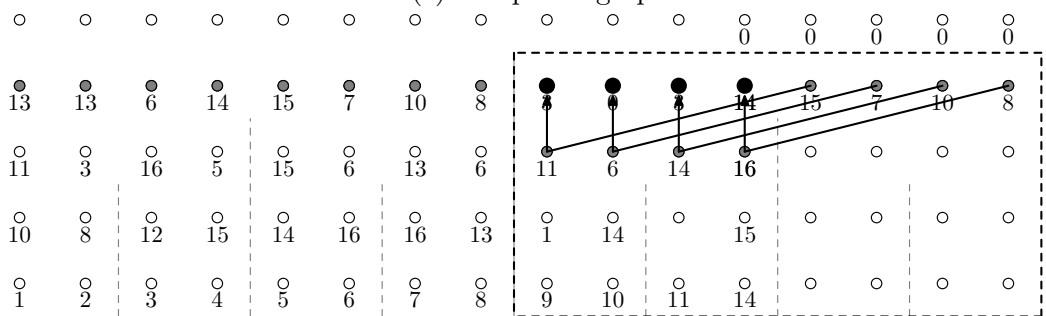
(a) Recursive call on right half.



(b) Push up.



(c) Self pushing up.



(d) Self pushing up.

Figure 3.12: The second part of ITFT example.

# Chapter 4

## The Relaxed General Radix TFT and Strict General Radix Inverse TFT

In Chapter 3, we reviewed the original algorithms for the forward and inverse TFT introduced by J. van der Hoeven. We turn our attention now to the variation of D. Harvey as well as that of J. Johnson and L.C. Meng. Both variations are based on Cooley-Tukey like formula. The former is called *strict general radix* as it strictly follows the specifications proposed by J. van der Hoeven, while the latter is called *relaxed general radix* as it requires some zero padding so as to improve data flow which supports full vectorization and parallelization.

Using general radix (instead of radix 2) and Cooley-Tukey like formula, instead of a naive 2-way divide-and-conquer algorithm, can be interpreted as using a blocking strategy. The reader should remember from Sections 2.14 and 2.15 that going from the latter to the former with Cooley-Tukey FFT algorithm yields optimal cache complexity. In addition, those TFT algorithm are cache-oblivious, that is, they automatically adapt to the memory hierarchy. Last but not least, serial algorithm with optimal cache optimality are likely to produce parallel algorithms with good (or minimal) communication costs.

### 4.1 Introduction

The discrete Fourier transform (DFT) is an important computation in scientific computing, and many applications require a high-performance implementation of DFT. Computational complexity can be reduced from  $O(n^2)$  to  $O(n \log n)$  using Fast Fourier Transform

(FFT) algorithms, usually by obtaining smaller transforms via recursive factorization of a large transform. FFTs are used for fast polynomial and integer arithmetic and modular methods in computer algebra.

For polynomial and integer multiplications, the convolution theorem is used, in which two forward FFTs and one inverse FFT are required, and the input size is arbitrary. Inputs are typically padded to the smallest power of two that is larger than the output, as many FFT implementations require the inputs to have a size that is a power of two. This results in a staircase phenomenon, in which the computing time for sizes between powers of two is virtually equal to the time for the larger power of two FFTs, leading to FFT computations in which some inputs are zero and not all outputs are needed.

Because of this situation, the pruned or truncated DFT was developed. Van der Hoeven presented a radix-2 algorithm for TFFT, as well as an Inverse Truncated Fourier Transform (ITFFT). Although the TFFT reduces the operation count and smooths out the staircase phenomenon, it must be optimized to have better performance than a highly-tuned power-of-two FFT with padding.

By decomposing the problem in a cache-friendly approach, David Harvey reported improved performance of the TFFT. In this chapter, we implement a general-radix TFFT algorithm expressed in the  $\Sigma$ -SPL formalism used by SPIRAL. A small relaxation is introduced while using more arithmetic operations to improve the data flow and allow automatic vectorization and parallelization, resulting in improved performance.

## 4.2 A relaxed general-radix TFFT algorithm

Like the DFT, a general-radix TFFT algorithm recursively breaks down a transform into smaller ones. In this section, we introduce an algorithm that allows for full vectorization and improved parallelism by using a small relaxation that slightly increases arithmetic cost for improved data flow.

Denote a truncated Fourier transform as  $TFFT_{n,\ell,m}$ , where  $n = rs$  is the size of the transform,  $1 \leq \ell \leq n$  is the size of the input (assuming  $x_\ell = \dots = x_{n-1} = 0$ ), and  $0 \leq m \leq n$  is the size of the truncated output. Let  $n = 2^e$ , for some positive integer  $e$ .

Define  $\ell_s = \lceil \ell/r \rceil$ ,  $m_s = \lceil m/r \rceil$ ,  $\ell_r = \min(\ell, r)$ ,  $m_r = \min(m, r)$ ,  $TFFT_{\bar{r}} = TFFT_{r,\ell_r,m_r}$  and

$\text{TFT}_{\bar{c}} = \text{TFT}_{s,\ell_s,m_s}$ , then the relaxed Cooley-Tukey algorithm for TFT is:

$$\text{TFT}_{n,\ell,m} = (\mathbf{I}_{m_s} \otimes \text{TFT}_{\bar{r}}) \cdot T_{r,s}^n \cdot (\text{TFT}_{\bar{c}} \otimes \mathbf{I}_{\ell_r}) \quad (4.1)$$

During the recursive factorization, the relaxed TFT algorithm does not perform explicit permutation. The transformed values are thus permuted based on the specific factorization path. The permuted values can be recovered if the inverse transform uses a symmetric factorization path that cancels the permutations of the forward transform, as the TFT is used mainly for high-performance implementation of convolution algorithms.

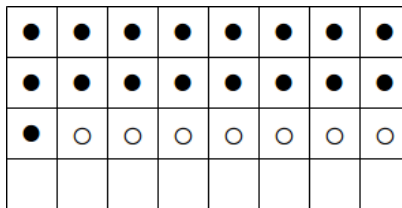


Figure 4.1: An example of factoring  $\text{TFT}_{32,17,17}$  with the relaxed general-radix TFT algorithm.

Figure 4.1 shows an example of the relaxation in one recursive step. The relaxation is applied to the columns 1 to 7, therefore obtaining uniform column transforms. The row transforms become uniform as well. The right tensor product in equation 2.1 containing  $\text{TFT}_{\bar{c}}$  and the twiddle matrix are applied to the relaxed columns. The left tensor product containing  $\text{TFT}_{\bar{r}}$  is applied to the relaxed rows. The relaxation is repeated in the recursive application of equation 2.1, until fixed-size base cases are reached.

### 4.3 A cache-friendly inverse TFT (ITFT)

The ITFT cannot be implemented by simply running the TFT in reverse, because when the ITFT commences there is insufficient information to perform all the row transforms. To circumvent this difficulty, we proceed as follows. We first perform as many row transforms as possible. We are then able to perform some of the column transforms. When these are complete, it becomes possible to execute the last row transform that was inaccessible before. After this row transform, the remainder of the column transforms may be completed. The following algorithm gives a precise statement.

---

**Algorithm 6:** CACHEFRIENDLYITFT( $L, \zeta, z, n, f; (x_0, \dots, x_{L-1})$ )

---

**Input:**  $L$  is the size of input vector,  $f$  is 0 initially.  $x$  is input vector.

 $L = 2^\ell \geq 2, \zeta \in R^x, f \in \{0, 1\}, 1 \leq n + f \leq L, 1 \leq z \leq L, z \leq n, x_i = \hat{a}_i$  for

 $0 \leq i < n, x_i = La_i$  for  $n \leq i < z$ .

**Output:**  $x_i = La_i$  for  $0 \leq i < n, x_n = \hat{a}_n$  if  $f = 1$ .

**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**  $f = 0$  **then**

     $x_0 \leftarrow (z == 1) ? 2x_0 : 2x_0 - x_1$ 

  **if**  $n = 0$  **then**

     $x_0 \leftarrow (z == 1) ? x_0/2 : (x_0 + x_1)/2$ 

  **return;**

/\* recursive case

\*/

 $L_1 \leftarrow 2^{\lfloor \ell/2 \rfloor}, L_2 \leftarrow 2^{\lceil \ell/2 \rceil}, n_2 \leftarrow n \bmod L_2, n_1 \leftarrow \lfloor n/L_2 \rfloor, z_2 \leftarrow z \bmod L_2, z_1 \leftarrow \lfloor z/L_2 \rfloor;$ 
**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**  $u = 0$  **to**  $n_1 - 1$  **do**

  CACHEFRIENDLYITFT( $L_2, \zeta^{L_1}, L_2, L_2, 0; r_u$ );

/\* rightmost column transforms

\*/

**for**  $u = n_2$  **to**  $m' - 1$  **do**

  CACHEFRIENDLYITFT( $L_1, \omega_L^u \zeta, z_1 + 1, n_1, f'; c_u$ );

**for**  $u = m'$  **to**  $z'_2 - 1$  **do**

  CACHEFRIENDLYITFT( $L_1, \omega_L^u \zeta, z_1, n_1, f'; c_u$ );

/\* last row transform

\*/

**if**  $f' = 1$  **then**

  CACHEFRIENDLYITFT( $L_2, \zeta^{L_1}, z'_2, n_2, f; r_{n_1}$ );

/\* leftmost column transforms

\*/

**for**  $u = 0$  **to**  $m - 1$  **do**

  CACHEFRIENDLYITFT( $L_1, \omega_L^u \zeta, z_1 + 1, n_1 + 1, 0; c_u$ );

**for**  $u = m$  **to**  $n_2 - 1$  **do**

  CACHEFRIENDLYITFT( $L_1, \omega_L^u \zeta, z_1, n_1 + 1, 0; c_u$ );

---

# Chapter 5

## Python Code Generator for TFT and Inverse TFT in C++/CilkPlus

We have implemented an automatic code generator for the implementation of forward TFT 4.2 and inverse TFT (ITFT) 4.3 algorithms. Both generated programs are (1) supported to be compiled in both serial and parallel mode, and (2) valid for any arbitrary input size. In order to generate efficient parallel code, optimized techniques, such as Montgomery tricks and unrolling loops are taken into account and integrated into the code generator.

This code generator is written in Python and described in Section 5.1. Our implementation framework in Section 5.2 is based on the Basic Polynomial Algebra Subprograms (BPAS) library [4]. In Section 5.3, we describe the architecture of our code generator and the specifications of the main methods in the code. Finally, in Section 5.4, optimization techniques and code generation in both serial and parallel mode are discussed.

### 5.1 C++ code generation in Python

This section discusses the use of Python as a tool for code generation. Python is a widely used general-purpose programming language <sup>1</sup>. It provides high-level features that emphasize code readability and allow programmers to express concepts in fewer lines of code than would be possible in languages such as C++, or Java. The automatic generation of code using Python is intended to be adaptive to different architectures, for which, for instance, the cache size or the number of registers varies.

A simple example is given below to demonstrate the use of Python to generate C++

---

<sup>1</sup>[https://en.wikipedia.org/wiki/Python\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Python_(programming_language))



code <sup>2</sup>:

```
from CodeGenerator import *
cpp = CppFile("tft_test.cpp")
cpp("#include <iostream>")
with cpp.block("void main()"):
    for i in range(6):
        cpp('std::cout << "' + str(i) + '" << std::endl;')
cpp.close()
```

such that we have the following output:

```
#include <iostream>
void main()
{
    std::cout << "0" << std::endl;
    std::cout << "1" << std::endl;
    std::cout << "2" << std::endl;
    std::cout << "3" << std::endl;
    std::cout << "4" << std::endl;
    std::cout << "5" << std::endl;
}
```

Using Python’s “with” keyword, statements can be encapsulated in {} blocks that are closed out automatically with the correct indentation so that the output remains readable. When generating more sophisticated code, the Python script becomes less readable because of numerous unseemly string concatenations. This issue can be addressed through the “subs” method as follows.

```
from CodeGenerator import *
cpp = CppFile("tft_test.cpp")
cpp("#include <iostream>")
with cpp.block("void main()"):
    for i in range(6):
        with cpp.subs(i=str(i), xi="x"+str(i+1)):
            cpp('int $xi$ = $i$;')
cpp.close()
```

---

<sup>2</sup><http://www.codeproject.com/Articles/571645/Really-simple-Cplusplus-code-generation-in-Python>

The substitutions are valid within the Python “with” block, which can be nested, producing:

```
#include <iostream>
void main()
{
    int x1 = 0;
    int x2 = 1;
    int x3 = 2;
    int x4 = 3;
    int x5 = 4;
    int x6 = 5;
}
```

## 5.2 The basic polynomial algebra subprograms

The serial and parallel TFT and ITFT algorithms are implemented based on the BPAS library [4]. The BPAS provides arithmetic operations such as multiplication, division and root isolation for uni-variate and multivariate polynomials over prime fields or with integer coefficients. The code is mainly written in CilkPlus [16] targeting multicore processors. The current distribution focuses on dense polynomials, while the sparse case is a work in progress.

Since the library supports a wide variety of situations in terms of problem sizes and available computing resources, our emphasis is on adaptive algorithms. One of the purposes of the BPAS project is to take advantage of hardware accelerators in the development of polynomial systems solvers. The BPAS library source code is publicly available at [www.bpaslib.org](http://www.bpaslib.org).

### 5.2.1 Design and specification

The BPAS functionalities are organized into three levels. Level 1 comprises basic arithmetic operations that are specific to a polynomial representation or coefficient ring. Examples of Level-1 operations are multi-dimensional FFTs/TFTs and uni-variate real root isolation. At Level 2, arithmetic operations are implemented for all types of coefficient rings supported by BPAS such as prime fields, ring of integers, field of rational numbers. Level 3 includes advanced arithmetic operations taking as input a zero-dimensional regular chain, e.g. the normal form of a polynomial and multivariate real root isolation.

### 5.2.2 User interface

The **BPAS** library makes use of type constructors to provide generic structures. For instance, `SparseUnivariatePolynomial` (**SUP**) can be instantiated over any **BPAS** ring. On the other hand, for efficiency considerations, certain polynomial type constructors, such as `DistributedDenseMultivariateModularPolynomial` (**DDMMP**), are only available over finite fields. This ensures that the data encoding a **DDMMP** polynomial consists only of consecutive memory cells.

For the same efficiency considerations, the most frequently used polynomial rings, such as `DenseUnivariateIntegerPolynomial` (**DUZP**) and `DenseUnivariateRationalNumberPolynomial` (**DUQP**), are primitive types. Consequently, **DUZP** and `SUP<Integer>` implement the same functionalities; however, the implementation of the former is further optimized.

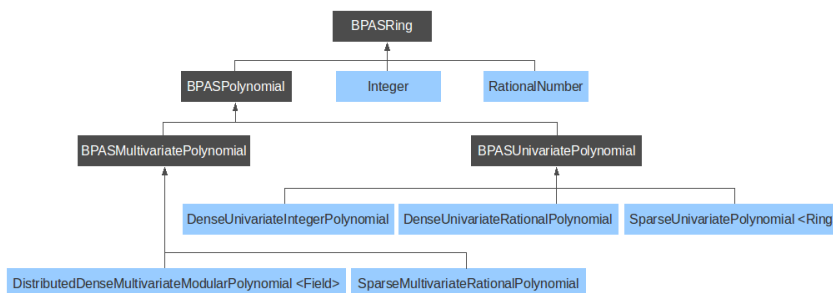


Figure 5.1: A snapshot of **BPAS** algebraic data structures.

Figure 5.1 [4] shows a subset of **BPAS**'s tree of algebraic data structures. Dark and blue boxes correspond to abstract and concrete classes, respectively. **BPAS** counts many other classes, for instance `Intervals` and `RegularChains`.

### 5.2.3 **BPAS**'s DFT code generator

**BPAS** also uses a Python generator to generate **C++** code for DFT 2.4 algorithms. The second author of [4] implements a self-generating code able to obtain an optimized DFT (using methods such as loop-unrolling and optimizing the pipeline) in any radix.

The entrance of the DFT algorithm is the function `DFT_eff` which is shown in Algorithm 7. The `Shuffle` function is used for permuting the inputs of DFT shown in Algorithm 8. The function `DFT_rec`, shown in Algorithm 9, and the function `DFT_iter`, shown in Algorithm 20, are called in `DFT_eff`.

When the size is broken down to less than or equal to 16, **BPAS** calls a straight-line program code directly. Algorithm 21 and Algorithm 22 show the snippet code of the

loop-unrolling technique for the functions `FFT_2POINT` and `FFT_4POINT` of input size 2 and 4, respectively.

---

**Algorithm 7:** `DFT_eff(n, A,  $\Omega$ , H)`


---

**Input:**  $n = 2^r$ ,  $A = [a_0, a_1, \dots, a^{n/2-1}, \dots, a^{n-1}]$  coefficient array of the input polynomial  $a$ ,  $\Omega = [1, \omega, \dots, \omega^{N/2-1}, \dots, \omega^{N-1}]$  an array of the consecutive powers of a primitive  $N$ -th root of unity  $\omega$ , where  $N \geq n$  and  $N$  is a power of two, and  $H$  is a threshold below which computations are expected to fit in cache.

**Output:**  $DFT(n, A, \Omega)$  computed as  $A[i] = a(\omega^i)$ , for  $0 \leq i \leq n-1$ .

**if**  $n = 1$  **then**

**return**;

**else if**  $n > H$  **then**

`DFT_rec(n, A,  $\Omega$ , H)`;

`ArrayBitReversal(n, A)`;

`DFT_iter(n, A,  $\Omega$ )`;

---



---

**Algorithm 8:** `Shuffle(n, A)`


---

**Input:**  $n = 2^r$ ,  $A = [a_0, a_1, \dots, a^{n/2-1}, \dots, a^{n-1}]$  coefficient array of a polynomial  $a$ ,

**Output:**  $A = [a_0, a_2, \dots, a_{n-2}, a_1, a_3, \dots, a_{n-1}]$

allocate  $B[0 \dots n/2 - 1]$ ;

**for**  $k$  *from* 0 *to*  $n/2 - 1$  **do**

$A[k] = A[2k]$ ;

$B[k] = A[2k + 1]$ ;

**for**  $k$  *from* 0 *to*  $n/2 - 1$  **do**

$A[k + n/2] = B[k]$ ;

---

### 5.2.4 The use of the BPAS library

The implementation of the TFT code relies on BPAS's modular arithmetic operations in the Montgomery mode. In other words, we take advantage of highly optimized machine-word operations, such as addition, subtraction and multiplication corresponding to functions `AddModSpe`, `SubModSpe` and `MontMulModSpe_OPT3_AS_GENE_INLINE`, respectively in BPAS. In particular, machine-word multiplication operations are implemented by assembly code, which can run twice as fast as its counterpart in C code.

**Algorithm 9:**  $\text{DFT\_rec}(n, A, \Omega, H)$ 


---

**Input:**  $n = 2^r$ ,  $A = [a_0, a_1, \dots, a^{n/2-1}, \dots, a^{n-1}]$  coefficient array of the input polynomial  $a$ ,  $\Omega = [1, \omega, \dots, \omega^{N/2-1}, \dots, \omega^{N-1}]$  an array of the consecutive powers of a primitive  $N$ -th root of unity  $\omega$ , where  $N \geq n$  and  $N$  is a power of two.

**Output:**  $\text{DFT}(n, A, \Omega)$  computed as  $A[i] = a(\omega^i)$ , for  $0 \leq i \leq n-1$ .  
 $\text{step} = (\text{the number of } \Omega) / n$ ;

$\text{Shuffle}(n, A)$ ;

$\text{DFT\_eff}(n/2, A, \Omega, H)$ ;

$\text{DFT\_eff}(n/2, A + n/2, \Omega, H)$ ;

**for**  $k$  *from* 0 *to*  $n/2 - 1$  **do**

$s = \text{step } k$ ;
$v = A[k + n/2]$ ;
$u = A[k]$ ;
$t = \Omega_s v$ ;
$A[k] = u + t$ ;
$A[k + n/2] = u - t$ ;

---

In addition, BPAS provides function calls to compute tables of (1)  $n$ -th roots of unity  $\omega$ , that is,  $\{\omega^0, \omega^1, \omega^2, \omega^3, \omega^4, \dots, \omega^{n-1}\}$  by `PBPAS::RootsTableSpe`; and (2) the inverse of these  $\omega$  by `PBPAS::InverseRootsTable`. Then we only pre-compute them once and store each table in its respective array. Thus, for each TFT iteration, we access these tables instead of redundant computations.

The BPAS library implements an efficient DFT algorithm restricted to input sizes equal to powers of two. Thus, we invoke its functions, such as `DFT_eff`, when either (1) the input size equals a power of two, or (2) during any TFT iteration, the divided block size equals a power of two. Furthermore, `Shuffle_tft` is called to permuted the outputs of BPAS's DFT into order for TFT.

## 5.3 Code generation for TFT and ITFT

Our TFT and ITFT source code are generated by (1) a Python file: `generate_tft_relax.py`, which is the relaxed TFT code generator; and (2) a template code: `generate_tft_tree_template.cpp`, for which the implemented algorithms are described in Section 4.2 and Section 4.3. To use this code generator, one executes `.config_tft_tree` to configure the generated files and their primes. To illustrate, the configuration file contains the following lines:

```
tft_tree1
```

```

4179340454199820289
tft_tree2
2485986994308513793

```

After configuration, two files are generated: `tft_tree1.cpp` and `tft_tree2.cpp` for the primes 4179340454199820289 and 2485986994308513793, respectively.

The generated C++ file contains three high-level functions for TFT (`TFT_Basecase`, `TFT_Core` and `TFT_Wrapper`) and three for ITFT (`ITFT_Basecase`, `ITFT_Core` and `ITFT_Wrapper`). The methods `TFT_Basecase` and `ITFT_Basecase` are used to compute the base case scenario, for which users can specify the base case size (16 by default). `TFT_Core` and `ITFT_Core` are the implementations of the core algorithms of the TFT and ITFT, respectively. Each one recursively divides the original problem into smaller sizes until it reaches the base case. Functions `TFT_Wrapper` and `ITFT_Wrapper` are top-level functions calling their core algorithms and generating random inputs.

### 5.3.1 Details of the Python code generator

To generate the header files of C++ code (e.g. \*.h) in Python, one can use `header.write` and define a constant variable as follows:

```

header.write("#include <iostream>\n")
header.write("#define Mont_two 3458764513820540920\n")

```

Corresponding to its source code, `code.write` is needed to generate C++ code for any .cpp file. Furthermore, `line.replace` is necessary when any string is to be replaced. To illustrate, in the following example, if the string `void TFT_Core` is found in a line, we replace the first parameter in `line.replace` by the second one, in both the header and source file.

```

if "void TFT_Core" in line:
    code.write(line.replace("TFT_Core", "TFT_Core_p%i"%num))
    header.write(line.replace("{", "; \n").replace("TFT_Core", "TFT_Core_p%i"%num))

```

This code generator requires a template file to open, such that each generated file has the same format as the content defined in `generate_tft_template.cpp`. In Python, we use `open` to read from a file as below.

```

template = open("generate_tft_template.cpp", "r")
for line in template:
    ...

```

For further detail, Appendix 7 shows the full content of our Python script.

### 5.3.2 The structure of the template file

The template file is used for different prime numbers and their  $n$ -th roots of unity  $\omega$ . Each high-level function is defined in the template file, and it is used by the code generator to rewrite. For instance, the method `TFT_Core` is passed to the code generator for a prime number  $p_1$  such that a method `TFT_Core_p1` is generated in the source file: `tft_tree1.cpp`. To illustrate, the following pieces of code are shown in template and generated file, respectively.

```

/* in template file: generate_tft_template.cpp */
void TFT_Core(int n, int l, int m, int basecase, ... ){
    ...
}

/* in generated file: tft_tree1.cpp */
void TFT_Core_p1(int n, int l, int m, int basecase, ... ){
    ...
}

```

This particular method, `TFT_Core`, implements the relaxed Cooley-Tukey algorithm as Equation 4.1. Algorithm 11 shows the `TFT_Core` code in the template file, which consists of the base case computation and two parts of the computation in Equation 4.1. We refer the implementation of the right part and shuffle part to  $T_{r,s}^n \cdot (\text{TFT}_{\bar{c}} \otimes I_{\ell_r})$  and that of the left part to  $(I_{m_s} \otimes \text{TFT}_{\bar{r}})$ . Note that parameter `invec` is the input vector and `invectmp` is an intermediate array for data reuse.

For the method `TFT_Basecase`, we apply the TFT algorithm for 5 base cases, namely, the input sizes of 2, 4, 8, 16 and 32. Algorithm 10 implements the forward TFT as described in 4, but for the base case of size of 8 in particular. We generate random inputs of size  $K$  (given by the user) in the body of the method `TFT_Wrapper`, as shown below:

```

sfixn *Ap = (sfixn *)calloc(K, sizeof(sfixn));
Ap = EX_RandomUniPolyCoeffsVec(K, p);

```

and then `TFT_Core` is called to execute the TFT on these inputs.

---

**Algorithm 10:** TFFT\_8POINT(sfixn \*A,sfixn \*W)

---

**Input:**  $A$  the coefficient array of the input polynomial,  $W$  a primitive 8-th root of unity.

**Output:** Array  $A$ .

sfixn \*Wp = W + (8 << 1) - 4;

sfixn u = A[0];

sfixn t = A[4];

A[0] = AddModSpe(u, t);

A[4] = SubModSpe(u, t);

u = A[2];

t = A[6];

A[2] = AddModSpe(u, t);

A[6] = SubModSpe(u, t);

u = A[1];

t = A[5];

A[1] = AddModSpe(u, t);

A[5] = SubModSpe(u, t);

u = A[3];

t = A[7];

A[3] = AddModSpe(u, t);

A[7] = SubModSpe(u, t);

A[6] = MontMulModSpe\_OPT3\_AS\_GENE\_INLINE(A[6], \*(Wp - 3));

A[7] = MontMulModSpe\_OPT3\_AS\_GENE\_INLINE(A[7], \*(Wp - 3));

TFFT\_AddSubSpeSSEModInplace(A, A + 4, A + 2, A + 6);

TFFT\_AddSubSpeSSEModInplace(A + 1, A + 5, A + 3, A + 7);

A[5] = MontMulModSpe\_OPT3\_AS\_GENE\_INLINE(A[5], \*(Wp - 11));

A[3] = MontMulModSpe\_OPT3\_AS\_GENE\_INLINE(A[3], \*(Wp - 10));

A[7] = MontMulModSpe\_OPT3\_AS\_GENE\_INLINE(A[7], \*(Wp - 9));

TFFT\_AddSubSpeSSEModInplace(A, A + 4, A + 1, A + 5);

TFFT\_AddSubSpeSSEModInplace(A + 2, A + 6, A + 3, A + 7);

---

## 5.4 Optimization techniques

### 5.4.1 The use of machine code

We optimize efficiency-critical low-level routines (like Montgomery modular multiplication) with the intention of fully taking advantage of hardware features, in particular instruction pipelining and vectorized instructions (SSE2, SSE4). An example of such usage of assembly code is shown in Algorithm 12.



### 5.4.2 Hard-coded constants

For a prescribed prime number, quantities like  $R^{-1}$  and  $p'$  used in Montgomery arithmetic, see Section 2.2. In the header files, `MY_PRIME1` is the prime number  $p$  used for the TFT and ITFT algorithms while `INV_PRIME1` is  $p'$ . Also, `Mont_two` and `INV_Mont_two` are  $R$  and  $R^{-1}$ , respectively:

```
#define MY_PRIME1 4179340454199820289
#define INV_PRIME1 4179340454199820287
#define Mont_two 3458764513820540920
#define INV_Mont_two 2559286960657440491
```

### 5.4.3 Unrolling loops

Loop unrolling is a well-known loop transformation technique interpreting the iterations into a sequence of instructions so as to reduce the loop execution overhead. For TFT on input sizes 2, 4, 8, 16, 32, an straight-line program (SLP) code was manually generated. Such code is meant to minimize arithmetic calculations and optimize the use of hardware pipelines. Algorithm 10 and algorithm 19 show examples for the un-rolling code. Note that the subroutines `AddModSpe` and `SubModSpe`, are called: they perform modular addition and modular subtraction, respectively. Those subroutines are defined in Algorithm 14 and Algorithm 15, respectively.

### 5.4.4 Work space

Dynamic allocation/deallocation of temporary arrays is avoided by passing the necessary work space to the top-level TFT function. This work space is a sufficiently large array which is passed as an argument of both recursive functions `TFT_Core` and `ITFT_Core`. In Algorithm 11, `TFT_Core` is recursively called with parameter `invectmp` which is this work space vector.

### 5.4.5 Montgomery arithmetic

Arithmetic operations in  $\mathbb{Z}/p\mathbb{Z}$  are all performed in Montgomery representation to speedup modular multiplication. Consider Algorithm 16, in which we seek to apply the Montgomery arithmetic to compute modular products and inverses. Let  $R > p$  with  $\gcd(R, p) = 1$  and  $p$  is a prime number greater than 2. We assume to have at hand two functions `Montgomery_convert_in` and `Montgomery_convert_out` to convert the representation of

a modular integer from the usual residual representation to the Montgomery representation. Then, we apply two operations `Montgomery_product` and `Montgomery_inverse`, whenever a modular product and modular inverse calculation are required. The former operation is defined in Section 2.6, and the latter is defined in Algorithm 2.25 of [11]. Hence, using the Montgomery arithmetic, one obtains the corresponding code in Algorithm 17.

### 5.4.6 Cache-efficient transpose

Transposition is an efficiency-critical subroutine for FFT and TFT algorithms. We use a cache-optimal transposition method. The serial version of this method is implemented in Algorithm 18, while the parallel version is implemented in Algorithm 1. Both versions are based on the divide and conquer method of [9] which is described in Section 2.13. Note that in Algorithms 18 and 1, `lda` and `ldb` are defined as the number of columns and rows, respectively, of the input matrix.

### 5.4.7 Parallel code generation

Our Python code generator can produce either `C++` and `CilkPlus` code. The user of the BPAS library switches between serial and parallel code by setting the environment variable `SERIAL` to either 0 or 1.

Following the work in [19] on the parallelization of multi-dimensional FFTs and TFTs, we parallelize the `TFT_Core` and `ITFT_Core` by executing for-loops with the `cilk_for` construct of `CilkPlus`. As we shall see in Section 6, we have verified experimentally that the default grain size of those `cilk_for` loops ensured that parallelism overheads were negligible.

While this parallelization scheme may look quite simple, one should note that the structure of the algorithms underlying `TFT_Core` and `ITFT_Core` made it easy.

**Algorithm 11:** TFT\_Core( $invec, \omega, p, n, \ell, m, basecase, invectmp$ )

---

**Input:**  $invec$  the coefficient array of the input polynomial,  $\omega$  a primitive  $n$ -th root of unity,  $n, \ell, m$  is defined in 4.2,  $invectmp$  a temporary variable.

**Output:** Array  $invec$  returns the results of TFT.

```

/* base case computation */
if  $n \leq basecase$  then
  TFT_Basecase( $n, e, invec, \omega, invectmp$ );
  return;
if  $n > basecase^2$  then
   $s = basecase; r = n/basecase;$ 
else
   $s = 2^{\lceil (\log_2 n)/2 \rceil}; r = 2^{\lfloor (\log_2 n)/2 \rfloor};$ 
 $\ell_s = \lceil \ell/r \rceil, m_s = \lceil m/r \rceil, \ell_r = \min(\ell, r), m_r = \min(m, r);$ 
/* right part and shuffle part:  $T_{r,s}^n \cdot (\text{TFT}_{\bar{e}} \otimes \mathbb{I}_{\ell_r})$  */
 $new\_r\_n = s, new\_r\_l = \ell_s, new\_r\_m = m_s, new\_r\_a = n/new\_r\_n, new\_r\_b = \ell_r;$ 
if  $new\_r\_n \leq basecase$  then
   $right\_step = \log_2 new\_r\_n;$ 
  transpose( $invec, new\_r\_b, invectmp, s, 0, s, 0, new\_r\_b$ );
  for  $j$  from 0 to  $new\_r\_b - 1$  do
     $pi = j * new\_r\_n;$ 
    TFT_Basecase( $new\_r\_n, right\_step, invectmp + pi, \omega, invec + pi$ );
  transpose( $invectmp, s, invec, new\_r\_b, 0, new\_r\_b, 0, s$ );
else
  for  $j$  from 0 to  $new\_r\_b - 1$  do
    TFT_Core( $invec + j * new\_r\_n, \omega, p, new\_r\_n, new\_r\_l, new\_r\_m, basecase, invectmp + j * new\_r\_n$ );
/* left part:  $(\mathbb{I}_{m_s} \otimes \text{TFT}_{\bar{r}})$  */
 $new\_l\_n = r, new\_l\_l = \ell_r, new\_l\_m = m_r, new\_l\_a = n/new\_l\_n, new\_l\_b = 1;$ 
if  $new\_l\_n \leq basecase$  then
   $left\_step = \log_2 new\_l\_n;$ 
  for  $j$  from 0 to  $new\_l\_a - 1$  do
     $pi = j * new\_l\_n;$ 
    TFT_Basecase( $new\_l\_n, left\_step, invec + pi, \omega, invectmp + pi$ );
else
  for  $j$  from 0 to  $new\_l\_a - 1$  do
    TFT_Core( $invec + i * new\_l\_n, \omega, p, new\_l\_n, new\_l\_l, new\_l\_m, basecase, invectmp + i * new\_l\_n$ );

```

---

---

**Algorithm 12:** MontMulModSpe\_OPT3\_AS\_GENE\_INLINE(sfixn  $a$ , sfixn  $b$ )

---

**Input:**  $a, b$  can be any arbitrary integer number.

**Output:**  $a$  is the product of  $a$  and  $b$  in Montgermory mode.

```
asm("mulq %2\n\t"
    "movq %%rax,%%rsi\n\t"
    "movq %%rdx,%%rdi\n\t"
    "imulq %3,%%rax\n\t"
    "mulq %4\n\t"
    "add %%rsi,%%rax\n\t"
    "adc %%rdi,%%rdx\n\t"
    "subq %4,%%rdx\n\t"
    "mov %%rdx,%%rax\n\t"
    "sar $63,%%rax\n\t"
    "andq %4,%%rax\n\t"
    "addq %%rax,%%rdx\n\t"
    : "=d" (a)
    : "a"(a), "rm"(b), "b"((sfixn) INV_PRIME1), "c"((sfixn) MY_PRIME1)
    : "rsi", "rdi");
return a;
```

---



---

**Algorithm 13:** unrolledSpe8MontMul(sfixn\*  $input1$ , sfixn\*  $input2$ ,  
 MONTP\_OPT2\_AS\_GENE \*  $pPtr$ )

---

**Input:**  $input1, input2$  can be any arbitrary integer number.

**Output:** Return the product of  $input1$  and  $input2$ .

```
asm ("movq (%%rsi),%%rax\n\t"
    "mulq (%%rdi)\n\t"
    "pinsrq $0,%%rdx,%%xmm0\n\t"
    "mulq %2\n\t"
    "movq %%rax,%%r8\n\t"
    "pinsrq $0,%%rdx,%%xmm4\n\t"
    "movq 8(%%rsi),%%rax\n\t"
    "mulq 8(%%rdi)\n\t"
    "pinsrq $1,%%rdx,%%xmm0\n\t"
    "mulq %2\n\t"
    "movq %%rax,%%r9\n\t"
    "pinsrq $1,%%rdx,%%xmm4\n\t"
    ...
```

---

---

**Algorithm 14:** AddModSpe(sfixn  $a$ , sfixn  $b$ )

---

**Input:**  $a, b$  long int numbers.  
**Output:**  $r$  is the sum of  $a$  and  $b$ .  
sfixn  $r = a + b$ ;  
 $r- = \text{MY\_PRIME1}$ ;  
 $r+ = (r \gg \text{BASE\_1}) \& \text{MY\_PRIME1}$ ;  
**return**  $r$ ;

---



---

**Algorithm 15:** SubModSpe(sfixn  $a$ , sfixn  $b$ )

---

**Input:**  $a, b$  long int numbers.  
**Output:**  $r$  returns the difference between  $a$  and  $b$ .  
sfixn  $r = a - b$ ;  
 $r+ = (r \gg \text{BASE\_1}) \& \text{MY\_PRIME1}$ ;  
**return**  $r$ ;

---



---

**Algorithm 16:** Prod\_Inv( $x, y, z, p$ )

---

**Input:**  $x, y, z$  are numbers in normal mode.  $p$  is a prime number.  
**Output:** Return  $x$  in  $Z/pZ$  where  $p$  is a prime number.  
**for**  $i$  from 1 to  $n$  **do**  
     $x = (x + y[i] * z[i]) \bmod p$ ;  
     $x = (1/x) \bmod p$ ;

---



---

**Algorithm 17:** Prod\_Inv\_Mont( $x, y, z, p$ )

---

**Input:**  $x, y, z$  are numbers in normal mode and  $p$  is a prime number.  
**Output:**  $x$  is converted to Montgomery mode.  
**for**  $i$  from 1 to  $n$  **do**  
     $y_M[i] = \text{Montgomery\_convert\_in}(y[i])$ ;  
     $z_M[i] = \text{Montgomery\_convert\_in}(z[i])$ ;  
 $x_M = \text{Montgomery\_convert\_in}(x)$ ;  
**for**  $i$  from 1 to  $n$  **do**  
     $prod_M = \text{Montgomery\_product}(y_M[i], x_M[i])$ ;  
     $x_M = (x_M + prod_M) \bmod p$ ;  
     $x_M = \text{Montgomery\_inverse}(x_M)$ ;  
 $x = \text{Montgomery\_convert\_out}(x_M)$ ;

---

---

**Algorithm 18:** transpose\_serial(sfixn \*A, int lda, sfixn \*B, int ldb, int i0, int i1, int j0, int j1)

---

**Input:**  $A, B$  matrix represented in array,  $lda$  number of columns,  $ldb$  number of rows,  $i0, i1$  index of rows,  $j0, j1$  index of columns.

**Output:** Array  $A$ .

tail:

int  $di = i1 - i0, dj = j1 - j0$ ;

**if**  $di \geq dj$  &&  $di > TRANSPOSETHRESHOLD$  **then**

    int  $im = (i0 + i1)/2$ ;  
    transpose\_serial( $A, lda, B, ldb, i0, im, j0, j1$ );  
     $i0 = im$ ; **goto** tail;

**else if**  $dj > TRANSPOSETHRESHOLD$  **then**

    int  $jm = (j0 + j1)/2$ ;  
    transpose\_serial( $A, lda, B, ldb, i0, i1, j0, jm$ );  
     $j0 = jm$ ; **goto** tail;

**else**

**for**  $i$  from  $i0$  to  $i1$  **do**  
        **for**  $j$  from  $j0$  to  $j1$  **do**  
             $B[j * ldb + i] = A[i * lda + j]$ ;

---



---

**Algorithm 19:** FFT\_8POINT(sfixn \*A, sfixn \*W)

---

**Input:**  $A$  the coefficient array of the input polynomial,  $W$  a primitive 8-th root of unity.

**Output:** Array  $A$  returns the result of FFT.

sfixn \* $Wp = W + (8 \ll 1) - 4$ ;

sfixn  $u = A[0]$ ;

sfixn  $t = A[1]$ ;

$A[0] = \text{AddModSpe}(u, t)$ ;

$A[1] = \text{SubModSpe}(u, t)$ ;

$u = A[2]$ ;

$t = A[3]$ ;

$A[2] = \text{AddModSpe}(u, t)$ ;

$A[3] = \text{SubModSpe}(u, t)$ ;

$u = A[4]$ ;

$t = A[5]$ ;

$A[4] = \text{AddModSpe}(u, t)$ ;

$A[5] = \text{SubModSpe}(u, t)$ ;

...

---

---

**Algorithm 20:** DFT\_iter( $n, A, \Omega$ )

---

**Input:**  $n = 2^r$ ,  $A$  the array for the coefficient of a polynomial  $a$  sorted by the *DFT ordering*.  $\Omega = [1, \omega, \dots, \omega^{N/2-1}, \dots, \omega^{N-1}]$  an array of the consecutive powers of a primitive  $N$ -th root of unity  $\omega$ , where  $N \geq n$  and  $N$  is a power of two,.

**Output:**  $DFT(n, A, \Omega)$  computed as  $A[i] = a(\omega^i)$ , for  $0 \leq i \leq n-1$ .  
 $step = (\text{the number of } \Omega) / n$ ;

**for**  $i$  from 1 to  $r$  **do**

```

    /* Traversing the tree, bottom-up                               */
     $m = 2^i$ ;
    for  $k$  from 0 to  $n-1$  by  $m$  do
        /* for each internal node from left to right               */
        for  $j$  from 0 to  $m/2-1$  do
            /* combine its two children                             */
             $s = \text{step } j \ n/m$ ;
             $t = \Omega_s \ A[k+j+m/2]$ ;
             $u = A[k+j]$ ;
             $A[k+j] = u+t$ ;
             $A[k+j+m/2] = u-t$ ;

```

---



---

**Algorithm 21:** FFT\_2POINT(sfixn  $*A$ , sfixn  $*W$ )

---

**Input:**  $A$  the coefficient array of the input polynomial,  $W$  a primitive 2-th root of unity.

**Output:** Array  $A$  returns the result of FFT.

sfixn  $u = A[0]$ ; sfixn  $t = A[1]$ ;

$A[0] = \text{AddModSpe}(u, t)$ ;  $A[1] = \text{SubModSpe}(u, t)$ ;

---



---

**Algorithm 22:** FFT\_4POINT(sfixn  $*A$ , sfixn  $*W$ )

---

**Input:**  $A$  the coefficient array of the input polynomial,  $W$  a primitive 4-th root of unity.

**Output:** Array  $A$  returns the result of FFT.

sfixn  $*Wp = W + (4 \ll 1) - 4$ ;

sfixn  $w = A[1]$ ;

$A[1] = A[2]$ ;  $A[2] = w$ ;

sfixn  $u = A[0]$ ; sfixn  $t = A[1]$ ;

$A[0] = \text{AddModSpe}(u, t)$ ;  $A[1] = \text{SubModSpe}(u, t)$ ;

$u = A[2]$ ;  $t = A[3]$ ;

$A[2] = \text{AddModSpe}(u, t)$ ;  $A[3] = \text{SubModSpe}(u, t)$ ;

$A[3] = \text{MontMulModSpe\_OPT3\_AS\_GENE\_INLINE}(A[3], *(Wp-3))$ ;

$\text{AddSubSpeSSEModInplace}(A, A+2)$ ;

---

# Chapter 6

## Experimentation of Serial and Inverse TFFT (ITFFT)

In this chapter, we first describe the environmental setup for our experimentation in Section 6.1. In Section 6.2, we compare running times, clock cycles and cache misses among serial FFT 3.1, serial TFFT and serial ITFFT. Due to the relaxed scheme of TFFT 4.2, our serial TFFT code is competitive with FFT code, while ITFFT 4.3 runs slower because of higher overheads. In Section 6.3, we show the trend of running times of serial TFFT and serial ITFFT when the input size increases within a range. Running times of both algorithms increase linearly. Finally, in Section 6.4, the experimental results of our parallel methods are displayed and analyzed. We show that the parallel TFFT is 5.31 times faster than the serial FFT (at input size  $2^{23}$ ) and 6.78 times faster than the serial TFFT (at input size  $2^{26}$ ).

### 6.1 Experimental setup

Our input vector consists of  $\{1, 2, \dots, n\}$  for any arbitrary size  $n$ . We collected experimentation results of various values of  $n$  on an Intel X5650 machine with 12 cores (24 cores with Hyper-Threading) with frequency of 2.67GHz. Our code was compiled by GCC version 4.8.1, with `-lbpas -lmodpnLINUXINTEL64 -lcilkrts` linking flags and `-c -O2 -g -fcilkplus -DLINUXINTEL64=1` compilation flags. Note that `-lbpas -lmodpnLINUXINTEL64` and `-DLINUXINTEL64=1` are required by the BPAS library and `-lcilkrts` and `-fcilkplus` are required by CilkPlus.

In order to collect performance measures such as clock cycles and cache misses, we used the Linux Perf performance analysis tool (see [https://perf.wiki.kernel.org/index.php/Main\\_Page](https://perf.wiki.kernel.org/index.php/Main_Page)).



We relied on the version `perf 3.2.18` with command line: `perf stat -e cycles -e cache-misses binary_name`. And we used `Cilkview` [13], extracted from the Intel `CilkPlus SDK build 4225` package, to collect *work*, *span*, *burdened span*, and *parallelism*. The collected data are then displayed with `plot 4.4`.

## 6.2 Comparison of serial code

Running times for serial FFT, serial TFFT and serial ITFFT can be found in Figure 6.1. The x-axis is a logarithm to base 2 of the input size, i.e.  $12 = \log_2(4096)$ . The y-axis is a logarithm to base 10 of the running time in seconds, i.e.  $-0.5 = \log_{10}(0.31622)$ . Clock cycles for serial FFT, serial TFFT and serial ITFFT can be found in Table 6.1. Cache misses for serial FFT, serial TFFT and serial ITFFT can be found in Table 6.2. There is a little variation in the number of clock cycles and the number of cache misses among code executions. For instance, among ten code executions, the number of clock cycles for serial TFFT ranges from 7,376,556,355 to 7,370,257,970 (a narrow range), for input size 8,388,608. We observe that our serial TFFT code runs slightly slower than the serial FFT code, while their clock cycles and cache misses are coherent with this result. This is due to the fact that our TFFT follows the relaxed scheme. Furthermore, our serial ITFFT runs much more slowly than the serial FFT code, since this strictly ITFFT algorithm has a complex data flow. Thus, ITFFT suffers from higher overheads and is not competitive with FFT code.

$\log_2(n)$	Serial FFT	Serial TFFT	Serial ITFFT
10	8,567,358	8,990,501	10,406,586
11	7,633,606	10,494,993	8,959,363
12	10,465,353	8,930,546	11,382,820
13	10,127,8003	10,088,607	19,096,836
14	15,285,211	15,176,968	32,262,386
15	19,398,703	26,807,539	59,212,666
16	39,789,124	41,899,422	128,884,441
17	71,928,652	98,468,975	259,321,821
18	175,679,865	184,111,555	571,758,675
19	344,479,786	312,873,524	1,130,050,020
20	723,018,779	807,968,662	2,531,523,723
21	1,439,760,2437	1,767,759,406	5,199,437,793
22	3,281,504,234	3,590,756,599	11,588,723,554
23	6,664,053,326	7,567,093,379	22,429,454,018
24	14,050,973,272	14,995,597,184	48,880,609,613

Table 6.1: Clock cycles for serial FFT, TFFT and ITFFT with input size  $n$ .

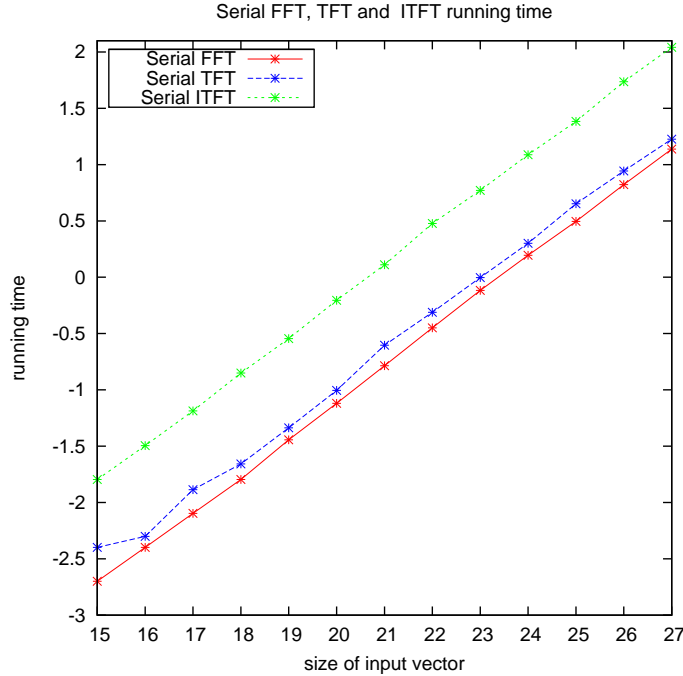


Figure 6.1: Running time (secs) of serial FFT, TFT and ITFT.

### 6.3 Results for serial TFT between two consecutive powers of two

For the choice of input size  $n$ , we use values of the form  $2^k + c_1 * 2^{k-1} + c_2 * 2^{k-2} + c_3 * 2^{k-3} + c_4 * 2^{k-4}$  where  $c_1, c_2, c_3, c_4$  are either 0 or 1. Thus,  $2^4 + 1 = 17$  choices of  $n$  are given, including  $2^{k+1}$  itself. Figure 6.2 and Figure 6.3 show running time comparisons between serial TFT and serial ITFT on this range of  $2^k \dots 2^{k+1}$ , where we set  $k$  to 22 and 23, respectively. The x-axis represents the increased value based on  $k$  and the y-axis represents a logarithm to base 10 of the running time in seconds. Both algorithms have a nearly straight curve between two consecutive powers of two, as expected. This is satisfactory.

### 6.4 Results for TFT and ITFT parallel code

We run the input size  $n = 2^k$ , for an integer  $k$  from 14 to 27 on 4 cores and 12 cores. Figure 6.4 and Figure 6.5 show increases in speed between serial TFT and parallel TFT, as well as between serial ITFT and parallel ITFT. One can observe that we obtain an increase in speed of approximately factor 7 for the case of TFT at input size  $2^{20}$ , and

$\log_2(n)$	Serial FFT	Serial TFT	Serial ITFT
10	11,999	12,132	8,796
11	8,493	15,880	8,811
12	17,612	7,399	7,288
13	9,324	11,301	16,342
14	14,696	12,902	17,289
15	11,881	36,331	21,017
16	28,055	40,844	43,707
17	55,692	90,325	95,380
18	210,514	209,629	273,559
19	400,919	449,267	530,346
20	935,616	1,289,325	1,442,793
21	1,773,831	2,900,084	3,391,734
22	4,090,681	5,965,302	8,234,309
23	8,424,800	13,568,997	18,699,274
24	17,780,019	26,304,109	33,878,567

Table 6.2: Cache misses for serial FFT, TFT and ITFT with input size  $n$ .

factor 4.5 at input size  $2^{27}$  for the case of ITFT on a 12 cores node. Table 6.3 and Table 6.4 show Cilkview results for TFT and ITFT, respectively, of input sizes of  $2^{22}$  and  $2^{23}$  on a 12 cores node. Note that for both cases, the span and burdened span are approximately equivalent, indicating that our parallel code has parallelism overhead under control.

The increase in speed may appear low for TFT, but for that type of mergesort-like algorithm on multi-cores, the increase in speed is as expected. In fact, this is confirmed by the Cilkview results. However, the speedup curves for ITFT are not satisfactory. According to Cilkview, the burden on the span is low, as desired. Nevertheless, an overhead undetectable by Cilkview seems to have a major negative impact. This could be due to cache misses of type false/true sharings; however, these cannot be measure by `perf`.

The speedup for parallel TFT and parallel ITFT with grain sizes of 512, 1024 and 2048 can be found in Figure 6.6 and Figure 6.7, respectively. Based on these figures, we choose 1024 as the grain size for both TFT and ITFT.

Table 6.5 compares the running times among serial FFT, serial TFT and parallel TFT, as well as the speedups between serial FFT and parallel TFT and between serial TFT and parallel TFT. We observe that our parallel code can outperform its serial code by a factor of 6.9 and the corresponding serial FFT code by a factor of 5.3.

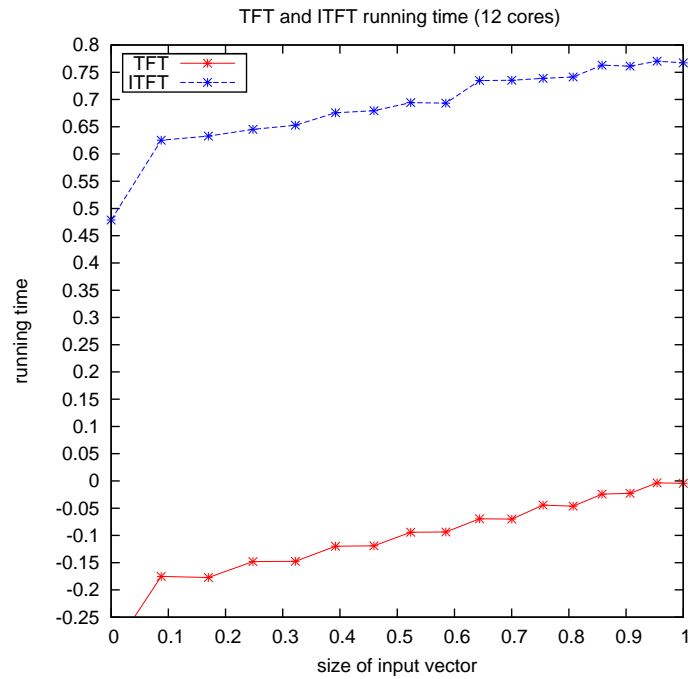


Figure 6.2: TFT and ITFT results on a range between  $2^{22}$  and  $2^{23}$  on a 12 cores node.

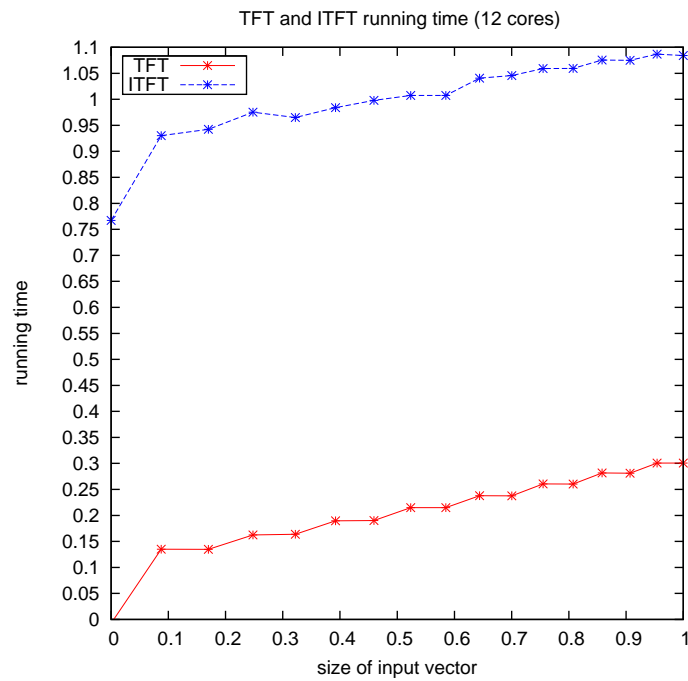


Figure 6.3: TFT and ITFT results on a range between  $2^{23}$  and  $2^{24}$  on a 12 cores node.

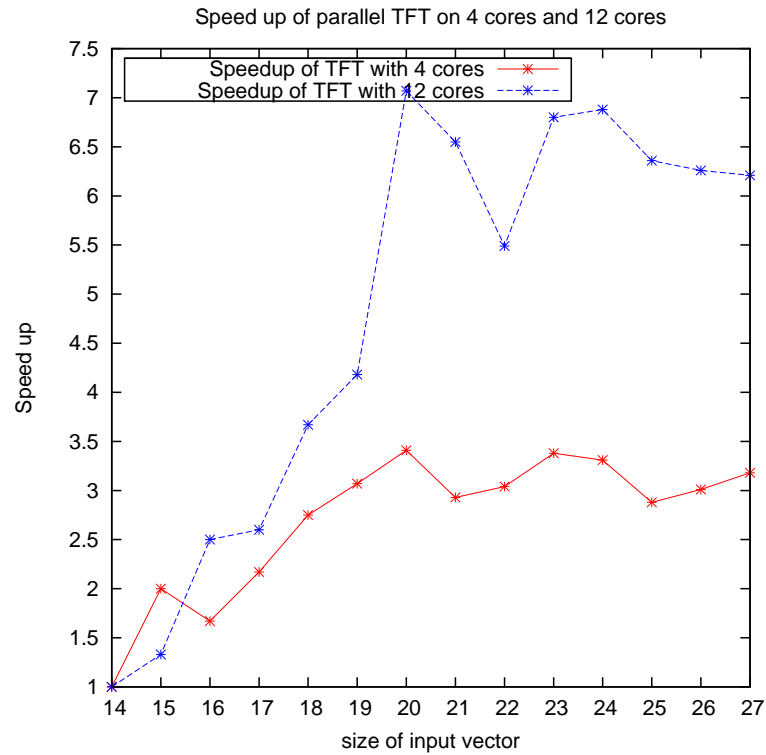


Figure 6.4: TFFT speedup on 4 cores and 12 cores.

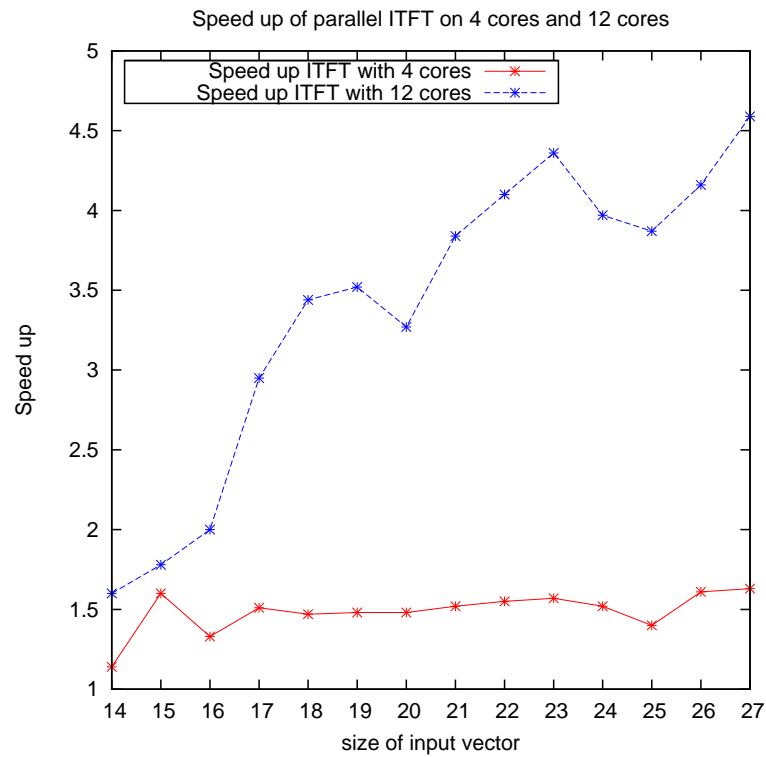


Figure 6.5: ITFT speedup on 4 cores and 12 cores.

N	4194304	8388608
Whole Program Statistics	TFT	TFT
Work	3423032744	6677120612
Span	453096525	898770886
Burdened span	455596525	901520886
Parallelism	7.55	7.43
Burdened parallelism	7.51	7.41
Number of spawns/syncs	1643758	2693870
Average instructions / strand	694	826
Strands along span	201	221
Average instructions / strand on span	2254211	4066836
Total number of atomic instructions	1643774	2693886
Frame count	3501596	5601820

Table 6.3: Cilkview analysis of parallel TFT on input size  $N$ , where *work*, and *span* rows are the number of instructions, and *parallelism* is the ratio of *Work*/*Span*.

N	4194304	8388608
Whole Program Statistics	ITFT	ITFT
Work	24383770364	48947869668
Span	466043686	902702311
Burdened span	468278686	905012311
Parallelism	52.32	54.22
Burdened parallelism	52.07	54.09
Number of spawns/syncs	29036550	58071054
Average instructions / strand	279	280
Strands along span	175	181
Average instructions / strand on span	2663106	4987305
Total number of atomic instructions	29036566	58071070
Frame count	81936396	163868700

Table 6.4: Cilkview analysis of parallel ITFT on input size  $N$ , where *work*, and *span* rows are the number of instructions, and *parallelism* is the ratio of *Work*/*Span*.

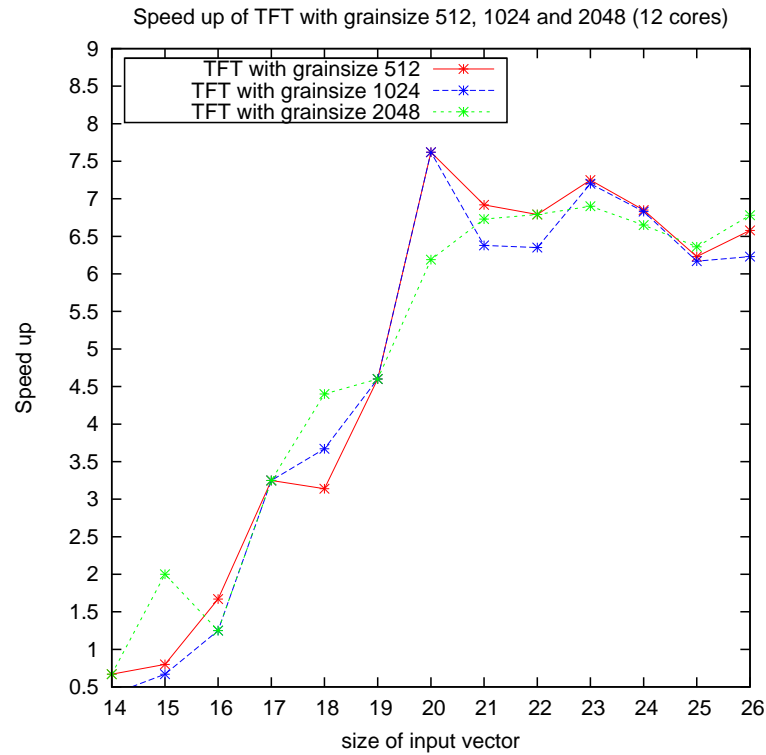


Figure 6.6: Parallel TFFT with different grain sizes.

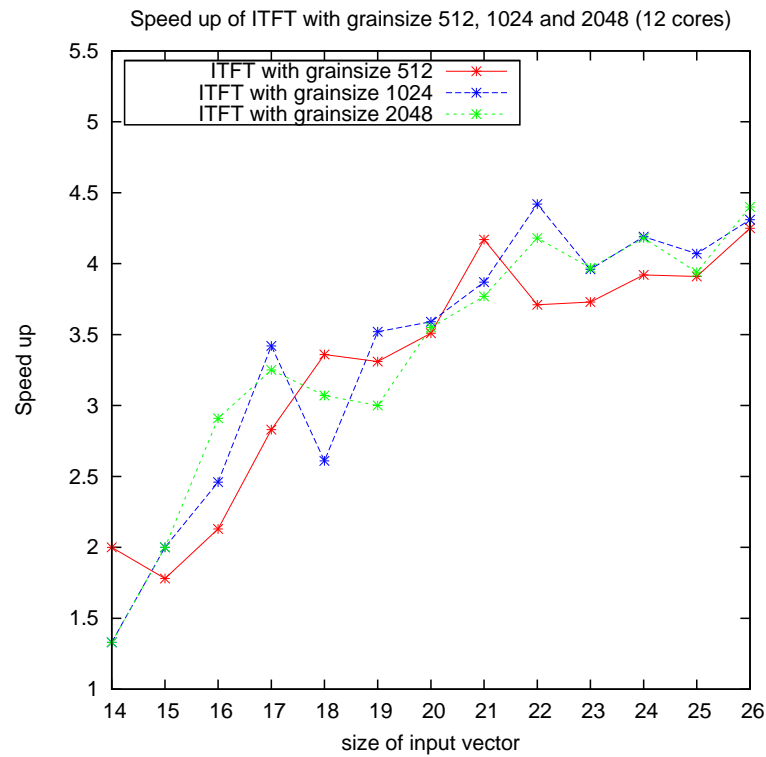


Figure 6.7: Parallel ITFT with different grain sizes.

$\log_2(n)$	Serial FFT	Serial TFT	Parallel TFT	$\frac{\text{SerialFFT}}{\text{ParallelTFT}}$	$\frac{\text{SerialTFT}}{\text{ParallelTFT}}$
14	0.001	0.002	0.003	0.333	0.667
15	0.002	0.004	0.002	1.000	2.000
16	0.004	0.005	0.004	1.000	1.250
17	0.008	0.013	0.004	2.000	3.250
18	0.016	0.022	0.005	3.200	4.400
19	0.036	0.046	0.01	3.600	4.600
20	0.076	0.099	0.016	4.750	6.188
21	0.164	0.249	0.037	4.432	6.730
22	0.355	0.489	0.072	4.931	6.792
23	0.764	0.993	0.144	5.306	6.896
24	1.568	2.001	0.301	5.209	6.648
25	3.128	4.493	0.707	4.424	6.355
26	6.645	8.773	1.294	5.135	6.780

Table 6.5: Running time (secs) for serial FFT, serial TFT and parallel TFT with grain size of 1024 on 12 cores) and the speedup between serial FFT and parallel TFT and between serial TFT and parallel TFT.



# Chapter 7

## Conclusion

In this thesis, we have reported on an implementation of the relaxed general radix forward TFT and a strict general radix inverse TFT. We have obtained a software tool written in Python that generates optimized serial C/C++ code as well as parallel CilkPlus code for forward and inverse TFT, extending a previous work dedicated to FFT code generation within the BPAS library. We have compared the practical efficiency of the strict and relaxed general radix schemes.

Our experimental results show that, in practice, the relaxed general radix forward TFT can reach similar performance (in terms of running time, clock cycles and cache misses) to the optimized FFT code of the BPAS library [4] on input vectors on which both codes apply without zero padding.

Moreover, for an input vector whose size ranges between two consecutive values for which FFT does not require zero padding, our relaxed TFT generated code provides an effective implementation. Unfortunately, the same satisfactory observation does not hold for the strict radix scheme when comparing the inverse TFT and FFT. With respect to parallelization, here also the relaxed general radix scheme is satisfactory while the strict general radix is not. W.r.t. to the FFT code, the parallel forward TFT code has a speedup factor of 5.31 and 6.78 for an input vector of size  $2^{23}$  and  $2^{26}$  respectively.

As for future work, we plan to implement a Python code generator for a relaxed inverse TFT. Moreover, based on our experience with the strict and relaxed schemes of TFT, we believe that it would be valuable to enhance an existing model of concurrent computations so as to better take data flow complexity into account.

# Bibliography

- [1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, 1974.
- [2] L. A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5:78–101, 1966.
- [3] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.
- [4] C. Chen, S. Covanov, F. Mansouri, M. M. Maza, N. Xie, and Y. Xie. The basic polynomial algebra subprograms. In *Mathematical Software–ICMS 2014*, pages 669–676. Springer, 2014.
- [5] J. Cooley and J. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.
- [6] J. Cooley and J. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comp.*, 19:297–301, 1965.
- [7] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other cilk++ hyperobjects. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 79–90, New York, NY, USA, 2009. ACM.
- [8] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th annual symposium on foundations of computer science*, FOCS '99, pages 285 – 297, 1999.
- [9] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms*, 8(1):4, 2012.
- [10] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, May 1998.

- [11] D. Hankerson, S. Vanstone, and A. Menezes. Finite field arithmetic. *Guide to Elliptic Curve Cryptography*, pages 25–73, 2004.
- [12] D. Harvey. A cache-friendly truncated FFT. *Theor. Comput. Sci.*, 410(27-29):2649–2658, 2009.
- [13] Y. He, C. E. Leiserson, and W. M. Leiserson. The Cilkview scalability analyzer. In *Proc. of SPAA*, pages 145–156, 2010.
- [14] J. W. Hong and H. T. Kung. I/o complexity: The red-blue pebble game. In *STOC*, pages 326–333. ACM, 1981.
- [15] S. G. Johnson and M. Frigo. A modified split-radix fft with fewer arithmetic operations. *Signal Processing, IEEE Transactions on*, 55(1):111–119, 2007.
- [16] C. E. Leiserson. The cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.
- [17] X. Li, M. Moreno Maza, R. Rasheed, and É. Schost. The modpn library: Bringing fast polynomial arithmetic into maple. *J. Symb. Comput.*, 46(7):841–858, July 2011.
- [18] Farnam Mansouri. On the parallelization of integer polynomial multiplication. 2014.
- [19] M. Moreno Maza and Y. Xie. Fft-based dense polynomial arithmetic on multi-cores. In *HPCS*, volume 5976 of *Lecture Notes in Computer Science*, pages 378–399. Springer, 2009.
- [20] M. Moreno Maza and Y. Xie. Balanced dense polynomial multiplication on multi-cores. *Int. J. Found. Comput. Sci.*, 22(5):1035–1055, 2011.
- [21] L. Meng and J. R. Johnson. High performance implementation of the inverse TFT. In Jean-Guillaume Dumas, Erich L. Kaltofen, and Clément Pernet, editors, *Proceedings of the 2015 International Workshop on Parallel Symbolic Computation, PASCO 2015, Bath, United Kingdom, July 10-12, 2015*, pages 87–94. ACM, 2015.
- [22] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.
- [23] M. Moreno Maza and W. Pan. Fast polynomial arithmetic on a gpu. *J. of Physics: Conference Series*, 256, 2010.
- [24] M. Moreno Maza and Y. Xie. Balanced dense polynomial multiplication on multi-cores. *International Journal of Foundations of Computer Science.*, 22(5), 2011.

- [25] T. G. Stockham, Jr. High-speed convolution and correlation. In *AFIPS '66 (Spring): Proceedings of the April 26-28, 1966, Spring joint computer conference*, pages 229–233, New York, NY, USA, 1966. ACM.
- [26] J. van der Hoeven. The truncated fourier transform and applications. In Jaime Gutierrez, editor, *ISSAC*, pages 290–296. ACM, 2004.

# Appendix A

## Python Script

```
def generate_everything(file,p,H,num):
    p_1 = p-1
    Npow = 0
    while (p_1%(1<<Npow)==0):
        Npow=Npow+1
    Npow = Npow - 1
    c = p_1>>Npow
    Rpow = int(math.log(p,2)+1)
    c_sft = c<<Npow
    (r,v,u)= extended_gcd(2**Rpow,p)
    y = u%(2**Rpow)
    x = ((u-y)/(2**Rpow))*p
    if y>0:
        y = y-2**Rpow
        x = x+p

    header = open("../..../include/ModularPolynomial/src/"+file+".h","w")
    header.write("#include <math.h>\n")
    header.write("#include <algorithm>\n")
    header.write("#include \"modpn.h\"\n")
    header.write("#ifndef TFTSPE%i\n"%num)
```

Figure A.1: Python code.