4-24-2014 12:00 AM

On The Parallelization Of Integer Polynomial Multiplication

Farnam Mansouri
The University of Western Ontario

Supervisor
Dr. Marc Moreno Maza
The University of Western Ontario

Graduate Program in Computer Science
A thesis submitted in partial fulfillment of the requirements for the degree in Master of Science

On The Parallelization Of Integer Polynomial Multiplication

(Thesis format: Monograph)

by

Farnam Mansouri

Graduate Program in Computer Science

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

Abstract

With the advent of hardware accelerator technologies, multi-core processors and GPUs, much effort for taking advantage of those architectures by designing parallel algorithms has been made. To achieve this goal, one needs to consider both algebraic complexity and parallelism, plus making efficient use of memory traffic, cache, and reducing overheads in the implementations.

Polynomial multiplication is at the core of many algorithms in symbolic computation such as real root isolation which will be our main application for now.

In this thesis, we first investigate the multiplication of dense univariate polynomials with integer coefficients targeting multi-core processors. Some of the proposed methods are based on well-known serial classical algorithms, whereas a novel algorithm is designed to make efficient use of the targeted hardware. Experimentation confirms our theoretical analysis.

Second, we report on the first implementation of subproduct tree techniques on many-core architectures. These techniques are basically another application of polynomial multiplication, but over a prime field. This technique is used in multi-point evaluation and interpolation of polynomials with coefficients over a prime field.

Keywords. Parallel algorithms, High Performance Computing, multi-core machines, 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 Dr. Sardar Anisul Haque, Ning Xie, Dr. Yuzhen Xie, and Dr. Changbo Chen for working along with me and helping me complete this research work successfully. Many thanks to Dr. Jürgen Gerhard of Maplesoft for his help and useful guidance during my internship. In addition, thanks to Svyatoslav Covanov 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

List of Algorithms

List of Tables

List of Figures

Chapter 1

Introduction

Polynomial multiplication and matrix multiplication are at the core of many algorithms in symbolic computation. Expressing, in terms of multiplication time, the algebraic complexity of an operation like univariate polynomial division or the computation of a characteristic polynomial is a standard practice, see for instance the landmark book [29]. At the software level, the motto "reducing everything to multiplication" [1] is also common, see for instance the computer algebra systems Magma[2] [6], NTL[3] or FLINT[4].

With the advent of hardware accelerator technologies, multi-core processors and Graphics Processing Units (GPUs), this reduction to multiplication is, of course, still desirable, but becomes more complex since both algebraic complexity and parallelism need to be considered when selecting and implementing a multiplication algorithm. In fact, other performance factors, such as cache usage or CPU pipeline optimization, should be taken into account on modern computers, even on single-core processors. These observations guide the developers of projects like SPIRAL[5] [57] or FFTW[6] [20].

In this thesis, we investigate the parallelization of polynomial multiplication on both multi-core processors and many-core GPUs. In the former case, we consider dense polynomial multiplication with integer coefficients. The parallelization of this operation was recognized as a major challenge for symbolic computation during the 2012 edition of the *East Coast Computer Algebra Day*[7]. A first difficulty comes from the fact that, in a computer memory, dense polynomials with arbitrary-precision coefficients cannot be represented by a segment of contiguous memory locations. A second difficulty follows from

[1] Quoting a talk title by Allan Steel, from the Magma Project.
[2] Magma: `http://magma.maths.usyd.edu.au/magma/`
[3] NTL: `http://www.shoup.net/ntl/`
[4] FLINT: `http://www.flintlib.org/`
[5] `http://www.spiral.net/`
[6] `http://www.fftw.org/`
[7] `https://files.oakland.edu/users/steffy/web/eccad2012/`

the fact that the fastest serial algorithm for dense polynomial multiplication are based on Fast Fourier Transforms (FFT) which, in general, is hard to parallelize on multi-core processors.

In the case of GPUs, our study shifts to parallel dense polynomial multiplication over finite fields and we investigate how this operation can be integrated into high-level algorithm, namely those based on the so-called *subproduct tree techniques*. The parallelization of these techniques is also recognized as a challenge that, before our work and to the best of our knowledge, has never been handled successfully. One reason is that the polynomial products involved in the construction of a subproduct tree cover a wide range of sizes, thus, making a naive parallelization hard, unless each of these products is itself computed in a parallel fashion. This implies having efficient parallel algorithms for multiplying small size polynomials as well as large ones. This latter constraint has been realized on GPUs and is reported in a series of papers [53, 36].

We note that the parallelization of *sparse* (both univariate and multivariate) polynomial multiplication on both multi-core processors and many-core GPUs has already been studied by Gastineau & Laskard in [26, 27, 25], and by Monagan & Pearce in [51, 50]. Therefore, throughout this thesis, we focus on dense polynomials.

For multi-core processors, the case of modular coefficients was handled in [46, 47] by techniques based on multi-dimensional FFTs. Considering now integer coefficients, one can reduce to the univariate situation via Kronecker's substitution, see for instance the implementation techniques proposed by Harvey in [39]. Therefore, we concentrate our efforts on the univariate case.

1.1 Integer polynomial multiplication on multi-core

A first natural parallel solution for multiplying univariate integer polynomials is to consider divide-and-conquer algorithms where arithmetic counts are saved thanks to the use of evaluation and interpolation techniques. Well-know instances of this solution are the multiplication algorithms of Toom & Cook, among which Karatsuba's method is a special case. As we shall see with the experimental results of Section 3.6, this is a practical solution. However, the parallelism is limited by the number of *ways* in the recursion. Moreover, increasing the number of ways makes implementation quite complicated, see the work by Bodrato and Zanoni for the case of integer multiplication [5, 69, 70]. As in their work, our implementation includes the 4-way and 8-way cases. In addition, we will see in Section 3 that the algebraic complexity of a k-way Toom-Cook algorithm is not in the desirable complexity class of algorithms based on FFT techniques (which is in

$O(N \log N)$ for the input size N).

Turning our attention to this latter class, we first considered combining Kronecker's substitution (so as to reduce multiplication in $\mathbb{Z}[x]$ to multiplication in $\mathbb{Z}$) and the algorithm of Schönhage & Strassen [58]. The GMP-library[8] provides indeed a highly optimized implementation of this latter algorithm [30]. Despite of our efforts, we could not obtain much parallelism from the Kronecker substitution part of this approach. It became clear at this point that, in order to go beyond the performance (in terms of arithmetic count and parallelism) of our parallel 8-way Toom-Cook code, our multiplication code had to rely on a parallel implementation of FFTs. These attempts to obtain an efficient parallel algorithmic solution for dense polynomial multiplication over $\mathbb{Z}$, from serial algorithmic solutions, such as 8-way Toom-Cook, are reported in Chapter 3.1.

Based on the work of our colleagues from the SPIRAL and FFTW projects, and based on our experience on the subject of FFTs [46, 47, 49], we know that an efficient way to parallelize FFTs on multi-core architectures is the so-called *row-column* algorithms[9] which implies to view 1-D FFTs as multi-dimensional FFTs and thus abandon the approach of Schönhage & Strassen.

Reducing polynomial multiplication in $\mathbb{Z}[y]$ to multi-dimensional FFTs over a finite field, say $\mathbb{Z}/p\mathbb{Z}$, implies transforming integers to polynomials over $\mathbb{Z}/p\mathbb{Z}$. As we shall see in Section 4.5, this change of data representation can contribute substantially to the overall running time. Therefore, we decided to invest implementation efforts in that direction. We refer the reader to our publicly available code[10].

In Chapter 4, we propose an FFT-based algorithm for multiplying dense polynomials with integer coefficients in a parallel fashion, targeting multi-core processor architectures. This algorithm reduces univariate polynomial over $\mathbb{Z}$ to 2-D FFT over a finite field of the form $\mathbb{Z}/p\mathbb{Z}$. This addresses the performance issues raised above. In addition, in our algorithm, the transformations between univariate polynomials over $\mathbb{Z}$ and 2-D arrays over $\mathbb{Z}/p\mathbb{Z}$ require only machine word addition and shift operation. Thus, our algorithm does not require to multiply integers at all. Our experimental results show that, for sufficiently large input polynomials, on sufficiently many cores, this new algorithm outperforms all other approaches mentioned above as well as the parallel dense polynomial multiplication over $\mathbb{Z}$ implemented in FLINT. This new algorithm is presented in a paper [10] accepted at the *International Symposium on Symbolic and Algebraic Computation* (ISSAC 2014)[11]. This is a joint work with Changbo Chen, Marc Moreno

[8] `https://gmplib.org/`

[9] `http://en.wikipedia.org/wiki/Fast_Fourier_transform`

[10] BPAS library: `http://www.bpaslib.org/`

[11] `http://www.issac-conference.org/2014/`

Maza, Ning Xie and Yuzhen Xie. Our code is part of the *Basic Polynomial Algebra Subprograms* (BPAS)[12].

1.1.1 Example

We illustrate below the main ideas of this new algorithm for multiplying dense polynomials with integer coefficients. Consider the following polynomials $a, b \in \mathbb{Z}[y]$:

$$\begin{aligned} a(y) &= 100\,y^8 - 55\,y^7 + 217\,y^6 + 201\,y^5 - 102\,y^4 + 225\,y^3 - 127\,y^2 + 84\,y + 40 \\ b(y) &= -26\,y^8 - 85\,y^7 - 110\,y^6 + 9\,y^5 - 114\,y^4 + 51\,y^3 - y^2 + 152\,y + 104 \end{aligned}$$

We observe that each of their coefficients has a bit size less that than $N = 10$. We write $N = KM$ with $K = 2$ and $M = 5$, and define $\beta = 2^M = 32$, such that the following bivariate polynomials $A, B \in \mathbb{Z}[x, y]$ satisfying $a(y) = A(\beta, y)$ and $b(y) = B(\beta, y)$. In other words, we chop each coefficient of a, b into K limbs. We have:

$$\begin{aligned} A(x,y) &= (3\,x+4)\,y^8 + (-x-23)\,y^7 + (6\,x+25)\,y^6 + (6\,x+9)\,y^5 + (-3\,x-6)\,y^4 + \\ &\quad (7\,x+1)\,y^3 + (-3\,x-31)\,y^2 + x + (2\,x+20)\,y + 8 \end{aligned}$$

$$\begin{aligned} B(x,y) &= -26\,y^8 + (-2\,x-21)\,y^7 + (-3\,x-14)\,y^6 + 9\,y^5 + (-3\,x-18)\,y^4 + \\ &\quad (x+19)\,y^3 - y^2 + 3\,x + (4\,x+24)\,y + 8 \end{aligned}$$

Then we consider the convolutions

$$C^-(x,y) \equiv A(x,y)\,B(x,y) \mod \langle x^K - 1 \rangle \text{ and } C^+(x,y) \equiv A(x,y)\,B(x,y) \mod \langle x^K + 1 \rangle.$$

We compute $C^-(x, y)$ and $C^+(x, y)$ modulo a prime number p which is large enough such that the above equations hold both over the integers and modulo p. Working modulo this prime allows us to use FFT techniques. In our example, we obtain:

$$\begin{aligned} C^+(x,y) &= (-104 - 78\,x)\,y^{16} + (-45\,x + 520)\,y^{15} + (-143\,x - 216)\,y^{14} + (-392 - 222\,x)\,y^{13} + \\ &\quad (-623 - 300\,x)\,y^{12} + (-16\,x + 695)\,y^{11} + (83 - 185\,x)\,y^{10} + (38\,x + 510)\,y^9 + \\ &\quad (-476 + 199\,x)\,y^8 + (-441 - 183\,x)\,y^7 + (567\,x + 1012)\,y^6 + (-947 - 203\,x)\,y^5 + \\ &\quad (225 + 149\,x)\,y^4 + (-614 - 112\,x)\,y^3 + (10\,x + 225)\,y^2 + (132\,x + 342)\,y + 32\,x + 61 \end{aligned}$$

[12] `http://www.bpaslib.org/`

$$
\begin{aligned}
C^{-}(x,y) \;=\; & (-104-78\,x)\,y^{16}+(-45\,x+508)\,y^{15}+(-143\,x-230)\,y^{14}+(-410-222\,x)\,y^{13}+\\
& (-701-300\,x)\,y^{12}+(-16\,x+683)\,y^{11}+(35-185\,x)\,y^{10}+(38\,x+480)\,y^{9}+\\
& (-426+199\,x)\,y^{8}+(-463-183\,x)\,y^{7}+(567\,x+1122)\,y^{6}+(-953-203\,x)\,y^{5}+\\
& (261+149\,x)\,y^{4}+(-594-112\,x)\,y^{3}+(10\,x+223)\,y^{2}+(132\,x+362)\,y+32\,x+67
\end{aligned}
$$

Now we observe that the following holds, as a simple application of the Chinese Remaindering Theorem:

$$
A(x,y)B(x,y) = C(x,y) = \frac{C^{+}(x,y)}{2}\,(x^{K}-1) + \frac{C^{-}(x,y)}{2}\,(x^{K}+1),
$$

where, in our case, we have:

$$
\begin{aligned}
C(x,y) \;=\; & (-104-78\,x)\,y^{16}+(514-6\,x^{2}-45\,x)\,y^{15}+(-143\,x-223-7\,x^{2})\,y^{14}+\\
& (-9\,x^{2}-222\,x-401)\,y^{13}+(-39\,x^{2}-662-300\,x)\,y^{12}+(689-6\,x^{2}-16\,x)\,y^{11}+\\
& (-24\,x^{2}+59-185\,x)\,y^{10}+(-15\,x^{2}+495+38\,x)\,y^{9}+(-451+199\,x+25\,x^{2})\,y^{8}+\\
& (-183\,x-11\,x^{2}-452)\,y^{7}+(1067+55\,x^{2}+567\,x)\,y^{6}+(-3\,x^{2}-950-203\,x)\,y^{5}+\\
& (149\,x+243+18\,x^{2})\,y^{4}+(-604-112\,x+10\,x^{2})\,y^{3}+(-x^{2}+10\,x+224)\,y^{2}+\\
& (352+132\,x+10\,x^{2})\,y+64+3\,x^{2}+32\,x
\end{aligned}
$$

Finally, by evaluating $C(x,y)$ at $x=\beta=32$, the final result is:

$$
\begin{aligned}
c(y) \;=\; & -2600\,y^{16}-7070\,y^{15}-11967\,y^{14}-16721\,y^{13}-50198\,y^{12}-5967\,y^{11}\\
& -30437\,y^{10}-13649\,y^{9}+31517\,y^{8}-17572\,y^{7}+75531\,y^{6}-10518\,y^{5}\\
& +23443\,y^{4}+6052\,y^{3}-480\,y^{2}+14816\,y+4160
\end{aligned}
$$

We stress the fact, that, in our implementation, the polynomial $C(x,y)$ is actually not computed at all. Instead, the polynomial $c(y)$ is obtained directly from the convolutions $C^{-}(x,y)$ and $C^{+}(x,y)$ by means of byte arithmetic. In fact, as mentioned above, our algorithm performs only computations in $\mathbb{Z}/p\mathbb{Z}$ and add/shift operations on byte vectors. Thus we avoid manipulating arbitrary-precision integers and ensure that all data sets that we generate are in contiguous memory location. As a consequence, and thanks to the 2-D FFT techniques that we rely on, we prove that the cache complexity of our algorithm is optimal.

1.2 Polynomial evaluation and interpolation on many-core

In the rest of this thesis, we investigate the use of Graphics Processing Units (GPUs) in the problems of evaluating and interpolating polynomials by means of subproduct tree techniques. Many-core GPU architectures were considered in [61] and [64] in the case of numerical computations, with a similar purpose as ours, as well as the long term goal of obtaining better support, in terms of accuracy and running times, for the development of polynomial system solvers.

Our motivation is also to improve the performance of polynomial system solvers. However, we are targeting symbolic, thus exact, computations. In particular, we aim at providing GPU support for solvers of polynomial systems with coefficients in finite fields, such as the one reported in [54]. This case handles problems from cryptography and serves as a base case for the so-called modular methods [16], since those methods reduce computations with integer number coefficients to computations with finite field coefficients.

Finite fields allow the use of asymptotically fast algorithms for polynomial arithmetic, based on FFTs or, more generally, subproduct tree techniques. Chapter 10 in the landmark book [28] is an overview of those techniques, which have the advantage of providing a more general setting than FFTs. More precisely, evaluation points do not need to be successive powers of a primitive root of unity. Evaluation and interpolation based on subproduct tree techniques have "essentially" (i.e. up to log factors) the same algebraic complexity estimates as their FFT-based counterparts. However, and as mentioned above, their implementation is known to be challenging.

In Chapter 5.1, we report on the first GPU implementation (using CUDA [55]) of subproduct tree techniques for multi-point evaluation and interpolation of univariate polynomials. In this context, we demonstrate the importance of *adaptive algorithms*. That is, algorithms that adapt their behavior to the available computing resources. In particular, we combine *parallel plain arithmetic* and *parallel fast arithmetic*. For the former we rely on [36] and, for the latter we extend the work of [53]. All implementation of subproduct tree techniques that we are aware of are serial only. This includes [9] for $GF(2)[x]$, the FLINT library[38] and the `Modpn` library [44]. Hence we compare our code against probably the best serial C code (namely the FLINT library) for the same operations. For sufficiently large input data and on NVIDIA Tesla C2050, our code outperforms its serial counterpart by a factor ranging between 20 to 30. This is a joint work with Sardar Anisul Haque and Marc Moreno Maza. Our code is part of the *CUDA*

Modular Polynomial (CUMODP)[13].

1.2.1 Example

We illustrate below an example of evaluating a polynomial with the degree 7 at 8 points. Here is the given polynomial over the prime filed 257:

$$P(x) \;=\; 92\,x^7 + 89\,x^6 + 24\,x^5 + 82\,x^4 + 170\,x^3 + 179\,x^2 + 161\,x + 250$$

We want to evaluate it at the points $(63, 100, 148, 113, 109, 26, 39, 206)$.

The corresponding subproduct tree which will be constructed using a bottom-up approach is illustrated in Figure 1.1.

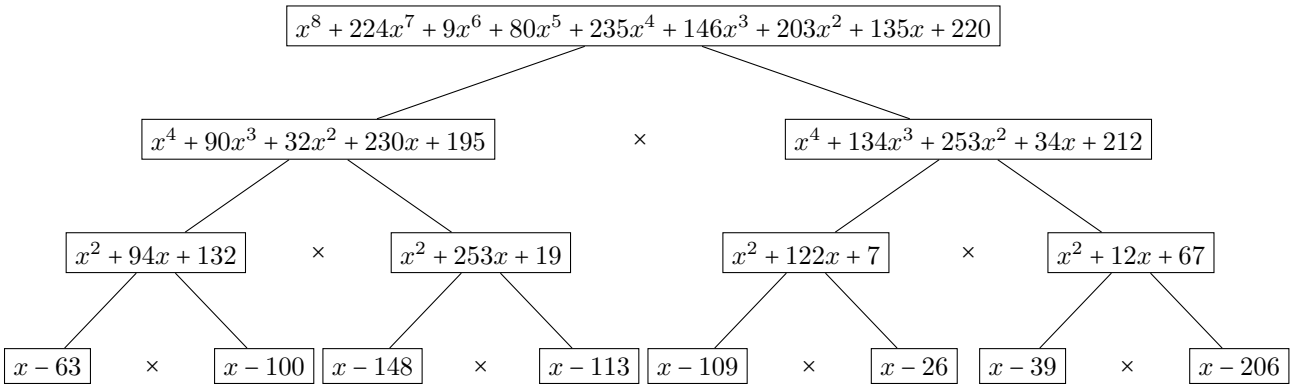


Figure 1.1: Subproduct tree for evaluating a polynomials of degree 7 at 8 points.

Then, we start the top-down approach for evaluating the polynomial in which, first, we need to compute the remainder of $P(x)$ over two children of the root of the subproduct tree, and evaluate each of the results at 4 points, and so on (do this recursively). In Figure 1.2, this remaindering process is shown.

The final results of the remaindering process will be the evaluated results of the polynomial at the given points over the prime field (257 in this case).

[13] `http://www.cumodp.org/`

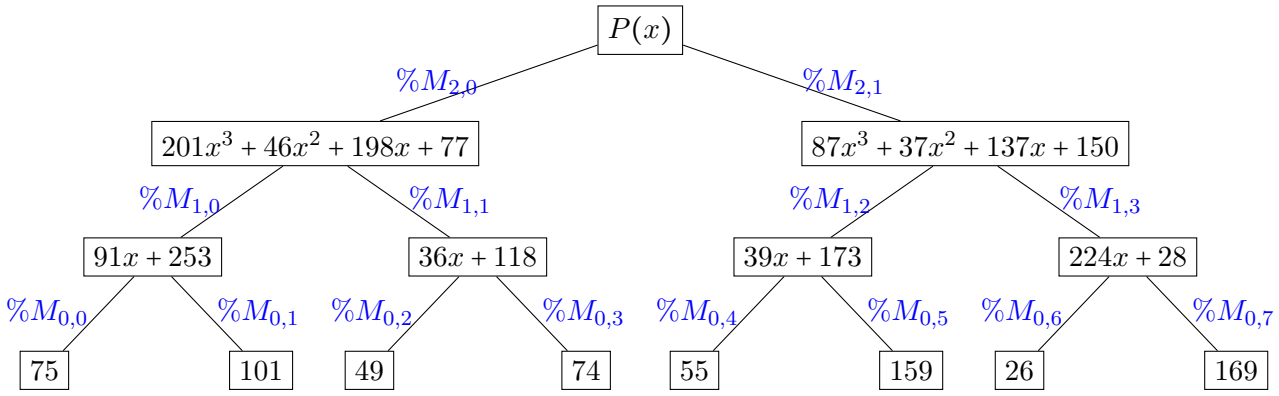


Figure 1.2: Top-down remaindering process associated with the subproduct tree for evaluating the example polynomial. The % symbol means mod operation. $M_{i,j}$ is the j-th polynomial at level i of the subproduct tree.

Chapter 2

Background

In this chapter, we review basic concepts related to high performance computing. We start with multi-core processors and the fork-join concurrency model. We continue with the *ideal cache model* since cache complexity plays an important role for our algorithms targeting multi-core processors. Then, we give an overview of GPGPU (general-purpose computing on graphics processing units) followed by a model of computation devoted to these many-core GPUs. We conclude this chapter by a presentation of FFT-based algorithms, stated for vectors with coefficients in finite fields. Indeed, the algorithms of Cooley-Tukey, Stockham, and Schönhage & Strassen play an essential role in our work.

2.1 Multi-core processors

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 addition, because of the physical limitations of increasing the frequency of processors, or designing more complex integrated circuits, most of the recent improvements were 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 others.

In a multi-core system, we may have multi-level cache memories which can have a

huge impact on the performance. Having cache memories on each of the processors, gives the programmers an opportunity of designing extremely fast memory access procedures. Implementing a program which can take benefits from the cache hierarchy, with low cache misses rates is known to be challenging.

There are numerous parallel programming models based on these architectures. There are some challenges whether in the programming models or in the application development layers. For instance, how to divide and partition the task, how to make use of cache and memory hierarchy, how to distribute and manage the tasks, how the tasks can communicate with each-other, what is the memory access for each task. Some of these worries will be handled in the concurrent programming platform, and some need to be handled by the developer. Some well-known examples of these concurrent programming models are CilkPlus [1], OpenMP [2], MPI [3], etc.

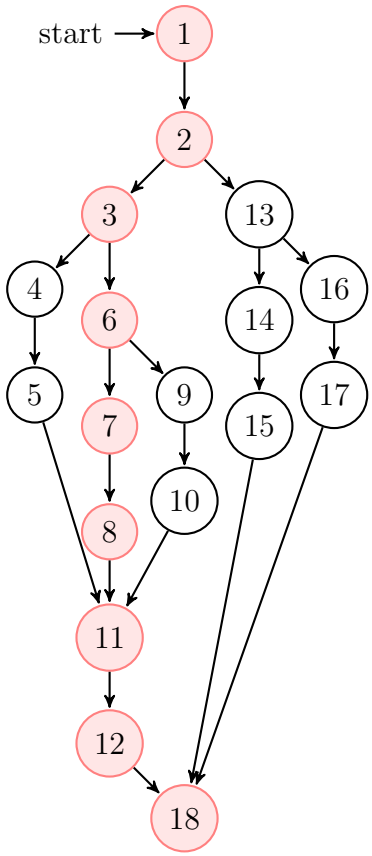
2.1.1 Fork-join parallelism model

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 vertices 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 will end and one new strand begins.

[1] `http://www.cilkplus.org/`
[2] `http://openmp.org/wp/`
[3] `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 to two threads 3 and 13. Following the the division of the program, the threads 15, 17 and 12 will be joined to 18.

CilkPlus is a C++based platform providing an implementation of this model [43, 23, 19] using *work-stealing* scheduling [3] in which every processor has a stack of tasks, and all of the processors can steal tasks from others' stacks when they are idle. In CilkPlus extension, one can use the keywords cilk_spawn to fork, and cilk_sync for join. According to theory analysis, this framework has minimal overhead for tasks scheduling; this helps the developers to exploit their applications to the maximum parallelism.



For analyzing the parallelism in the fork-join model, we measure T_1 and T_∞ 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_∞): 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 called the *critical path* too.

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) is 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_∞).

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: $T_p \leq T_1/p + T_\infty$.

Programming in Cilkplus

Here is an example of programming in Cilkplus for transposing a given matrix:

```
void transpose(T *A, int lda, T *B, int ldb, int i0, int i1, int j0, int j1){
  tail:
    int di = i1 - i0, dj = j1 - j0;
    if (di >= dj && di > THRESHOLD) {
        int im = (i0 + i1) / 2;
        cilk_spawn transpose(A, lda, B, ldb, i0, im, j0, j1);
        i0 = im; goto tail;
    } else if (dj > THRESHOLD) {
        int jm = (j0 + j1) / 2;
        cilk_spawn transpose(A, lda, B, ldb, i0, i1, j0, jm);
        j0 = jm; goto tail;
    } else {
        for (int i = i0; i < i1; ++i)
            for (int j = j0; j < j1; ++j)
                B[j * ldb + i] = A[i * lda + j];
    }
}
```

In this implementation, we divide the problem into two subproblems based on the input sizes. Then, the subproblems will be forked and executed in parallel. Note that when the size of the problem is small enough, we execute the serial method; the THRESHOLD would be decided based on the cache line size for which we make sure that the input and output data would fit into the cache.

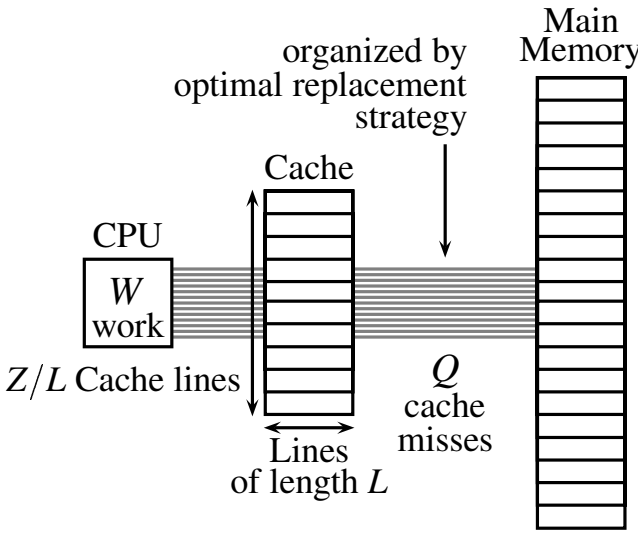


Figure 2.1: The ideal-cache model.

2.2 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. This idea was first introduced by Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran in 1999 [21]. 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 *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 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*

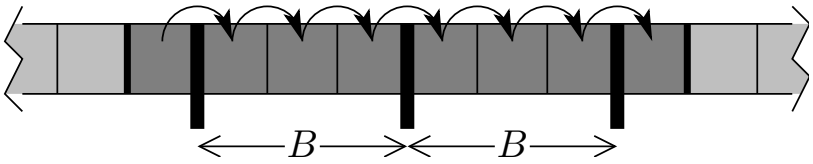


Figure 2.2: Scanning an array of $n = N$ elements, with $L = B$ words per cache line.

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. 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 runtime, 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 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.

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 (See Proposition 1 and its proof), the cache-oblivious bound is an additive 1 away from the external-memory bound [40]. 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) be the total number of words in fully (not fully) used cache lines. Thus, we have $n = u + w$.
- If $w = 0$ 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 $0 < w < L$ then $(q, r) = (\lfloor n/L \rfloor, w)$ 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 $L \leq w < 2L$ then $(q, r) = (\lfloor n/L \rfloor, w - L)$ and the scanning costs exactly $q + 1$; the conclusion is clear again.

2.3 General-purpose computing on graphics processing units

General-purpose computing on graphics processing units (GPGPU) is a way of doing typical computations on Graphical processing units (GPU). The architecture of GPU, has known to be suitable for some kind of computations in which one can achieve highly efficient computing power compared to traditional ways of computing on CPU.

The architecture of a typical GPU, consists of streaming multiple-processors (*SM*) which have multiple (8, typically) streaming processors which are SIMD (single instruction, multiple data) processors targeted for executing light threads. In a SM, there is a shared memory which is accessible by each of the streaming processors. In addition, each of the streaming processors have their own local registers. Each of the *SM*s will be targeted for executing a block of threads.

There are two dominant platforms for programming GPUs: OpenCL [4] and CUDA [5] (from *Nvidia*). Here we investigate (and later use) CUDA.

2.3.1 CUDA

CUDA is a parallel programming architecture and model created by NVIDIA, which is a C/C++ extension providing specific instruction sets for programming GPUs. It naively supports multiple computational interfaces such as standard languages and APIs, but having low overheads. It also provides accessing different hierarchy of the memory.

The CUDA programming model consists of the *host* which is a traditional CPU, and one or more computing devices that are massively data-parallel co-processors (GPUs).

[4] http://www.khronos.org/opencl/

[5] http://www.nvidia.com/object/cuda_home_new.html

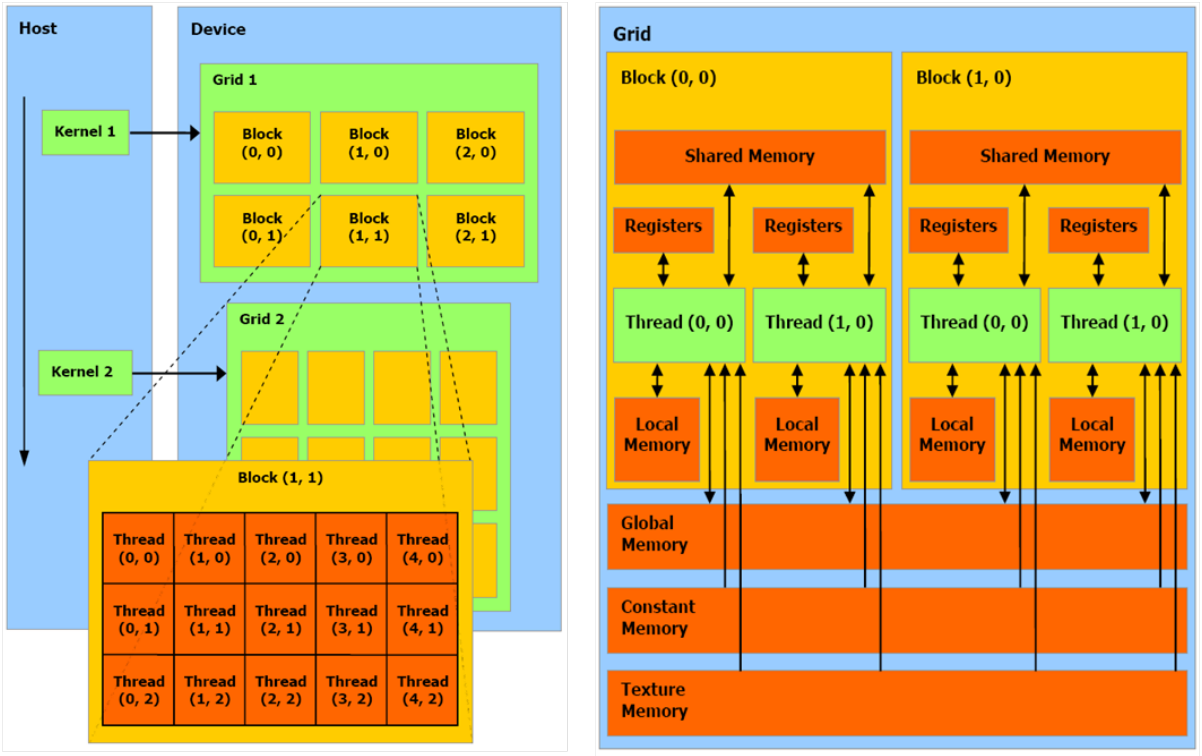


Figure 2.3: Illustration of the CUDA memory hierarchy [56]

Each device is equipped with a large number of arithmetic execution units that has its own DRAM, and runs many threads in parallel.

To invoke calculations on the GPU one has to perform a kernel launch, which is basically a function written with the intent of what each thread on the GPU is to perform. The GPU has a specific architecture of threads where they are divided into blocks and where blocks are divided into a grid, see Figure 2.3. The grid has two dimensions and can contain up to 65536 blocks in each dimension. While each block contains threads in three dimensions, and can contain up to 512 threads in two dimensions and 64 in the third. When executing a kernel one specifies the dimensions of the grid and blocks to specify how many threads will be executing the kernel.

GPU has its own DRAM which is used in communicating with the host system. To accelerate calculations within the GPU itself there are several other layers of memory such as constant, shared and texture. Table 2.1 shows the relation between these.

Programming in CUDA

Here is an example of programming in CUDA for transposing of a given matrix:

```
__global__ void transpose(float *odata, float *idata, int width, int height) {
```

Memory	Location	Cached	Access	Scope in Architecture
Register	On-chip	No	Read/Write	Single thread
Local	Off-chip	No	Read/Write	Single thread
Shared	On-chip	No	Read/Write	Threads in a Block
Global	On-chip	No	Read/Write	All
Constant	On-chip	Yes	Read	All
Texture	On-chip	Yes	Read	All

Table 2.1: Table showing CUDA memory hierarchy [56]

```
    __shared__ float tile[TILE_DIM][TILE_DIM];

    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
    int index_in = xIndex + yIndex * width;
    xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
    yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
    int index_out = xIndex + yIndex * height;

    for (int i = 0; i < TILE_DIM; i += BLOCK_ROWS) {
        tile[threadIdx.y + i][threadIdx.x] = idata[index_in + i*width];
    }
    __syncthreads();

    for (int i = 0; i < TILE_DIM; i += BLOCK_ROWS) {
        odata[index_out + i*height] = tile[threadIdx.x][threadIdx.y + i];
    }
}
```

The __global__ identifies that this function is a kernel which means it will be executed on the GPU. The host will invoke this kernel by specifying the number of thread blocks, and number of threads per block. Then, each of the threads will execute this function having access to the shared memory allocated on the thread block and the global memory. Note that each thread has different *id* (see threadIdx.x and threadIdx.y), as each thread block has different *id* (see blockIdx.x and blockIdx.y).

In this implementation, tile is the allocated memory on the shared memory. Then, in the first loop, we copy the elements of the matrix from global memory to the shared

memory. After synchronizing, which means to make sure that all of the memory-reading are completed, we copy the elements of the transposed matrix from the shared memory to the correct index in the global memory. So, each of the thread blocks are responsible for transposing a tile of the matrix.

2.4 Many-core machine model

Many-core Machine Model (MMM) is a model of multi-threaded computation, combining fork-join and single-instruction-multiple-data parallelisms, with an emphasis on estimating parallelism overheads of programs written for modern many-core architectures [35]. Using this model, one can minimize parallelism overheads by determining an appropriate value range for a given program parameter.

Architecture An MMM abstract machine has a similar architecture as a GPU in which we have infinite and identical number of streaming multiprocessors. Each of the SM has a finite number of processing cores and a fixed-size local memory. Plus, it has 2-level memory hierarchy: one unbounded global memory with high latency and low throughput, and SM local memory having low latency and high throughput.

Program An MMM program is a directed acyclic graph (DAG) whose vertices are kernels and where edges indicate dependencies. A kernel is a SIMD (single instruction multi-threaded data) program decomposed into a number of thread-blocks. Each thread-block is executed by a single SM and each SM executes a single thread-block at a time.

Scheduling and synchronization At run time, an MMM machine schedules thread-blocks onto the SMs, based on the dependencies among kernels and the hardware resources required by each thread-block. Threads within a thread-block cooperate with each other via the local memory of the SM running the thread-block. Thread-blocks interact with each other via the global memory

Memory access policy All threads of a given thread-block can access simultaneously any memory cell of the local memory or the global memory. Read/Write conflicts are handled by the CREW (concurrent read exclusive write) policy.

For the purpose of analyzing program performance, we define two machine parameters:

- **U**: Time (expressed in clock cycles) spent for transferring one machine word between the global memory and the local memory of any SM (so-called shared memory).

- **Z**: Size (expressed in machine words) of the local memory of SM.

Kernel DAG Each MMM program P is modeled by a directed acyclic graph (K, e) , called the kernel DAG of P, where each node represents a kernel, and each edge represents a kernel call which must precede another kernel call. (a kernel call can be executed whenever all its predecessors in the DAG completed their execution)

Since each kernel of the program P decomposes into a finite number of thread-blocks, we map P to a second graph, called the **thread block DAG** of P, whose vertex set $B(P)$ consists of all thread-blocks of the kernels of P, such that (B_1, B_2) is an edge if B_1 is a thread-block of a kernel preceding the kernel of B_2 in P.

2.4.1 Complexity measures

Work The work of a thread-block is defined as the total number of local operations performed by the threads in that block. The work of a kernel k is defined as the sum of the works of its thread-blocks ($W(k)$). The work of an entire program P is defined as $W(P)$ which is defined as the total work of all its kernels:

$$W(P) = \sum_{k \in K} W(k)$$

Span The span of a thread-block is defined as the maximum number of local operations performed by the threads in that block. The span of a kernel k is defined as the maximum span of its thread-blocks ($S(k)$). The span of the path γ if defined as the sum of the span of all kernels in that path: $S(\gamma) = \sum_{k \in \gamma} S(k)$

The span of an entire program P is defined as:

$$S(P) = \max_{\gamma} \ S(\gamma)$$

Overhead The overhead of a thread-block B is defined as $(r+w)\,U$, assuming that each thread of B reads (at-most) r words and writes (at-most) w words to the global memory. The overhead of a kernel k is defined as the sum of the overheads of its thread-blocks ($O(k)$). The overhead of an entire program P is defined as $O(P)$ which is defined as the

total overhead of all its kernels:

$$O(P) = \sum_{\alpha} O(\alpha)$$

Graham-Brent Theorem for MMM We have the following estimate for the running time T_ρ of the program ρ when executing it on p Streaming Multiprocessors:

$$T_p \leq (N(\rho)/p + L(\rho)) \ . \ C(\rho)$$

where $N(\rho)$ is the number of vertices in the thread-block DAG of ρ, $L(\rho)$ is the critical path length (the length of the longest path) in the thread-block DAG of ρ, and $C(\rho) = \max_{B' \ \in \ B(\rho)} (S(B') + O(B'))$.

2.5 Fast Fourier transform over finite fields

Most multiplication algorithm such as Karatsuba, Toom-Cook, and FFT use evaluation-interpolation approach. The fast Fourier transform (FFT) is also based on this approach in which the evaluation and interpolation are done on specific points (roots of unity) which is relatively efficient compared to its counterparts.

Definition let R be a ring, and $K \geq 2 \in \mathbb{N}$, and ω be the Kth root of unity in R; this means that $\omega^K = 1$ and $\sum_{j=0}^{K-1} \omega^{ij} = 0$ for $1 \leq i < K$. The *Fourier transform* of a vector $A = [A_0, \ldots, A_{K-1}]$ from R is $\hat{A} = [\hat{A}_0, \ldots, \hat{A_{K-1}}]$ where for $0 \leq i < K$: $\hat{A}_i = \sum_{j=0}^{K-1} w^{ij} A_j$.

Fourier transform computing $\hat{A}$ in the naive way, takes $O(K^2)$, but using *fast Fourier transform* which is a divide & conquer algorithm, it takes $O(K \log K)$.

Inverse Fourier transform is the reverse operation of *Fourier transform* which computes the vector A by having $\hat{A}$ as an input. The complexity of this operation is also $O(K \log K)$. It can be proven that the inverse transform is the same operation as the transform, but the points of the vector are shuffled.

Multiplication given input polynomials f and g of degree K defined as $f(x) = \sum_{i=0}^{K} f_i x^i$ and $g(x) = \sum_{i=0}^{K} g_i x^i$.

1. Say ω is the $2K$th root of unity. Then, we evaluate both polynomials at points $\omega^0, \ldots, \omega^{2K-1}$ which is equivalent to computing the Fourier transform which was

defined above. This step is called Discrete Fourier Transform.

2. Point-wise multiplication: $(f(\omega^0).g(\omega^0), \ldots, f(\omega^{2K-1}).g(\omega^{2K-1}))$.

3. Interpolating the result polynomial using inverse Fourier transform.

The overall cost of multiplication using FFT algorithm is $O(K \log K)$.

2.5.1 Schönhage-Strassen FFT

The Schönhage-Strassen [58] FFT algorithm which is known to be asymptotically the best multiplication algorithm with complexity $O(n \log n \log\log n)$ (until Fürer's algorithm [24]) works on the ring $\mathbb{Z}/(2^n + 1)\mathbb{Z}$. Algorithm 1 (this algorithm is from [8, Section 2.3]) is the Schönhage-Strassen approach for multiplying two integers having at-most n bits. Note that in the Algorithm 1, if the chosen n' is large, we call the algorithm recursively.

Algorithm 1: SchönhageStrassen

Input: $0 \le A, B < 2^n + 1$, $K = 2^k$, $n = MK$
Output: $C = A.B \mod (2^n + 1)$
$A = \sum_{j=0}^{K-1} a_j 2^{jM}$, $B = \sum_{j=0}^{K-1} b_j 2^{jM}$ where $0 \le a_j, b_j < 2^M$;
choose $n' \ge 2n/K + k$ which is multiple of K;
$\theta = 2^{n'/K}, \omega = \theta^2$;
for $j = 0 \ldots k-1$ **do**
    $a_j = \theta^j a_j \mod (2^{n'} + 1)$;
    $b_j = \theta^j b_j \mod (2^{n'} + 1)$;
$a = \mathsf{FFT}(a, \omega, K)$;
$b = \mathsf{FFT}(b, \omega, K)$;
for $j = 0 \ldots k-1$ **do**
    $c_j = a_j b_j \mod (2^{n'} + 1)$;
$c = \mathsf{InverseFFT}(c, \omega, K)$;
for $j = 0 \ldots k-1$ **do**
    $c_j = c_j/(K\theta^j) \mod (2^{n'} + 1)$;
    if $c_j \ge (j+1)2^{2M}$ **then**
        $c_j = c_j - (2^{n'} + 1)$;
$C = \sum_{j=0}^{K-1} c_j 2^{jM}$;
return C;

2.5.2 Cooley-Tukey and Stockham FFT

This section reviews the Fast Fourier Transform (FFT) in the language of tensorial calculus, see [45] for an extensive presentation. Throughout this section, we denote by $\mathbb{K}$ a field. In practice, this field is often a prime field $\mathbb{Z}/p\mathbb{Z}$ where p is a prime number greater than 2.

Basic operations on matrices

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

$$A \otimes B = [a_{k\ell} B]_{k,\ell} \quad \text{with} \quad A = [a_{k\ell}]_{k,\ell} \tag{2.1}$$

For example, let

$$A = \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}. \tag{2.2}$$

Then their tensor products are

$$A \otimes B = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 2 & 2 & 3 & 3 \\ 2 & 2 & 3 & 3 \end{bmatrix} \quad \text{and} \quad B \otimes A = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 2 & 3 & 2 & 3 \\ 0 & 1 & 0 & 1 \\ 2 & 3 & 2 & 3 \end{bmatrix}. \tag{2.3}$$

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 $\mathbb{K}$.

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

and can be viewed as an opportunity for *vector-parallelism* as illustrated below:

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

The direct sum of A and B is an $(m+q) \times (n+s)$ matrix over $\mathbb{K}$ denoted by $A \oplus B$ and defined by

$$A \oplus B = \begin{bmatrix} A & 0 \\ 0 & B \end{bmatrix}. \tag{2.4}$$

More generally, for n matrices $A_0, \ldots, A_{n-1}$ over $\mathbb{K}$, the direct sum of $A_0, \ldots, A_{n-1}$ is defined as $\oplus_{i=0}^{n-1} A_i = A_0 \oplus (A_1 \oplus (\cdots \oplus A_{n-1}) \cdots)$. The stride permutation matrix L_m^{mn} permutes an input vector $\mathbf{x}$ of length mn as follows

$$\mathbf{x}[im + j] \mapsto \mathbf{x}[jn + i], \tag{2.5}$$

for all $0 \le j < m$, $0 \le i < n$. If $\mathbf{x}$ is viewed as an $n \times m$ matrix, then L_m^{mn} performs a transposition of this matrix.

Discrete Fourier transform

We fix an integer $n \ge 2$ and an n-th primitive root of unity $\omega \in \mathbb{K}$. The n-point Discrete Fourier Transform (DFT) at ω is a linear map from the $\mathbb{K}$-vector space $\mathbb{K}^n$ to itself, defined by $\mathbf{x} \longmapsto \mathrm{DFT}_n \mathbf{x}$ with the n-th DFT matrix

$$\mathrm{DFT}_n = [\omega^{k\ell}]_{0 \le k, \ell < n}. \tag{2.6}$$

In particular, the DFT of size 2 corresponds to the butterfly matrix

$$\mathrm{DFT}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \tag{2.7}$$

The well-known Cooley-Tukey Fast Fourier Transform (FFT) [14] in its recursive form is a procedure for computing $\mathrm{DFT}_n \mathbf{x}$ based on the following factorization of the matrix

DFT_n, for any integers q, s such that $n = qs$ holds:

$$\mathrm{DFT}_{qs} = (\mathrm{DFT}_q \otimes I_s) D_{q,s} (I_q \otimes \mathrm{DFT}_s) L^{qs}_q, \tag{2.8}$$

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

$$D_{q,s} = \bigoplus_{j=0}^{q-1} \mathrm{diag}(1, \omega^j, \ldots, \omega^{j(s-1)}), \tag{2.9}$$

Formula (2.10) illustrates Formula (2.8) with DFT_4:

$$\begin{aligned}
\mathrm{DFT}_4 &= (\mathrm{DFT}_2 \otimes I_2) D_{2,2} (I_2 \otimes \mathrm{DFT}_2) L^2_2 \\
&= \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \omega \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & -1 & -\omega \\ 1 & -1 & 1 & -1 \\ 1 & -\omega & -1 & \omega \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix}.
\end{aligned} \tag{2.10}$$

Assume that n is a power of 2, say $n = 2^k$. Formula (2.8) can be unrolled so as to reduce DFT_n to DFT_2 (or a base case 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

$$\Delta(i, j, h) = (I_i \otimes \mathrm{DFT}_j \otimes I_h) \tag{2.11}$$

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

$$\mathrm{DFT}_{2^k} = \left(\prod_{i=1}^{k-\ell} \Delta\left(2^{i-1}, 2, 2^{k-i}\right) \left(I_{2^{i-1}} \otimes D_{2,2^{k-i}}\right) \right) \Delta\left(2^{k-\ell}, m, 1\right) \left(\prod_{i=k-\ell}^{1} (I_{2^{i-1}} \otimes L^{2^{k-i+1}}_2) \right). \tag{2.12}$$

Therefore, Formula (2.12) reduces the computation of DFT_{2^k} to composing DFT_2, DFT_{2^ℓ}, diagonal twiddle endomorphisms and stride permutations. Another recursive factoriza-

tion of the matrix DFT_{2^k} is

$$\mathrm{DFT}_{2^k} = (\mathrm{DFT}_2 \otimes I_{2^{k-1}}) D_{2,2^{k-1}} L_2^{2^k} (\mathrm{DFT}_{2^{k-1}} \otimes I_2), \tag{2.13}$$

from which one can derive the Stockham FFT [59] as follows

$$\mathrm{DFT}_{2^k} = \prod_{i=0}^{k-1} (\mathrm{DFT}_2 \otimes I_{2^{k-1}})(D_{2,2^{k-i-1}} \otimes I_{2^i})(L_2^{2^{k-i}} \otimes I_{2^i}). \tag{2.14}$$

This is a basic routine which is implemented in our library (CUMODP [6]) as the FFT over a finite field (prime) targeted GPUs [53].

[6] `http://cumodp.org/`

Chapter 3

Parallelizing classical algorithms for dense integer polynomial multiplication

For a given algorithmic problem, such as performing dense polynomial multiplication over a given coefficient ring, there are at least two natural approaches for obtaining an efficient parallel solution. The first one is to start from a good serial solution and parallelize it, if possible. The second one is to start from a good parallel solution of a related algorithmic problem and transform it into a good parallel solution of the targeted problem.

In this chapter, for the question of performing dense polynomial multiplication over the integers, we follow the first approach, reserving the second one for the next chapter. To be more specific, we consider below standard sequential algorithms and discuss their parallelization in `CilkPlus` targeting multi-core systems.

Interpreting experimental performance results of a parallel program is often a challenge, since many phenomena may interfere with each other: the underlying algorithm, its implementation, issues related to the hardware or to the concurrency platform executing the program. Complexity estimates of the underlying algorithms can help with this interpretation. However, it is essential not to limit algorithm analysis to arithmetic count. To this end, we review those algorithms and analyze their algebraic complexity in Sections 3.2 through 3.4. Then, we discuss their parallelization and analyze the corresponding complexity measures in Section 3.5. Experimental results are reported and analyzed in Section 3.6.

This is a joint work with M. Moreno Maza.

3.1 Preliminary results

Notation 1 *We consider two polynomials* $f, g \in \mathbb{Z}[x]$ *written as follows:*

$$f(x) = \sum_{i=0}^{m-1} f_i x^i \ , \ g(x) = \sum_{i=0}^{n-1} g_i x^i \tag{3.1}$$

where $n,\ m \in \mathbb{N}_{>0}$ *and, for* $0 \leq i < m$ *and* $0 \leq j < n$*, we have* $f_i \ , \ g_j \in \mathbb{Z}$ *together with* $|f_i| \leq 2^{b_f}$ *and* $|g_j| \leq 2^{b_g}$*, for some* $b_f, b_g \in \mathbb{N}$*. We want to compute the polynomial* $h \in \mathbb{Z}[x]$ *defined by:*

$$h(x) = \sum_{i=0}^{n+m-2} h_i x^i = f(x) \times g(x). \tag{3.2}$$

Algorithm 2: Schoolbook(f, g, m, n)

Input: $f, g \in \mathbb{Z}[x]$ and $m, n \in \mathbb{N}_{>0}$ as in Notation 1.
Output: $h \in \mathbb{Z}[x]$ such that $h = f.g$.

```
for i = 0...m − 1 do
    for j = 0...n − 1 do
        h_{i+j} = h_{i+j} + f_i . g_j;
return h;
```

One naive solution for computing the product h is the so-called *schoolbook method* (Algorithm 2) which requires $O(nm)$ arithmetic operation on the coefficients. If the coefficients are all of small size in comparison to n and m, then this upper bound can be regarded as a running time estimate, in particular if the coefficients are of machine word size. However, if the coefficients are large, it becomes necessary to take their size into account when estimating the running time of Algorithm 2. This explains the introduction of Assumption 1.

This phenomenon occurs frequently with the univariate polynomials resulting from solving multivariate polynomials systems of equations and inequalities. In such case, the resulting univariate polynomials are often dense in the sense that the bit size of each coefficient is the same order of magnitude as the degree, thus implying $b_f \in \Theta(n)$.

For us, these polynomials are of great interest since isolating their real roots is a key step in the process of solving polynomials systems. At the same time, the size of these coefficients makes real root isolation an extremely expensive task in terms of computing resources, leading to the use of high-performance computing techniques, such as parallel processing [11] and asymptotically fast algorithms [65].

Assumption 1 *The implementation of all the algorithms studied in this work relies on the* GNU Multi-Precision *(GMP) library* [1]. *The purpose of Proposition 2 is to give an asymptotic upper bound for GMP's integer multiplication. We will use this estimate in the analysis of our algorithms for dense polynomial multiplication with integer coefficients.*

Proposition 2 *Multiplying two integers X and Y, with b_X and b_Y bits, amounts at most to $C_{mul}\,(b_X + b_Y)\log(b_X + b_Y)\log(\log(b_X + b_Y))$ bit operations, for some constant C_{mul} (that we shall use in our later estimates).*

Proof. The GMP library relies on different algorithms for multiplying integers [32]. For the largest input sizes, the FFT-based *Schönhage and Strassen algorithms* (SSA) is used. These sizes are often reached in the implementation of our algorithms for dense polynomial multiplication with integer coefficients. The bit complexity for SSA to multiply two integers with a N-bit result is $O(N \log N \log\log N)$ [32, Chapter 16]. The conclusion follows from the fact that the product $X \times Y$ has $N = b_X + b_Y$ bits. □

Proposition 3 *Multiplying two integers X and Y, with b_X and b_Y bits, assuming Y of machine word size, amounts at most to $C'_{mul}\,b_X$ bit operations, for some constant C'_{mul} (that we shall use in our later estimates).*

Proof. Indeed, under the assumption that Y is of machine word size, the GMP library relies on schoolbook multiplication [32]. The conclusion follows. Note that C'_{mul} depends on the machine word size. □

Remark 1 *In Proposition 2, the cost is calculated in bit operations. In other circumstances, counting machine-word operations will be more convenient. For us, the bit size of a machine word is a "small" constant, denoted by MW, which, in practice is either 32 or 64 bits. Hence, we will state our algebraic cost estimates either in bit operations or in machine-word operations, whichever works best in the context of the estimate,*

Notation 2 *For all real number $X \geq 2$, we define*

$$\mathsf{G}(X) = C_{mul}\,X\,\log X \log\log X.$$

Consequently, the estimate stated in Proposition 2 becomes $\mathsf{G}(b_X + b_Y)$.

Remark 2 *The algebraic complexity of adding two integers, each of at most h bits, is at most $C_{add}\,h$, for some constant C_{add} (that we shall use in our later estimates).*

[1] https://gmplib.org/

Proposition 4 *Adding $p \geq 2$ integers, each of at most h bits, amounts at most to $C_{add}\, h\, p$ bit operations.*

PROOF. Up to adding zero items in the input, we can freely assume that p is a power of 2. At first, we are adding $p/2$ pairs of integers having at most h bits which produce $p/2$ integers having at most $h+1$ bits. Then we will have $p/4$ pairs of integers which their results will have $h+2$ bits. By continuing in this manner and using Remark 2 we have:

$$\begin{aligned}
C_{add}\left(\tfrac{p}{2}\,h + \tfrac{p}{2^2}\,(h+1) + \cdots + \tfrac{p}{2^{\log_2 p}}\,(h+\log_2 p - 1)\right) &= C_{add}\left(p\,h\,\textstyle\sum_{i=1}^{\log_2 p}\frac{1}{2^i} + p\,\sum_{i=1}^{\log_2 p}\frac{i-1}{2^i}\right)\\
&= C_{add}\left(p\,h\left(\frac{2^{\log_2 p}-1}{2^{\log_2 p}}\right) + h\left(\frac{2^{\log_2 p}-\log_2 p-2}{2^{\log_2 p}}\right)\right)\\
&= C_{add}\left(p\,h\left(\frac{p-1}{p}\right) + h\left(\frac{p-\log_2 p-2}{p}\right)\right)\\
&= C_{add}\left(h\,(p-1) + h\left(1-\frac{\log_2 p+2}{p}\right)\right)\\
&\leq C_{add}\,(h\,(p-1)+h)\\
&= C_{add}\,p\,h
\end{aligned}$$

□

Assumption 2 *For simplicity, we assume that $m \leq n$ holds in all subsequent results.*

Proposition 5 *The number of bit operations performed by Algorithm 2 is at most:*

$$m\,n\,\mathsf{G}(b_f+b_g) + C_{add}\,m\,n\,(b_f+b_g)$$

PROOF. We have $m \times n$ multiplications of two integers with maximum sizes b_x and b_y, respectively. Using Proposition 2, this gives us the first term of the desired complexity estimate. For the additions, assuming that $m \leq n$ holds, we need to add m integers with the maximum size of $b_x + b_y$ in order to obtain the coefficient of x^i in $f \times g$, for $m-1 \leq i \leq n-1$. For $0 \leq i < m-1$, the coefficient of x^i requires adding $i+1$ integers with b_x+b_y bits. Similarly, for $n \leq i \leq m+n-2$, the degree of x^i requires adding $(n+m-2)-i+1$ integers with b_x+b_y bits. Therefore, the total cost for the additions sums up to:

$$C_{add}\left(m\,((n-1)-(m-1)+1) + 2\sum_{i=0}^{m-2}(i+1)\right)(b_x+b_y) = C_{add}\,m\,n\,(b_x+b_y)$$

□

Assumption 3 *For the purpose of our application to real root isolation, we shall assume that the degrees of the input polynomials are equal, that is, $n = m$. Furthermore, we shall assume that the maximum number of bits for representing any coefficient of an input*

polynomial is of the same order as the degree of that polynomial. This implies that, for all $0 \leq i < n$*, we have* $|f_i| \in \Theta(2^n)$ *and* $|g_i| \in \Theta(2^n)$.

Corollary 1 *Under Assumption 3, the complexity estimate of Algorithm 2 becomes* $2\,C_{mul}\,n^3\,\log(2\,n)\,\log\log(2\,n) + 2\,C_{add}\,n^3$.

3.2 Kronecker substitution method

In this section, we investigate an algorithm for doing polynomial multiplication which takes advantage of integer multiplications. For this purpose, we convert polynomials to relatively large integers, do the integer multiplication, then convert back the result into a polynomial. This method is an application of the so-called *Kronecker substitution*, see [29, Section 8.3] and [18].

As in Section 3.1, we assume that f and g are univariate polynomials over $\mathbb{Z}$ with variable x and with respective positive degrees m and n, see Notation 1. Furthermore, we start by assuming that all of the coefficients of f and g are non-negative. The case of negative coefficients is handled in Section 3.2.1.

With this latter assumption, we observe that every coefficient in the product $f(x)\cdot g(x)$ is strictly bounded over by $H = \min(m,n)H_fH_g + 1$ where H_f and H_g are the maximum value of the coefficients in f and g, respectively. Recall that the binary expansion of a positive integer A has exactly $b(A) = \lfloor \log_2(A) \rfloor + 1$ bits. Thus, every coefficient of f, g, and $f \cdot g$ can be encoded with at most β bits, where $\beta = b(H)$.

Here are the steps of Kronecker substitution for doing integer polynomial multiplication:

1. **Convert-in**: we respectively map the input polynomials $f(x)$ and $g(x)$ to the integers Z_f and Z_g defined as:

$$Z_f \;=\; \sum_{0 \leq i < m} f_i\, 2^{\beta\, i} \quad \text{and} \quad Z_g \;=\; \sum_{0 \leq i < n} g_i\, 2^{\beta\, i}. \tag{3.3}$$

2. **Multiplying**: multiply Z_f and Z_g: $Z_{fg} = Z_f \;.\; Z_g$.

3. **Convert-out**: retrieve coefficients of the product $(f.g)$ from Z_{fg}.

One can regard 2^β as a variable, and thus, Z_f and Z_g are polynomials in $\mathbb{Z}[2^\beta]$. By definition of H, each coefficient of $Z_f \cdot Z_g$ (computed in $\mathbb{Z}[2^\beta]$) is in the range $[0, H)$. Thus, each such coefficient is encoded with at most β bits. Therefore, one can compute

the product Z_{fg} of Z_f and Z_g as an integer number and retrieve the coefficients of the polynomial $f \cdot g$ from the binary expansion of Z_{fg}. This trick allows us to take advantage of asymptotically fast algorithms for multi-precision integers, such as those implemented in the GMP-library.

3.2.1 Handling negative coefficients

We apply a trick which was proposed by R. Fateman [18] in which for each of the negative coefficients (starting from the least significant one), 2^β is borrowed from the next coefficient. For instance, assuming

$$f(x) = f_0 + \cdots + f_i x^i + f_{i+1} x^{i+1} + \cdots + f_d x^d \tag{3.4}$$

where $f_i < 0$ holds, we replace $f(x)$ by

$$f_0 + \cdots + (2^\beta - f_i) x^i + (f_{i+1} - 1) x^{i+1} + \cdots + f_d x^d. \tag{3.5}$$

There's only one special case to his procedure: when the leading coefficient itself is negative. In this situation, the polynomial $f(x)$ is replaced by $-f(x)$ and Z_{fg} by $-Z_{fg}$. Since Fateman's trick may increment each coefficient of f or g by 1, it is necessary to replace β by $\beta' = \beta + 1$ in the above algorithm.

For the **Convert-out** step, if the most significant bit of a $2^{\beta'}$-digit is 1, then it represents a negative coefficient and it should be adjusted by subtracting from $2^{\beta'}$. Otherwise, it is a positive coefficient, and no adjustment is required. Algorithm 3 below sketches integer polynomial via Kronecker substitution.

Algorithm 3: KroneckerSubstitution(f, g, m, n)

Input: $f, g \in \mathbb{Z}[x]$ and $m, n \in \mathbb{N}$ are the sizes respectively.
Output: $h \in \mathbb{Z}[x]$ such that $h = f.g$.
$\beta =$ DetermineResultBits(f, g, m, n);
$Z_f =$ ConvertToInteger(f, m, β);
$Z_g =$ ConvertToInteger(g, n, β);
$Z_{fg} = Z_f \times Z_g$;
$h =$ ConvertFromInteger$(Z_{fg}, \beta, m + n - 1)$;
return h;

Remark 3 *Using the bound H which was discussed earlier, the function call* Determine-

$\mathsf{ResultBits}(f, g, m, n)$ *is computed as follows:*

$$\beta = b_f + b_g + \log_2(\min(m, n)) + 2, \tag{3.6}$$

where $b_f = \lfloor \log_2 H_f \rfloor + 1$*)* *and* $b_g = \lfloor \log_2 H_g \rfloor + 1$ *are the maximum numbers of bits required for representing all of the coefficients of the input polynomials* $f(x)$ *and* $g(x)$*, respectively. In addition, 1 more bit is considered for handling negative coefficients as discussed above.*

The following lemma is an elementary observation and we skip its proof.

Lemma 1 *The cost for converting a univariate integer polynomial into a big integer or converting an integer into a univariate integer polynomial, according to Algorithm 3, is at most* $C_{conv}\, s\, b$ *bit operations, where* s *is the number of terms of the polynomial,* b *is the number of bits required for representing each of the coefficients and* C *is a constant.*

Proposition 6 *The number of bit operations performed by Algorithm 3 is at most:*

$$C_{conv}\ .\ 2(m+n)\beta + \mathsf{G}((m+n)\,\beta)\,. \tag{3.7}$$

where C_{conv} *is the constant introduced in Lemma 1 and* $\beta = b_f + b_g + \log_2 m$ *is as defined in Remark 3.*

Proof. By using Lemma 1, the cost of the **Convert-in** step for the polynomials $f(x)$ and $g(x)$ into Z_f and Z_g are at most $C_{conv}\, m\, \beta$ and $C_{conv}\, n\, \beta$ respectively. In addition, the cost of the **Convert-out** step for $Z_{f.g}$ into $h(x)$ is at most $C_{conv}\,(m+n-1)\,\beta$. The number of bits of the integers Z_f and Z_g are $m\,\beta$ and $n\,\beta$, respectively. Using Proposition 2, the cost for multiplying these two integers is at most $\mathsf{G}\,((m+n)\,\beta)$. Summing all of these helps us to conclude the final estimate. □

Corollary 2 *Using Assumption 3, the algebraic complexity estimate of Algorithm 3 becomes:*

$$C_{mul}\,(4n^2 + 2n\log n)\log(4n^2 + 2n\log n)\log\log(4n^2 + 2n\log n) + C_{conv}\,(8n^2 + 4n\log n)$$

Remark 4 *Corollaries 1 and 2 imply that, under Assumption 3, Algorithm 3 is asymptotically faster than Algorithm 2 by one order of magnitude. Indeed, Corollaries 1 and 2 bring the respective asymptotic upper bounds* $O(n^3\,\log(n)\,\log\log(n))$ *and* $O(n^2\,\log(n)\,\log\log(n))$.

3.2.2 Example

We illustrate an example showing how Kronecker substitution algorithm works.

We have the given polynomials $f(x)$ and $g(x)$ with integer coefficients as follows:

$$f(x) = 41x^3 + 49x^2 + 38x + 29,\ g(x) = 19x^3 + 23x^2 + 46x + 21$$

For simplicity, we consider our computations in the base 10. Thus, the maximum value of the result polynomial would be less than 10^4 (this is equivalent to 2^β defined earlier). In the convert-in step, we evaluate both polynomials at the point 10^4:

$$Z_f = f(10^4) = 41004900380029,\ Z_g = g(10^4) = 19002300460021$$

Then, we multiply the large integers:

$$Z_h = h(10^4) = Z_f \times Z_g = 779187437354540344421320609$$

Finally, we have to retrieve the coefficients of the result polynomial from the large integer. This can be done easily, since we know each 4 digits represent one coefficient of the polynomial:

$$h(x) = 779x^6 + 1874x^5 + 3735x^4 + 4540x^3 + 3444x^2 + 2132x + 609$$

3.3 Classical divide & conquer

We sketch below the classical *Divide & Conquer* approach in which we divide each of the polynomials into 2 polynomials with half of the sizes of the original polynomials and solve the 4 sub-problems, before merging the results together:

1. **Division**: divide each of the input polynomials into 2 polynomials with half sizes.

$$\begin{aligned} f(x) &= \textstyle\sum_{0\le i<m} f_i\, x^i = F_1(x)x^{m/2} + F_0(x) \\ g(x) &= \textstyle\sum_{0\le i<n} g_i\, x^i = G_1(x)x^{n/2} + G_0(x), \end{aligned}$$

   where $F_1(x), F_0(x)$ (resp. $G_1(x), G_0(x)$) are the quotient and remainder in the Euclidean division of $f(x)$ (resp. $g(x)$) by $x^{m/2}$ (resp. $x^{n/2}$).

2. **Sub-Problems**: compute the four polynomial products $F_0(x) \cdot G_0(x)$, $F_1(x) \cdot G_1(x)$, $F_0(x) \cdot G_1(x)$ and $F_1(x) \cdot G_0(x)$ recursively.

3. **Merge**: compute

$$F_1(x)G_1(x)\,x^{(n+m)/2} \;+\; F_1(x)G_0(x)x^{m/2} \;+\; F_0(x)G_1(x)x^{n/2} + F_0(x)G_0(x), \quad (3.8)$$

   which requires to perform coefficient addition.

Remark 5 *With respect to the method based on Kronecker substitution, this divide-and-conquer algorithm provides opportunities for concurrency by means of the four recursive calls which can be executed independently of each other. Those recursive calls themselves may be executed by the same divide-and-conquer procedure. Of course, after a few levels of recursion, it is necessary to use a serial base-case algorithm.*

If for the base-case we choose to reach $n = m = 1$, the whole procedure performs $O(n\,m)$ recursive calls leading to an algorithm which, clearly, is less efficient than the method based on Kronecker substitution in terms of algebraic complexity. Therefore, in practice, we use a small number of recursion levels, denoted by d. In our implementation, this integer d is either 2 or 3. In the base-case, we call Algorithm 3. This design is taken account in Algorithm 4 below.

Algorithm 4: Divide&Conquer(f, g, m, n, d)

Input: $f, g \in \mathbb{Z}[x]$ with the sizes $m, n \in \mathbb{N}$, and $d \in \mathbb{N}$ is the number of recursions.
Output: $h \in \mathbb{Z}[x]$ such that $h = f.g$.

```
BASE = m/2^d;
if n < BASE or m < BASE then
    return KroneckerSubstitution(f,g,m,n);
else
    H0 = Divide&Conquer(F0, G0, m/2, n/2, d − 1);
    H3 = Divide&Conquer(F1, G1, m/2, n/2, d − 1);
    H2 = Divide&Conquer(F0, G1, m/2, n/2, d − 1);
    H1 = Divide&Conquer(F1, G0, m/2, n/2, d − 1);
    return H3(x) x^((n+m)/2) + H2(x) x^(m/2) + H1(x) x^(n/2) + H0(x)
```

As mentioned, in Algorithm 4, the recursive calls can be executed in parallel fashion. For this purpose, auxiliary memory space are used in order to store intermediate results and reduce memory space usage. However, for the sake of simplicity, we do not reflect that in Algorithm 4.

Lemma 2 *Having G defined as in Notation 2, we have:*

$$\mathsf{G}(\frac{X}{2^d}) \leq \frac{\mathsf{G}(X)}{2^d}$$

PROOF. *By virtue of Notation 2, we have:*

$$\begin{aligned}
\mathsf{G}(\tfrac{X}{2^d}) &= C_{mul}\ \tfrac{X}{2^d} \log\left(\tfrac{X}{2^d}\right) \log\log\left(\tfrac{X}{2^d}\right) \\
&= C_{mul}\ \tfrac{X}{2^d} (\log X - d) \log(\log X - d) \\
&\leq C_{mul}\ \tfrac{X}{2^d} \log X \log\log X = \tfrac{\mathsf{G}(X)}{2^d}
\end{aligned}$$

□

Proposition 7 *The number of bit operations necessary to run Algorithm 4 is at most:*

$$2^d \left(C_{conv}\, 2\, (m+n)\, (\beta - d) + \mathsf{G}((m+n)(\beta - d))\right) + C_{add}\,(m+n)\,(\beta - d)$$

where $d = \log_2(m/\mathsf{BASE})$ is the number of recursion levels and C_{conv}, C_{add}, β are as in Section 3.2.

PROOF. Elementary calculations lead to the following relation:

$$T(n,m) \;=\; 4^d T'(n/2^d, m/2^d) + C_{add}\,(m+n)\,(\beta - d)$$

where $T'(n',m')$ is the cost for multiplying two polynomials with degrees $n' = n/2^d$ and $m' = m/2^d$. Note that the number of bits of the result of the Kronecker substitution algorithm for these sizes will be $b_f + b_g + \log_2(m/2^d) = b_f + b_g + \log_2 m - d$ which is equal to $\beta - d$. Using the result of Proposition 6 to evaluate $T'(n',m')$, we have:

$$T(n,m) = 4^d \left(C_{conv}\, 2 \left(\frac{m+n}{2^d}\right)(\beta - d) + \mathsf{G}\left(\left(\frac{m+n}{2^d}\right)(\beta - d)\right)\right) + C_{add}\,(m+n)\,(\beta - d)$$

The second term is due to copying the intermediate results from the auxiliary arrays to the final result: using Proposition 4 and the fact that we have $2\,(m/2 + n/2 - 1)$ additions of integers with size of $\beta - d$, explains this second term. Moreover, using Lemma 2 we obtain:

$$\mathsf{G}\left(\left(\frac{m+n}{2^d}\right)(\beta - d)\right) < \frac{\mathsf{G}\left((m+n)\,(\beta - d)\right)}{2^d}$$

The desired result follows after elementary simplifications. □

Corollary 3 *Using Assumption 3, Proposition 7 and Remark 3, we have the following*

algebraic complexity estimate for Algorithm 4:

$$C_{conv}\, 2^{d+2}\, \left(2\, n^2 + n\, \log_2 n - d\, n\right) + C_{add}\, 2\, \left(2\, n^2 + n\, \log_2 n - d\, n\right) +$$
$$C_{mul}\, 2^{d}\, \left(2\, n^2 + n\, \log_2 n - d\, n\right) \log\left(2\, n^2 + n\, \log_2 n - d\, n\right) \log\log\left(2\, n^2 + n\, \log_2 n - d\, n\right)$$

Remark 6 *From Corollary 3, we observe that the arithmetic count of the divivide & conquer algorithm is roughly 2^d times that of algorithm based on Kronecker-Substitution (see Corollary 2).*

3.4 Toom-Cook algorithm

The famous Toom-Cook algorithm, which is credited to Toom [62] and Cook [13], is based on the same approach as Karatsuba's trick [41], in which polynomial multiplication is divided into small sub-problems by means of polynomial evaluation, which are recombined via interpolation and linear combination of terms. In this section, we propose an approach combining Toom-Cook algorithm and Kronecker substitution, as stated in Algorithm 6.

Assumption 4 *The basic Toom-Cook algorithm works when the degree of the input polynomials are equal. Here, we assume that $n = m$ holds too (see Notation 1). The case where the input polynomials are not balanced is an on-going research topic, see [70] for instance.*

Remark 7 *As a result of Assumption 4, when our implementation of Toom-Cook algorithm is called on two polynomials with different degrees, "zero leading coefficients" are added to the smaller polynomial of degree in order to "equalize degrees".*

Our method is explained below:

1. **Convert-in**: convert each of the polynomials into so-called *Toom-polynomials* with degree k. For this, first we divide each of the polynomials into k sub-polynomials. Note that we can assume that m is multiple of k (otherwise, we add "zero leading coefficients" so as the number of coefficients becomes a multiple of k):

$$f(x) = \sum_{i=0}^{k-1} F_i(x)(x^{m/k})^i \ , \ g(x) = \sum_{i=0}^{k-1} G_i(x)(x^{m/k})^i, \tag{3.9}$$

   where, for $0 \leq i < k$, the polynomials $F_i, G_i \in \mathbb{Z}[x]$ have degree at most $m/k - 1$.

Then, we apply Kronecker Substitution to produce $Z_{F_i}, Z_{G_i} \in \mathbb{Z}$ corresponding to $F_i(x),\ G_i(x)$, for $0 \le i < k$, and we define:

$$F(x') = \sum_{i=0}^{k-1} Z_{F_i} x'^i\ ,\ G(x') = \sum_{i=0}^{k-1} Z_{G_i} x'^i. \tag{3.10}$$

In the above equation, $x^{m/k}$ is replaced by x'.

2. **Multiplying using k-way Toom-Cook**: compute $H(x') = F(x') \times G(x') = \sum_{i=0}^{2k-2} Z_{H_i} x'^i$ using k-way Toom-Cook. In this step, first, we evaluate the polynomials $F(x')$ and $G(x')$ at $2k-1$ distinct points $v_0, \ldots, v_{2k-2}$ where $v_i \in \mathbb{Q}$, for $0 \le i < 2k-1$. Thus, we define:

$$A_i \coloneqq F(v_i) \quad \text{and} \quad B_i = G(v_i). \tag{3.11}$$

Then we do point-wise multiplications.

$$(C_0, \ldots, C_{2k-2}) \ = \ (H(v_0), \ldots, H(v_{2k-2})) = (A_0 \times B_0, \ldots, A_{2k-2} \times B_{2k-2}) \tag{3.12}$$

At last, we recover the coefficients $Z_{H_0}, \ldots, Z_{H_{2k-2}}$ of $H(x')$ from the $2k-1$ pairs (C_i, v_i), for $0 \le i < 2k-1$. In practice, 2 trivial points are chosen, namely $v_0 = 0$ and $v_{2k-2} = \infty$. This helps to minimize the number of arithmetic operations for evaluation and interpolation parts of the algorithm.

3. **Convert-out**: We recover the polynomial product $h(x) \coloneqq f(x)\, g(x)$ from the integers $Z_{H_0}, \ldots, Z_{H_{2k-2}}$. This requires to convert these integers to polynomials $H_0(x), \ldots, H_{2k-2}(x)$, each of degree at most $2m/k - 2$, such that we have:

$$h(x) = \sum_{i=0}^{2k-2} H_i(x)(x^{m/k})^i \to h(x) = \sum_{i=0}^{2m-2} h_i x^i. \tag{3.13}$$

As it can be seen in the above formula that, some of the expressions $H_i(x)(x^{m/k})^i$ may have terms of the same degree, implying some necessary additions. In our implementation we have used auxiliary arrays for handling those overlapping expressions: this helps executing all conversions and additions concurrently. This implies that all of the H_is will be computed in a parallel fashion; and at last, we have to iterate over the auxiliary arrays to add their elements to the final result array, see Algorithm 5.

Toom-Cook Matrices. One can view the evaluation and interpolation phases of Toom-Cook algorithm as the computation of three matrix products [5, 4]:

- **Evaluation**: we define the vector containing the values of the polynomials F and G, as A and B respectively. The Toom matrix for evaluating the polynomials is called M_k which is depicted below. Then this evaluation process is obtained as the matrix-vector product $M_k \times F$.

$$
\begin{aligned}
A = \begin{bmatrix} F(0) \\ F(v_1) \\ . \\ . \\ . \\ F(v_{2k-3}) \\ F(\infty) \end{bmatrix} &= \begin{bmatrix} 1 & 0 & \dots & 0 \\ v_1^0 & v_1^1 & \dots & v_1^{k-1} \\ . & . & \dots & . \\ . & . & \dots & . \\ . & . & \dots & . \\ v_{2k-3}^0 & v_{2k-3}^1 & \dots & v_{2k-3}^{k-1} \\ 0 & 0 & \dots & 1 \end{bmatrix} \times \begin{bmatrix} Z_{F_0} \\ Z_{F_1} \\ . \\ . \\ . \\ Z_{F_{k-1}} \end{bmatrix} = M_k \times F \\
B = \begin{bmatrix} G(0) \\ G(v_1) \\ . \\ . \\ . \\ G(v_{2k-3}) \\ G(\infty) \end{bmatrix} &= \begin{bmatrix} 1 & 0 & \dots & 0 \\ v_1^0 & v_1^1 & \dots & v_1^{k-1} \\ . & . & \dots & . \\ . & . & \dots & . \\ . & . & \dots & . \\ v_{2k-3}^0 & v_{2k-3}^1 & \dots & v_{2k-3}^{k-1} \\ 0 & 0 & \dots & 1 \end{bmatrix} \times \begin{bmatrix} Z_{G_0} \\ Z_{G_1} \\ . \\ . \\ . \\ Z_{G_{k-1}} \end{bmatrix} = M_k \times G
\end{aligned}
\tag{3.14}
$$

- **Interpolation**: we define the vector containing the coefficients of the polynomial H as C. The Toom matrix M_{2k-1} for interpolating H is depicted below. Then this interpolation process is obtained as the matrix-vector product $M_{2k-1}^{-1} \times C$.

$$
H = \begin{bmatrix} Z_{H_0} \\ Z_{H_1} \\ . \\ . \\ Z_{H_{2k-2}} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ v_1^0 & v_1^1 & \dots & v_1^{2k-2} \\ . & . & \dots & . \\ . & . & \dots & . \\ . & . & \dots & . \\ v_{2k-3}^0 & v_{2k-3}^1 & \dots & v_{2k-3}^{2k-2} \\ 0 & 0 & \dots & 1 \end{bmatrix}^{-1} \times \begin{bmatrix} C_0 \\ C_1 \\ . \\ . \\ . \\ C_{2k-2} \end{bmatrix} = M_{2k-1}^{-1} \times C \tag{3.15}
$$

Algorithm 5: Recover$(H, \beta, size)$

Input: $H \in \mathbb{Z}[x]$ with the size of $2k-1$.
$\beta \in \mathbb{N}$ is the number of bits per coefficient.
$size \in \mathbb{N}$ is the size of the result polynomial.
Output: $h \in \mathbb{Z}[x]$.
for $i = 0 \ldots k-1$ **do**
    H_{2i} = CovertFromInteger$(Z_{H_{2i}}, \beta, size/k)$;
    $tmp_{H_{2i+1}}$ = CovertFromInteger$(Z_{H_{2i+1}}, \beta, size/k)$;
for $i = 0 \ldots 2k-1$ **do**
    add H_i and tmp_{H_i} to h;
return h;

Algorithm 6: ToomCookK(f, g, m)

Input: $f, g \in \mathbb{Z}[x]$ with the size of $m \in \mathbb{N}$ and the evaluation $points$.
Output: $h \in \mathbb{Z}[x]$ such that $h = f.g$.
β = DetermineResultBits(f, g, m, m);
Chop f, g into k sub-polynomials;
for $i = 0 \ldots k-1$ **do**
    $Z_{F_i} = F_i\ (\beta)$;
    $Z_{G_i} = G_i\ (\beta)$;
A = Evaluate$(F, points)$;
B = Evaluate$(G, points)$;
for $i = 0 \ldots 2k-1$ **do**
    $C[i] = A[i] \times B[i]$;
H = Interpolate$(C, points)$;
h = Recover$(H, \beta, 2m-1)$;
return h;

Complexity analysis An algebraic complexity estimate of multiplying two polynomials with degree n by applying k-way Toom-Cook recursively is $O(n^{\log_k(2k-1)})$ coefficient operations, see [42, p. 280]. However this estimate considers only the costs for the point-wise multiplications which are known to dominate the whole algorithm. Yet, the algebraic complexity of the evaluation and interpolation steps of the algorithm grow as k increases. Both our experimentation and theoretical analysis confirm that, by choosing relatively large k, the evaluation and interpolation steps start to dominate the whole algorithm.

In our proposed method (Algorithm 6), we are not applying the Toom-Cook trick recursively. Moreover, we present a more precise analysis for Algorithm 6 by computing its number of arithmetic operations. We analyze the *span* (in the sense of the fork-join model, see Section 2.1.1) of the algorithm when applying the parallel implementation in Section 3.5.

Assumption The size of each element of the Toom-matrix (or its inverse) are relatively small, namely in the order of a machine-word size MW. In our complexity estimates, we assume that they all have the same size MW.

Lemma 3 *The number of bit operations for converting the input polynomials into the Toom-polynomials is at most* $2\,C_{conv}\,\beta\,m$*, where* C_{conv} *is the constant introduced in Lemma 1 and* β *is as defined in Remark 3.*

Proof. The number of bits for coefficients of the Toom-polynomial for polynomial $F(x')$ and $G(x')$ is $m/k\,\beta$. Thus, by using Lemma 1, the number of operations for converting both polynomials would be $(2m/k\,\beta)\;k$. □

Lemma 4 *The number of bit operations for evaluating the polynomials* $F(x')$ *and* $G(x')$ *at* $2k-1$ *points is at most (where* C_{add}*,* C'_{mul} *are constants, and* β *as defined in Remark 3):*

$$2\;(2\,k-3)\;(C'_{mul}+C_{add})\;m\,\beta. \tag{3.16}$$

Proof. We estimate the algebraic complexity for one polynomial, say $F(x')$: each of the point evaluation is equivalent of a vector multiplication: $(v_i^0,\ldots,v_i^{k-1})\times(Z_{F_0},\ldots,Z_{F_{k-1}})$. As a result, we have k multiplications of integers with the size of $m/k\,\beta$ and relatively small integers (in the order of machine-word integers MW), plus $k-1$ additions of those intermediate result. Using Propositions 3, the cost for all of the multiplications is: $C'_{mul}\,k\;(m/k\,\beta) = C'_{mul}\,m\,\beta$. In addition, by using Proposition 4, the cost for additions is $C_{add}\,k\;(m/k\beta) = C_{add}\,m\,\beta$.

Considering that we have $2k-1$ points of evaluating in which 2 points are trivial, the cost for evaluating F will be: $(2\,k-3)\,\left(C'_{mul}\,m\,\beta + C_{add}\,m\,\beta\right)$.

Considering the cost of evaluating $G(x')$, and putting them all together, helps us to estimate the overall cost as it is stated above. □

Remark 8 *The number of bits of the evaluated values for F and G are $m/k\beta + \mathsf{MW} + \log_2(k-1)$. We can consider them as $m/k\beta + \log_2 k$.*

Remark 9 *The number of bits of the evaluated values of the result Toom-polynomial (Z_{H_i} for $0 \le i < 2k-1$) is $2\,m/k\beta + 2\mathsf{MW} + 2\log_2(k-1)$. For simplicity, we ignore the constant term, leading to the new estimate as $2\,m/k\beta + 2\log_2 k$.*

Lemma 5 *The number of bit operations in the point-wise multiplications in the algorithm is at most:*

$$(2k-1)\,\mathsf{G}\left(\frac{2\,m}{k}\beta + 2\log_2 k\right)$$

where $\beta = b_f + b_g + \log_2 m$ from Remark 3.

Proof. Considering Remark 9, having $2k-1$ point-wise multiplications, and by using Proposition 2, the overall cost can be computed as stated above □

Lemma 6 *The number of bit operations in the interpolation of the polynomial by having $2k-1$ points and their evaluated values is at most:*

$$4\,(2\,k-3)\,(C'_{mul} + C_{add})\,(m\,\beta + k\,\log_2 k)\,, \tag{3.17}$$

where C_{add} and C'_{mul} are constants and $\beta = b_f + b_g + \log_2 m$ from Remark 3.

Proof. We calculate the number of operations as we did for the evaluation. For each of the rows, we have $2k-1$ multiplications of an integer with $2\,m/k\,\beta + 2\log_2 k$ (see Remark 9) bits and a relatively small and constant number of bits and $2k-2$ additions. Using Proposition 4, the cost for additions is:

$$C_{add}\,(2k-1)\,\left(\frac{2\,m}{k}\,\beta + 2\,\log_2 k\right) < C_{add}\,\left(4\,m\,\beta + 4\,k\,\log_2 k\right)$$

Using Proposition 3, the cost for multiplications is:

$$C'_{mul}\,(2k-1)\,\left(\frac{2\,m}{k}\,\beta + 2\,\log_2 k\right) < C'_{mul}\,\left(4\,m\,\beta + 4\,k\,\log_2 k\right)$$

By simplifying formulas we can conclude. □

Lemma 7 *The number of bit operations for converting out and recovering the result is at most:*

$$2\,C_{conv}\,(2\,m-k)\,\beta$$

where C_{conv} is the constant introduced in Lemma 1 and β is as defined in Remark 3.

PROOF. The converting out is a linear algorithm where we want to extract the coefficients of the result polynomial which have β (see Remark 3) bits, with the size of $2m/k-1$. Using Lemma 1 and having $2k-1$ intermediate results which we have to recover from we will have:

$$C_{conv}\,(2\,k-1)\,(2\,m/k-1)\,\beta < C_{conv}\,2\,(2\,m-k)\,\beta$$

□

Proposition 8 *The number of bit operations of Algorithm 6 is at most:*

$$C_{conv}\,\beta\,(6\,m-2\,k)+(2k-1)\,\mathsf{G}\left(\frac{2\,m}{k}\,\beta+2\,\log_2 k\right)+(2k-3)(C'_{mul}+C_{add})\,(6\,m\,\beta+4\,\log_2 k)\,,$$

where C_{conv}, C'_{mul}, C_{add} are constants and β from Remark 3.

PROOF. We sum up the costs in Lemmas 3, 4, 5, 6, 7 to compute the cost. The first term is for converting the representations, the second term is for doing point-wise multiplications, and the third one is due to evaluations/interpolation. □

Corollary 4 *Using Assumption 3, Proposition 8, and replacing β by using Remark 3, the complexity of the Algorithm 6 will become:*

$$C_{conv}\left(12\,n^2+6\,n\,\log_2 n-4\,k\,n-2\,k\,\log_2 n\right)+(2k-1)\,\mathsf{G}\left(\frac{4\,n^2+2\,n\,\log_2 n}{k}+2\,\log_2 k\right)+$$
$$(2k-3)(C'_{mul}+C_{add})\left(12\,n^2-6\,n\,\log_2 n+4\,\log_2 k\right)$$

Remark 10 *From Corollary 4, we observe that when k is fixed, the term $n^2\,\log n\log\log n$ dominate the arithmetic count. However when k grows, the terms coming from the* **Convert-in** *and* **Convert-out** *(which are easily identified by the constants C_{conv}, C'_{mul}, C_{add}) will grow. As a result, one should not expect much progress in implementing this algorithm for larger k than 8, as we, and others, did not.*

3.5 Parallelization

In this section, we describe the parallelization of the algorithms presented in Sections 3.2 through 3.4. For the purpose of our parallel computations on multi-core processors, our code is written in `Cilkplus` [2] [43] and is part of the *Basic Polynomial Algebra Subprograms* (BPAS)[3]. As mentioned before, our implementation relies on the GMP-library [4] for computing with multi-precision integer numbers.

3.5.1 Classical divide & conquer

In our classical divide and conquer implementation, we have used 2 levels of recursion ($d = 2$), and then calling Kronecker substitution for the base cases. This means that we divide the input problem into 16 subproblems ($4^d = 16$) with the sizes of $n/4$ and $m/4$. These 16 subproblems are run concurrently.

Corollary 5 *Using Proposition 7 and the fact that $d = 2$, the work of the Algorithm 4 becomes at-most:*

$$4\,(2\,C_{conv}\,(m+n)\,\beta + \mathsf{G}((m+n)\,\beta)) + C_{add}\,(m+n)\,\beta,$$

where C_{conv} is the constant introduced in Lemma 1 and β is as defined in Remark 3.

Proposition 9 *The span of Algorithm 4 is at most:*

$$\frac{2\,C_{conv}\,(m+n)\,\beta + \mathsf{G}((m+n)\,\beta)}{4} + C_{add}\,(m+n)\,\beta,$$

where C_{conv} is the constant introduced in Lemma 1 and β is as defined in Remark 3.

Proof. Since we will be executing each subproblem in parallel, we have:

$$T(n,m) = T'(n/2^d, m/2^d) + C_{add}\,(m+n)\,\beta,$$

in where $T'(n',m')$ is the cost for multiplying two polynomials with the degree of $n' = n/4$ and $m' = m/4$ using the Kronecker-Substitution algorithm. By replacing T' from the result of Proposition 6 we can conclude the result. Note that the additions are not parallelized, since they are considered to be cheap operations. □

[2] https://www.cilkplus.org/
[3] `http://www.bpaslib.org`
[4] https://gmplib.org/

3.5.2 4-way Toom-Cook

Points. The points considered for the 4-way Toom-Cook in our implementations are

$$(0, 1, -1, 1/2, -1/2, 2, \infty).$$

These points will cause the Toom-Cook matrices to be *appropriate* in the sense of algebraic complexity, see [5].

Evaluation. The Toom-Cook matrix for $k = 4$ looks like:

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 8 & 4 & 2 & 1 \\ 8 & -4 & 2 & -1 \\ 1 & 2 & 4 & 8 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} Z_{F_0} \\ Z_{F_1} \\ Z_{F_2} \\ Z_{F_3} \end{bmatrix} = M_4 \times F$$

Note that we have multiplied Rows 4 and 5 by 8 for simplicity. This means that we are computing $8 \times F(1/2)$ and $8 \times F(-1/2)$, respectively. In the interpolation phase, this will be taken into account by multiplying Rows 4 and 5 by 64 in M_7, which means that we recover from $64 \times H(1/2)$ and $64 \times H(-1/2)$, see Section 3.5.3.

Remark 11 *When evaluating a polynomial at $1/2^i$ (or $-1/2^i$) in the k-way Toom-Cook algorithm, we multiply the corresponding row in M_k by 2^{k-1} which is equivalent to computing $2^{k-1}F(1/2^i)$ (or $2^{k-1}F(-1/2^i)$). For recovering the result polynomial in the interpolation, we must know that we have computed $2^{2(k-1)}H(1/2^i)$ (or $2^{2(k-1)}H(-1/2^i)$); therefore, we multiply the corresponding row of the matrix M_{2k-1} in the interpolation with $2^{2(k-1)}$.*

The pseudo-code for evaluating a polynomial in 4-way Toom-Cook are in Algorithm 7.

Consider that for each row of M_4, we have a complementary row which has same elements and negative elements with respect to the original row. Having this helps us to reuse some intermediate results which has been taken into account in Algorithm 7. Moreover, evaluating each of the polynomials are independent tasks and can be executed with different cores as it is written in Algorithm 6.

Algorithm 7: Evaluate4(F)

Input: $F \in \mathbb{Z}[x']$: $F(x') = Z_{F_0} + Z_{F_1}x' + Z_{F_2}x'^2 + Z_{F_3}x'^3$.
Output: A is the vector of evaluated points.
$tmp_0 = Z_{F_0} + Z_{F_2}$;
$tmp_1 = Z_{F_1} + Z_{F_3}$;
$tmp_2 = 8 \,.\, Z_{F_0} + 2 \,.\, Z_{F_2}$;
$tmp_3 = 4 \,.\, Z_{F_1} + Z_{F_3}$;
$A_0 = Z_{F_0}$;
$A_1 = tmp_0 + tmp_1$;
$A_2 = tmp_0 - tmp_1$;
$A_3 = tmp_2 + tmp_3$;
$A_4 = tmp_2 - tmp_3$;
$A_5 = Z_{F_0} + 2 \,.\, Z_{F_1} + 4 \,.\, Z_{F_2} + 8 \,.\, Z_{F_3}$;
$A_6 = Z_{F_3}$;
return A;

Interpolation. Given the 7 evaluated points, we have to find a polynomial with a degree of 7 which goes from those points. Using Remark 11, the interpolation step becomes:

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 \\ 64 & 32 & 16 & 8 & 4 & 2 & 1 \\ 64 & -32 & 16 & -8 & 4 & -2 & 1 \\ 1 & 2 & 4 & 8 & 16 & 32 & 64 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \times \begin{bmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{bmatrix} = M_7^{-1} \times C$$

M. Bodrato and A. Zanoni investigated the optimal so-called *Inversion-Sequence* for converting the Toom-Cook M_7 into the identity matrix by proposing an optimality criteria [5], based on the algebraic complexity of this conversion. We have implemented their approach and the optimal *Inversion-Sequence* which converts M_7 to the identity matrix I_7 is shown in Algorithm 8.

Other parallelism We have implemented Algorithm 6 for $k = 4$ and adapting it to fork-join parallel model in `CilkPlus`. For this, all of the conversions of sub-polynomials into large integers (coefficients of the Toom polynomial) are executed in parallel, for both input polynomials. As a result, for this step, we will have 8 concurrent processes. Furthermore, the point-wise multiplications (7 multiplications in 4-way Toom-cook) is

Algorithm 8: Interpolate4(c)

Input: C is a vector of evaluated points.
Output: The coefficients of the polynomial H (In-Place, $Z_{H_i} = C_i$).
$C_5 += C_3$; $C_4 += C_3$; $C_4 /= 2$; $C_2 += C_1$; $C_2 /= 2$; $C_3 -= C_4$;
$C_1 -= C_2$; $C_5 -= 65 * C_2$; $C_2 -= C_6$; $C_2 -= C_0$;
$C_5 += 45 * C_2$; $C_4 -= C_6$; $C_4 /= 4$; $C_3 /= 2$; $C_5 -= 4 * C_3$;
$C_3 -= C_1$; $C_3 /= 3$; $C_4 -= 16 * C_0$; $C_4 -= 4 * C_2$; $C_4 /= -3$; $C_2 -= C_4$;
$C_5 /= 30$; $C_1 -= C_5$; $C_1 -= C_3$; $C_1 /= -3$; $C_3 -= 5 * C_1$; $C_5 += C_1$;

executed in parallel fashion. At last, the recovering step of the algorithm is parallelized based on Algorithm 5 by executing each of the conversions in parallel (the first loop in Algorithm 5).

Corollary 6 *Using Proposition 8 and the fact that $k = 4$, the work of the 4-way Toom-Cook becomes at most:*

$$6\,C_{conv}\,\beta\,m + 7\,\mathsf{G}\left(\frac{m}{2}\,\beta\right) + 30\,(C'_{mul} + C_{add})m\,\beta$$

Proposition 10 *The span of 4-way Toom-Cook is at most:*

$$\frac{3}{4}\,C_{conv}\,\beta\,m + \mathsf{G}\left(\frac{m}{2}\,\beta\right) + \frac{47}{4}\,C_{add}\,m\,\beta + 5\,C'_{mul}\,m\,\beta$$

Proof. The Convert-in will be executed in parallel for both polynomials (so is Convert-out), and we will have the parallelism of 8. The 7 point-wise multiplications will be executed concurrently too with the parallelism of 7. (The first 2 terms)

For the evaluation, in the Algorithm 7, we have 11 additions, and 6 multiplication of integers with size roughly $m/4$ β using Remark 9. Plus, evaluating for each of the polynomials will be executed in parallel. Then, the span for this part of the algorithm will be:

$$11\,C_{add}\,\frac{m}{4}\,\beta + 6\,C'_{mul}\left(\frac{m}{4}\,\beta\right)$$

For the interpolation, in Algorithm 8, it can be seen that we have 18 additions, and 7 shifts and multiplications:

$$18\,C_{add}\left(\frac{m}{2}\,\beta\right) + 7\,C'_{mul}\left(\frac{m}{2}\,\beta\right)$$

Summing up these intermediate results helps us to conclude. □

Corollary 7 *Comparing Lemmas 4 and 6 for $k = 4$, and the results in the proof of Proposition 10, we can see that the parallelism for evaluation and interpolation steps of*

the algorithm is roughly 4-5. Moreover, the parallelism for the point-wise multiplications and converting steps are 7 and 8 respectively.

3.5.3 8-way Toom-Cook

Points. The points considered for the 8-way Toom-Cook in our implementations are:

$$(0, 1, -1, 1/2, -1/2, 2, -2, 1/4, -1/4, 4, -4, 1/8, -1/8, 8, \infty).$$

These points will cause the Toom-Cook matrices to be appropriate in the sense of arithmetic calculations and algebraic complexity. Recall Remark 11 which will be used in generating the Toom matrix for $k = 8$.

Evaluation. The Toom matrix for evaluation is well-structured as it is shown below. Hence, we can use some tricks to have an efficient parallel implementation of it. Here are some ideas that were taken into account in our implementation:

- First & last evaluation points chosen to be 0 and ∞ respectively.

- Since the coefficients of the polynomial are extremely large, we somehow need to manage efficient reading in order to avoid memory contention. For this, we consider one core to be responsible for multiplying even indexes of the coefficients with the corresponding elements of the Toom matrix. In the picture, all of the blue elements will be handled by 1 core (so will the red ones). We call the intermediate sum/multiply result of blue elements of row i as X_i (and Y_i for red ones).

- For each one of the other rows, we have a complimentary row with some negative elements to each row, so we can compute the common factors only once. Considering this, we have reused each of X_is and Y_is for the two final results. Each of these sums will be handled by a single core. This, again, makes the memory accesses more efficient.

- Finally, like 4-Way Toom-Cook, evaluating each of the polynomials are independent tasks and can be executed with different cores.

$$A = \begin{pmatrix} f_0 \\ X_1 + Y_1 \\ X_1 - Y_1 \\ X_2 + Y_2 \\ X_2 - Y_2 \\ X_3 + Y_3 \\ X_3 - Y_3 \\ X_4 + Y_4 \\ X_4 - Y_4 \\ X_5 + Y_5 \\ X_5 - Y_5 \\ X_6 + Y_6 \\ X_6 - Y_6 \\ X_7 + Y_7 \\ f_7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2 & 1 \\ 2^7 & -2^6 & 2^5 & -2^4 & 2^3 & -2^2 & 2 & -1 \\ 1 & 2 & 2^2 & 2^3 & 2^4 & 2^5 & 2^6 & 2^7 \\ 1 & -2 & 2^2 & -2^3 & 2^4 & -2^5 & 2^6 & -2^7 \\ 2^{14} & 2^{12} & 2^{10} & 2^8 & 2^6 & 2^4 & 2^2 & 1 \\ 2^{14} & -2^{12} & 2^{10} & -2^8 & 2^6 & -2^4 & 2^2 & -1 \\ 1 & 2^2 & 2^4 & 2^6 & 2^8 & 2^{10} & 2^{12} & 2^{14} \\ 1 & -2^2 & 2^4 & -2^6 & 2^8 & -2^{10} & 2^{12} & -2^{14} \\ 2^{21} & 2^{18} & 2^{15} & 2^{12} & 2^9 & 2^6 & 2^3 & 1 \\ 2^{21} & -2^{18} & 2^{15} & -2^{12} & 2^9 & -2^6 & 2^3 & -1 \\ 1 & 2^3 & 2^6 & 2^9 & 2^{12} & 2^{15} & 2^{18} & 2^{21} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} Z_{F_0} \\ Z_{F_1} \\ Z_{F_2} \\ Z_{F_3} \\ Z_{F_4} \\ Z_{F_5} \\ Z_{F_6} \\ Z_{F_7} \end{pmatrix} = M_8 \times F$$

Interpolation. The inverse matrix (M_{15}^{-1}) is not well-structured; furthermore, by our experimental observation, the generated Inversion-Sequence could not help us to beat our parallel code. Here are some tricks that were taken to account in our parallel implementation:

- The first and last row of the inverse matrix would be the same as M_{15}.
- We parallelize multiplying the matrix by dividing it into *4-columns-M_{15}^{-1}/4-elements* C. This is due to avoid concurrent memory access to the same data, since elements of C are extremely large (because they are large integers corresponding to some polynomials with large coefficients).
- After computing intermediate results, we can merge/add them in a trivial concurrent fashion.

The work devision is shown in the matrix below:

$$H = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 1 & 1 & 1 & \dots & 1 \\ 1 & -1 & 1 & \dots & 1 \\ 2^{14} & 2^{13} & 2^{12} & \dots & 1 \\ \cdot & \cdot & \cdots & \cdot & \\ 1 & 8 & 8^2 & \dots & 8^{14} \\ 0 & 0 & 0 & \dots & 1 \end{pmatrix} \times C = \begin{pmatrix} \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \end{pmatrix} \times \begin{pmatrix} C_0 \\ \cdot \\ C_3 \\ C_4 \\ \cdot \\ C_7 \\ C_8 \\ \cdot \\ C_{11} \\ C_{12} \\ \cdot \\ C_{14} \end{pmatrix} = M_{15}^{-1} \times C$$

Additional comments on the parallelization of 8-way Toom-Cook We have implemented Algorithm 6 for $k = 8$ and adapting it to fork-join parallel model. For this purpose as it was discussed for the 4-way Toom-Cook, all of the converting sub-polynomials into large integers and converting back (based on the Algorithm 5) will be executed in parallel. Furthermore, the point-wise multiplications (15 multiplications in 8-way Toom-cook) will be executed in parallel fashion.

Corollary 8 *Using Proposition 8 and the fact that $k = 8$, the work of the 8-way Toom-Cook becomes at most:*

$$6\,C_{conv}\,\beta\,m + 15\,\mathsf{G}\left(\frac{m}{4}\,\beta\right) + 78\,(C'_{mul} + C_{add})m\,\beta$$

Proposition 11 *The span of 8-way Toom-Cook is at most:*

$$\frac{3}{8}\,C_{conv}\,\beta\,m + \mathsf{G}\left(\frac{m}{4}\,\beta\right) + \frac{213}{16}\,C_{add}\,m\,\beta + \frac{79}{4}\,C'_{mul}\,m\,\beta$$

Proof. The Convert-in will be executed in parallel for both polynomials (so is Convert-out), and we will have the parallelism of 16. The 15 point-wise multiplications will be executed concurrently too with the parallelism of 15. (The first 2 terms)

For the evaluation, as it were discussed on how we parallelize them, we have 21 multiplications and 28 additions for each of the blue and red elements which will be executed in parallel. Plus 14 another additions for using these intermediate results will be executed using 7 processes simultaneously. The span for this part of the algorithm

will be:

$$21\,C'_{mul}\left(\frac{m}{8}\beta\right) + 28\,C_{add}\left(\frac{m}{8}\beta\right) + \frac{1}{2}\,C_{add}\left(\frac{m}{8}\beta\right)$$

For the interpolation, for each of 4 columns, there will be 28 multiplications, and 39 additions. Plus, for computing each of the final results, 4 additions would be needed. Then the span for the interpolation will be:

$$29\,C'_{mul}\left(\frac{m}{4}\beta\right) + 39\,C_{add}\left(\frac{m}{4}\beta\right)$$

Summing up these intermediate results helps us to conclude. □

Corollary 9 *Comparing Lemmas 4 and 6 for $k = 8$, and the results in the proof of Proposition 11, we can see that the parallelism for evaluation and interpolation steps of the algorithm is roughly 8. Plus, the parallelism for the point-wise multiplications and converting steps are 15 and 16 respectively.*

3.6 Experimentation

Execution times for the different algorithms, that were discussed in this chapter, can be found in Table 3.3. In these benchmarks, the number N of bits of the generated coefficients for both input polynomials are equal to the size s (i.e. the degree plus one) of the polynomial (see Assumption 3). We have executed our benchmarks on two different machines:

1. Intel X5650 having 12 Cores (24 Cores with Hyper-Threading) with frequency of 2.67GHz.

2. AMD Opteron Processor 6168 having 48 cores with frequency of 1.9GHz.

In the Table 3.5 the data from `Cilkview`, the performance analysis tool of `CilkPlus` are gathered. There are some interesting points that can be interpreted from these data:

- **8-way Toom-Cook** has the best span and work as it is computed in Proposition 11 and Corollary 8. In the benchmarks on the Intel node, this algorithm is slower because this node has only 12 physical cores, which is not sufficient to expose all the parallelism of this algorithm. Indeed, this algorithm can split into 16 parallel processes. We can see that **8-way Toom-Cook** outperforms the other algorithms on the AMD nodes, which has 48 physical cores.

N, s	KS	DnC	Toom-4	Toom-8
128	0.001	0.015	0.006	0.01
256	0.004	0.023	0.008	0.019
512	0.008	0.027	0.015	0.016
1024	0.033	0.066	0.033	0.038
2048	0.147	0.114	0.078	0.1
4096	0.834	0.491	0.278	0.366
8192	3.138	2.129	1.298	1.497
16384	11.895	8.15	4.529	6.175
32768	52.388	30.202	17.487	23.498
65536	*Err.	140.657	75.06	89.6

Table 3.1: Intel node

KS	DnC	Toom-4	Toom-8
0.002	0.064	0.007	0.012
0.009	0.061	0.01	0.017
0.022	0.049	0.021	0.025
0.06	0.136	0.042	0.062
0.252	0.261	0.132	0.179
1.378	0.82	0.623	0.658
5.421	3.412	2.85	2.491
20.773	9.047	10.569	8.974
91.843	31.081	35.756	34.499
*Err.	125.024	128.938	117.374

Table 3.2: AMD node

Table 3.3: Execution times for the discussed algorithms. The size of the input polynomials (s) equals to the number of bits of their coefficients (N). The error for the largest input in Kronecker-substitution method is due to memory allocation limits. (Times are in seconds.)

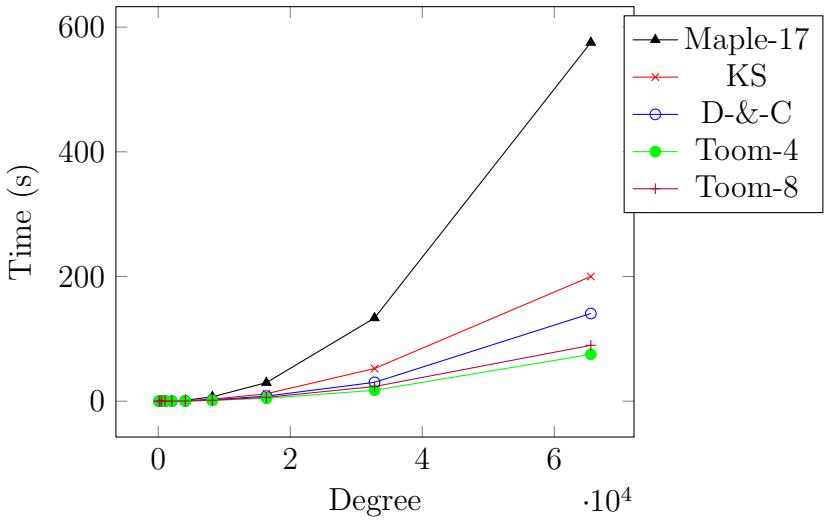


Table 3.4: Execution times for the discussed algorithms compared with Maple-17 which also uses Kronecker-substitution algorithm.

- The span of **4-way Toom-Cook** and **Divide & Conquer** algorithms are the same as it can be seen in our theoretical estimates in Propositions 10 and 9 which suggests it should be around 4-times better than **Kronecker substitution**'s work. But **4-way Toom-Cook** has significantly less work (see Corollary 6 and 5). Considering that the **Divide & Conquer** approach has 16 sub-problems, whereas **4-way Toom-Cook** has only 7 subproblems, we can justify why the execution time of **4-way Toom-Cook** is much better on the Intel node. On an ideal machine having a large number of cores, these two algorithms should perform very similarly.

- In any case, the work of **Kronecker substitution** is the best as it can be seen by comparing the estimates in Propositions 6, 7 and 8 (for instance, the work of the **Divide & Conquer** is almost 4 times larger for $d = 2$ which is consistent with Remark 6). This means that in the serial execution, the **Kronecker substitution** approach is the best. The reason is that in the integer multiplication of the GMP library relies on FFT-based algorithms which are asymptotically superior in term of algebraic complexity to all the other algorithms presented in this chapter. However, by dividing the work (even having a worse algebraic complexity), the other three algorithms can take advantage of parallel execution and outperform **Kronecker substitution** on both Intel and AMD nodes.

- The parallelism measured for the **Divide & Conquer** algorithms is close to 16 as it was predicted by Proposition 9 and Corollary 5; considering that there will be 16 sub-problems for $d = 2$, it shows we are almost reaching the linear speedup which suggest that our implementation is effective.

- Considering that we have 7 sub-problems in **4-way Toom-Cook**, and challenging evaluation/interpolation steps (in terms of parallelization), the measured parallelism (i.e. speedup factor) of 6.5 is satisfying (see Corollary 7). It is the same for **8-way Toom-Cook** having 15 sub-problems, but having a more complex evaluation/interpolation phase in the sense of algebraic complexity, see Corollary 9.

- Kronecker substitution is the fastest algorithm when the sizes of the input polynomials are less than 2^10. This is because there is not enough work for the parallel algorithms to beat Kronecker substitution.

4-Way-Toom-Cook Profiled execution times for the different stages of the **4-Way-Toom-Cook** algorithm are being shown in Table 3.6. Note that the dominant part of

N, s	Algorithm	Parallelism	Work	Span	Work/ks-work	Span/ks-work
2048	KS	1	795549545	795549545	1	1
	DnC	15.6	2706008620	173354669	3.401	0.218
	Toom-4	6.12	1107493602	180724889	1.392	0.227
	Toom-8	11.1	1165853687	104043963	1.465	0.131
4096	KS	1	4302927423	4302927423	1	1
	DnC	15.76	12587108834	798637458	2.925	0.186
	Toom-4	6.27	5213499242	831848170	1.211	0.193
	Toom-8	11.6	5211756088	448107503	1.211	0.104
8192	KS	1	16782031611	16782031611	1	1
	DnC	15.86	67219963043	4237361827	4.005	0.252
	Toom-4	6.46	28289524380	4382446030	1.686	0.261
	Toom-8	12.08	24448581048	2023752279	1.457	0.121
16384	KS	1	63573232166	63573232166	1	1
	DnC	15.87	253683811232	15980189635	3.990	0.251
	Toom-4	6.43	106591330172	16575876924	1.677	0.261
	Toom-8	12.58	121566206661	9662331241	1.912	0.151
32768	KS	1	269887534779	269887534779	1	1
	DnC	15.88	1003739269119	63197112068	3.719	0.234
	Toom-4	6.43	420041846756	65345935194	1.556	0.242
	Toom-8	12.49	462974961591	37063888492	1.715	0.137

Table 3.5: `Cilkview` analysis of the discussed algorithms for problems having different sizes (The size of the input polynomials (s) equals to the number of bits of their coefficients N). The columns *work*, and *span* are showing the number of instructions, and the *parallelism* is the ratio of *Work/Span*. The work and span of each algorithm are compared with those of Kronecker-substitution method which has the best work.

N, s	Division-&-Conversion	Evaluation	Integer Multiplication	Interpolation	Conversion-&-Merge
4096	0.056	0.01	0.119	0.026	0.064
8192	0.171	0.042	0.765	0.12	0.186
16384	0.603	0.155	2.612	0.518	0.627
32768	2.16	0.547	11.058	1.975	2.317
65536	6.511	2.08	49.328	8.179	8.724

Table 3.6: Profiled execution times for different sections of the algorithm in the 4-way Toom-Cook method. (Times are in seconds.)

N, s	Division-&-Conversion	Evaluation	Integer Multiplication	Interpolation	Conversion-&-Merge
4096	0.074	0.038	0.102	0.124	0.067
8192	0.289	0.189	0.522	0.475	0.175
16384	0.686	0.543	2.806	1.619	0.604
32768	2.218	2.251	9.843	6.14	2.29
65536	7.258	7.258	39.158	24.639	7.869

Table 3.7: Profiled execution times for different sections of the algorithm in the 8-way Toom-Cook method. (Times are in seconds.)

the algorithm is the integer-multiplication which is takes about 60% of the total running time.

8-Way-Toom-Cook Profiled execution times for the different stages of the **8-Way-Toom-Cook** algorithm are being shown in Table 3.7. Comparing these results with Table 3.6 indicates that the conversion stages are almost the same as before, whereas the evaluation and interpolation parts are much worse which confirms the theory analysis in Lemmas 6 and 4, and Propositions 10 and 11. However the multiplication times are better than that of **8-Way-Toom-Cook**, since the sizes of the sub-problems are smaller, see Lemma 5.

3.7 Conclusion

We investigated different methods for multiplying dense univariate polynomials with relatively large integer coefficients. Numerous tricks for parallelizing well-known methods and utilizing multi-core architectures have been used in our implementation, and large speed-up factors over the best known serial implementation have been observed. Besides, we presented precise algebraic complexity estimates for different algorithms (as well as span estimates) which confirm our experimental observations. However, all of these algorithms have a "static" parallelism, meaning that we divide the problem to a fixed

number of sub-problems. Thus, these algorithms cannot scale on an (ideal) machine with a large (infinite) number of cores. Moreover, we are relying on GMP's integer multiplication in all of our algorithm which causes us some overheads in data conversion. In Section 4 we investigate an FFT-based approach which has a dynamic parallelism and which does not rely on integer multiplication at all!

The source of the algorithms discussed in this chapter are freely available at the web site of the website of *Basic Polynomial Algebra Subprograms* (BPAS-Library) [5].

[5] `http://bpaslib.org/`

Chapter 4

Parallel polynomial multiplication via two convolutions on multi-core processors

We propose an FFT-based algorithm for multiplying dense polynomials with integer coefficients in a parallel fashion, targeting multi-core processor architectures. Complexity estimates and experimental results demonstrate the advantage of this new approach. We also show how parallelizing integer polynomial multiplication can benefit procedures for isolating real roots of polynomial systems.

This chapter is a joint work with M. Moreno Maza, C. Chen, N. Xie, and Y. Xie. The corresponding paper [10] is accepted at ISSAC 2014[1]

4.1 Introduction

Let $a(y), b(y) \in \mathbb{Z}[y]$ with degree at most $d-1$, for an integer $d \geq 1$. We aim at computing the product $c(y) := a(y)\, b(y)$. We propose an algorithm whose principle is sketched below. A precise statement of this algorithm is given in Section 4.2, while complexity results and implementation techniques appear in Sections 4.3 and 4.4.

1. Convert $a(y), b(y)$ to bivariate polynomials $A(x, y), B(x, y)$ over $\mathbb{Z}$ (by converting the integer coefficients of $a(y), b(y)$ to univariate polynomials of $\mathbb{Z}[x]$, where x is a new variable) such that $a(y) = A(\beta, y)$ and $b(y) = B(\beta, y)$ hold for some $\beta \in \mathbb{Z}$ (and, of course, such that we have $\deg(A, y) = \deg(a)$ and $\deg(B, y) = \deg(b)$).
2. Let $m > 4H$ be an integer, where H is the maximum absolute value of the coefficients

[1] `http://www.issac-conference.org/2014/papers.html`

of the integer polynomial $C(x,y) := A(x,y)B(x,y)$. The integer m and the polynomials $A(x,y), B(x,y)$ are built such that the polynomials $C^+(x,y) := A(x,y)B(x,y) \mod \langle x^K+1\rangle$ and $C^-(x,y) := A(x,y)B(x,y) \mod \langle x^K-1\rangle$ are computed over $\mathbb{Z}/m\mathbb{Z}$ via FFT techniques while the following equation holds over $\mathbb{Z}$:

$$C(x,y) = \frac{C^+(x,y)}{2}(x^K - 1) + \frac{C^-(x,y)}{2}(x^K + 1). \tag{4.1}$$

3. Finally, one recovers the product $c(y)$ by evaluating the above equation at $x = \beta$.

Of course, the polynomials $A(x,y), B(x,y)$ are also constructed such that their total bit size is proportional to that of $a(y), b(y)$, respectively. In our software experimentation, this proportionality factor ranges between 2 and 4. Moreover, the number β is a power of 2 such that evaluating the polynomials $C^+(x,y)$ and $C^-(x,y)$ (whose coefficients are assumed to be in binary representation) at $x = \beta$ amounts only to addition and shift operations. Further, for a software implementation on 64-bit computer architectures, the number m can be chosen to be either one machine word size prime p or a product p_1p_2 of two such primes. Therefore, in practice, the main arithmetic cost of the whole procedure is that of either two or four convolutions, those latter being required for computing $C^+(x,y)$ and $C^-(x,y)$. All the other arithmetic operations (for constructing $A(x,y), B(x,y)$ or evaluating the polynomials $C^+(x,y)$ and $C^-(x,y)$) are performed in single or double fixed precision at a cost which is proportional to that of reading/writing the byte vectors representing $A(x,y)$, $B(x,y)$, $C^+(x,y)$ and $C^-(x,y)$.

Theorem 1 below gives estimates for the work, the span and the cache complexity of the above algorithm as we have implemented it. Recall that our goal is not to obtain an algorithm which is asymptotically optimal for one of these complexity measures. Instead, our algorithm is designed to be practically faster, on multi-core architectures, than the other algorithms that are usually implemented for the same purpose of multiplying dense (univariate) polynomials with integer coefficients.

Theorem 1 *Let w be the number of bits of a machine word. Let N_0 be the maximum bit size of a coefficient among $a(y)$ and $b(y)$. There exist positive integers N, K, M, with $N = KM$ and $M \leq w$, such that the integer N is w-smooth (and so is K), we have $N_0 < N \leq N_0 + \sqrt{N_0}$ and the above algorithm for multiplying $a(y)$ and $b(y)$ has a work of $O(dK\log_2(dK)(\log_2(dK)+2M))$ word operations, a span of $O(K\log_2(d)\log_2(dK))$ word operations and incurs $O(\lceil dN/wL\rceil + \lceil(\log_2(dK)+2M)\rceil dK/L)$ cache misses, where double logarithmic factors are neglected for the span (and only for the span).*

A detailed proof of this result appears in Section 4.3. It follows from this result that this algorithm is not asymptotically as fast as an approach based on a combination

of Kronecker's substitution and Schönhage & Strassen, essentially by a $O(\log_2(d\,K))$ factor. This is because we directly reduce our multiplication to FFTs over a *small prime* finite field, instead of using the recursive construction of Schönhage & Strassen and its computation modulo Fermat numbers. However, by using multi-dimensional FFTs, we obtain a parallel algorithm which is practically efficient. We note also that the above span estimate is still linear in K, instead of the expected poly-log estimate. This is because we are controlling parallelism overheads by parallelizing only when this is cheap, see Section 4.2 for details. Nevertheless our ratio work to span is in the order of d that is, proportional to the input size. In contrast, parallelizing a k-way Toom Cook algorithms (by executing concurrently the point-wise multiplication, see Chapter 3) yields only a ratio work to span in the order of k, that is, a very limited scalability. Finally, our cache complexity estimate is sharp. Indeed, we control finely all intermediate steps with this respect, see Section 4.3.

As mentioned above, our code is publicly available as part of the BPAS library[2]. To illustrate the benefits of a parallelized dense univariate polynomial multiplication, we integrated our code into the univariate real root isolation code presented in [11] together with a parallel version of Algorithm (E) from [65] for Taylor Shifts. The results reported in Section 4.5 show that this integration has substantially improved the performance of our real root isolation code.

4.2 Multiplying integer polynomials via two convolutions

Notations. We write

$$a(y) = \sum_{i=0}^{d-1} a_i y^i, \quad b(y) = \sum_{i=0}^{d-1} b_i y^i \quad \text{and} \quad c(y) = \sum_{i=0}^{2d-2} c_i y^i, \tag{4.2}$$

where a_i, b_i, c_i are integers. Let N be a non-negative integer such that each coefficient α of a or b satisfies

$$-2^{N-1} \;\leq\; \alpha \;\leq\; 2^{N-1} - 1 \tag{4.3}$$

Therefore, using two's complement, every such coefficient α can be encoded with N bits. In addition, the integer N is chosen such that N writes

$$N = KM \quad \text{with} \quad K \neq N \quad \text{and} \quad M \neq N, \tag{4.4}$$

[2]BPAS library: `http://www.bpaslib.org/`

for $K, M \in \mathbb{N}$. It is helpful to think of K as a power of 2, and M as a small number, say less than w, where w is the bit-size of a machine word. For the theoretical analysis of our algorithm, we shall simply assume that N is a w-smooth integer, that is, none of its prime factors is greater than w. The fact that one can choose such an N will be discussed in Section 4.3.1. We denote by $\mathsf{DetermineBase}(a, b, w)$ a function call returning (N, K, M) satisfying the constraints of (4.4) and such that N is a w-smooth integer, minimum with the constraints of (4.3).

From $\mathbb{Z}[y]$ to $\mathbb{Z}[x, y]$. Let $(N, K, M) \coloneqq \mathsf{DetermineBase}(a, b, w)$ and define $\beta = 2^M$. We write

$$a_i = \sum_{j=0}^{K-1} a_{i,j}\beta^j, \quad \text{and} \quad b_i = \sum_{j=0}^{K-1} b_{i,j}\beta^j, \tag{4.5}$$

where each $a_{i,j}$ and $b_{i,j}$ are signed integers in the closed range $[-2^{M-1}, 2^{M-1} - 1]$. Then, we define

$$A(x, y) = \sum_{i=0}^{d-1} \Big(\sum_{j=0}^{K-1} a_{i,j}x^j \Big) y^i, B(x, y) = \sum_{i=0}^{d-1} \Big(\sum_{j=0}^{K-1} b_{i,j}x^j \Big) y^i, \tag{4.6}$$

and

$$C(x, y) \coloneqq A(x, y)B(x, y) \quad \text{with} \quad C(x, y) = \sum_{i=0}^{2d-2} \Big(\sum_{j=0}^{2K-2} c_{i,j}x^j \Big) y^i \tag{4.7}$$

where $c_{i,j} \in \mathbb{Z}$. We denote by $\mathsf{BivariateRepresentation}(a, N, K, M)$ a function call returning $A(x, y)$ as defined above. Observe that the polynomial $c(y)$ is clearly recoverable from $C(x, y)$ as

$$\begin{aligned} C(\beta, y) &= A(\beta, y)B(\beta, y) \\ &= a(y)b(y) \\ &= c(y). \end{aligned} \tag{4.8}$$

The following sequence of equalities will be useful:

$$\begin{aligned} C(x, y) &= A(x, y)\, B(x, y) \\ &= \Big(\textstyle\sum_{i=0}^{d-1} \big(\sum_{j=0}^{K-1} a_{i,j}x^j \big) y^i \Big) \Big(\sum_{i=0}^{d-1} \big(\sum_{j=0}^{K-1} b_{i,j}x^j \big) y^i \Big) \\ &= \textstyle\sum_{i=0}^{2d-2} \Big(\sum_{\ell+m=i} \Big(\sum_{k=0}^{K-1} a_{\ell,k}x^k \Big) \Big(\sum_{h=0}^{K-1} b_{m,h}x^h \Big) \Big) y^i \\ &= \textstyle\sum_{i=0}^{2d-2} \Big(\sum_{\ell+m=i} \Big(\sum_{j=0}^{2K-2} \Big(\sum_{k+h=j} a_{\ell,k}b_{m,h} \Big) x^j \Big) \Big) y^i \\ &= \textstyle\sum_{i=0}^{2d-2} \Big(\sum_{j=0}^{2K-2} c_{i,j}x^j \Big) y^i \\ &= \textstyle\sum_{j=0}^{2K-2} \Big(\sum_{i=0}^{2d-2} c_{i,j}y^i \Big) x^j \\ &= \textstyle\sum_{j=0}^{K-1} \Big(\sum_{i=0}^{2d-2} c_{i,j}y^i \Big) x^j \;+\; x^K \sum_{j=0}^{K-2} \Big(\sum_{i=0}^{2d-2} c_{i,j+K}y^i \Big) x^j, \end{aligned} \tag{4.9}$$

where we define

$$c_{i,j} \;\coloneqq\; \sum_{\ell+m=i} \sum_{k+h=j} a_{\ell,k} b_{m,h}, 0 \le i \le 2d-2, 0 \le j \le 2K-2, \tag{4.10}$$

with the convention

$$c_{i,2K-1} \;\coloneqq\; 0 \quad \text{for} \quad 0 \le i \le 2d-2. \tag{4.11}$$

Since the modular products $A(x,y)B(x,y) \mod \langle x^K+1\rangle$ and $A(x,y)B(x,y) \mod \langle x^K-1\rangle$ are of interest, we define the bivariate polynomial over $\mathbb{Z}$

$$C^+(x,y) \;\coloneqq\; \sum_{i=0}^{2d-2} c_i^+(x)\, y^i \quad \text{where} \quad c_i^+(x) \coloneqq \sum_{j=0}^{K-1} c_{i,j}^+ \, x^j \quad \text{and} \quad c_{i,j}^+ \coloneqq c_{i,j} - c_{i,j+K} \tag{4.12}$$

and the bivariate polynomial over $\mathbb{Z}$

$$C^-(x,y) \;\coloneqq\; \sum_{i=0}^{2d-2} c_i^-(x)\, y^i \quad \text{where} \quad c_i^-(x) \coloneqq \sum_{j=0}^{K-1} c_{i,j}^- \, x^j \quad \text{and} \quad c_{i,j}^- \coloneqq c_{i,j} + c_{i,j+K}. \tag{4.13}$$

Thanks to Equation (4.9), we observe that we have

$$\begin{array}{rcl} C^+(x,y) & \equiv & A(x,y)B(x,y) \mod \langle x^K+1\rangle, \\ C^-(x,y) & \equiv & A(x,y)B(x,y) \mod \langle x^K-1\rangle. \end{array} \tag{4.14}$$

Since the polynomials x^K+1 and x^K-1 are coprime for all integer $K \ge 1$, we deduce Equation (4.1).

Since β is a power of 2, evaluating the polynomials $C^+(x,y)$, $C^-(x,y)$ and thus $C(x,y)$ (whose coefficients are assumed to be in binary representation) at $x = \beta$ amounts only to addition and shift operations. A precise algorithm is described in Section 4.2.1. Before that, we turn our attention to computing $C^+(x,y)$ and $C^-(x,y)$ via FFT techniques.

From $\mathbb{Z}[x,y]$ to $\mathbb{Z}/m[x,y]$. From Equation (4.14), it is natural to consider using FFT techniques for computing both $C^+(x,y)$ and $C^-(x,y)$. Thus, in order to compute over a finite ring supporting FFT, we estimate the size of the coefficients of $C^+(x,y)$ and $C^-(x,y)$. Recall that for $0 \le i \le 2d-2$, we have

$$\begin{array}{rcl} c_{i,j}^+ & = & c_{i,j} - c_{i,j+K} \\ & = & \sum_{\ell+m=i} \sum_{k+h=j} a_{\ell,k} b_{m,h} - \sum_{\ell+m=i} \sum_{k+h=j+K} a_{\ell,k} b_{m,h} \end{array} \tag{4.15}$$

Since each $a_{\ell,k}$ and each $b_{m,h}$ has bit-size at most M, the absolute value of each coefficient $c^+_{i,j}$ is bounded over by $2\,d\,K\,2^{2M}$. The same holds for the coefficients $c^-_{i,j}$. Since the coefficients $c^+_{i,j}$ and $c^-_{i,j}$ may be negative, we consider a positive integer m such that

$$m > 4\,d\,K\,2^{2M}. \tag{4.16}$$

From now on, depending on the context, we freely view the coefficients $c^+_{i,j}$ and $c^-_{i,j}$ either as elements of $\mathbb{Z}$ or as elements of $\mathbb{Z}/m$. Indeed, the integer m is large enough for this identification and we use the integer interval $[-\frac{m-1}{2}, \frac{m-1}{2}]$ to represent the elements of $\mathbb{Z}/m$.

One may want to choose the integer m such that the ring $\mathbb{Z}/m$ admits appropriate primitive roots of unity for computing the polynomials $C^+(x,y)$ and $C^-(x,y)$ via cyclic convolution and negacylic convolution in $\mathbb{Z}/m[x,y]$, see details below. Finally, we observe that in a computer program, the bound constraint on m would be achieved by picking a number m whose bit-size exceeds $2 + \lceil \log_2(d\,K) \rceil + 2M$.

From $\mathbb{Z}/m[x,y]$ to $\mathbb{Z}/p[x,y]$, for a prime number p. Since an integer m as specified above may not be represented by a single machine word, it is natural to adopt a "small prime" approach by means of the Chinese Remaindering Algorithm (CRA). This is the point of view that we shall follow in the rest of this section. To be more specific, we choose prime numbers $p_1, \ldots, p_e$ of machine word size such that their product satisfies

$$p_1 \cdots p_e \;>\; 4\,d\,K\,2^{2M}. \tag{4.17}$$

and such that the following divisibility relations hold

$$2^r \;\mid\; p_i - 1 \quad \text{and} \quad K \;\mid\; p_i - 1, \tag{4.18}$$

for all $i = 1 \cdots e$, where $r = \lceil \log_2(2d-1) \rceil$. Note that we require that 2^r (instead of $2d-1$) divides $p-1$ so as to apply the *Truncated FFT algorithm* of [63]; we shall return to this latter point in Section 4.3. This allows us to compute

$$C^+_i(x,y) \coloneqq A(x,y)B(x,y) \mod \langle x^K+1, p_i \rangle \text{ and } C^-_i(x,y) \coloneqq A(x,y)B(x,y) \mod \langle x^K-1, p_i \rangle, \tag{4.19}$$

via FFT/TFT techniques, for all $i = 1 \cdots e$. Then, we combine $C^+_1(x,y), \ldots, C^+_e(x,y)$ (resp. $C^-_1(x,y), \ldots, C^-_e(x,y)$) by CRA in order to recover $C^+(x,y)$ (resp. $C^-(x,y)$) over $\mathbb{Z}$, denoting this computation by $\mathsf{CombineBivariate}\ (C^+_1(x,y), \ldots, C^+_e(x,y), p_1, \ldots, p_e)$ (resp.

CombineBivariate $(C_1^-(x,y),\ldots,C_e^-(x,y),p_1,\ldots,p_e)$). Finally, we denote by RecoveryPrimes(d,K,M) a sequence of prime numbers $p_1,\ldots,p_e$ satisfying (4.17) and (4.18) with e minimum.

Cyclic and negacylic convolutions in $\mathbb{Z}/p[x,y]$. Let p be one prime returned by RecoveryPrimes(d,K,M). Let θ be a $2K$-th primitive root of unity in $\mathbb{Z}/p$. We define $\omega = \theta^2$, thus ω is a K-th primitive root in $\mathbb{Z}/p$. For univariate polynomials $u(x), v(x) \in \mathbb{Z}[x]$ of degree at most $K-1$, computing $u(x)\,v(x) \mod \langle x^K-1, p\rangle$ via FFT is a well-known operation, see Algorithm 8.16 in [29]. Using the *row-column* algorithm for two-dimensional FFT, one can compute $C^-(x,y) \equiv A(x,y)B(x,y) \mod \langle x^K-1,p\rangle$, see [47, 46] for details. We denote by CyclicConvolution(A,B,K,p) the result of this calculation.

We turn our attention to the negacylic convolution, namely $A(x,y)B(x,y) \mod \langle x^K+1,p\rangle$. We observe that the following holds:

$$C^+(x,y) \equiv A(x,y)B(x,y) \mod \langle x^K+1,p\rangle \iff C^+(\theta x,y) \equiv A(\theta x,y)B(\theta x,y) \mod \langle x^K-1,p\rangle \tag{4.20}$$

Thus, defining

$$C'(x,y) \coloneqq C^+(\theta x,y), \quad A'(x,y) \coloneqq A(\theta x,y) \text{ and } B'(x,y) \coloneqq B(\theta x,y), \tag{4.21}$$

we are led to compute

$$A'(x,y)B'(x,y) \mod \langle x^K-1,p\rangle, \tag{4.22}$$

which can be done as CyclicConvolution(A',B',K,p). Then, the polynomial $C^+(x,y) \mod \langle x^K-1,p\rangle$ is recovered from $C'(x,y) \mod \langle x^K-1,p\rangle$ as

$$C^+(x,y) \equiv C'(\theta^{-1}x,y) \mod \langle x^K-1,p\rangle, \tag{4.23}$$

and we denote by NegacyclicConvolution(A,B,K,p) the result of this process. We dedicate a section to the final step of our algorithm, that is, the recovery of the product $c(y)$ from the polynomials $C^+(x,y)$ and $C^-(x,y)$.

4.2.1 Recovering $c(y)$ from $C^+(x,y)$ and $C^-(x,y)$

We naturally assume that all numerical coefficients are stored in binary representation. Thus, recovering $c(y)$ as $C(\beta,y)$ from Equation (4.1) involves only additions and shift operations. Indeed, β is a power of 2. Hence, the algebraic complexity of this recovery is essentially proportional to the sum of the bit sizes of $C^+(x,y)$ and $C^-(x,y)$. Therefore, the arithmetic count for computing these latter polynomials (by means of cyclic and nega-

cyclic convolutions) dominates that of recovering $c(y)$. Nevertheless, when implemented on a modern computer hardware, this recovery step may contribute in a significant way to the total running time. The reasons are that both the convolution computations and recovery steps incur similar amounts of cache misses and that the memory traffic implied by those cache misses are a significant portion of the total running time.

We denote by $\mathsf{RecoveringProduct}(C^+(x,y), C^-(x,y), \beta)$ a function call recovering $c(y)$ from $C^+(x,y)$, $C^-(x,y)$ and $\beta = 2^M$. We start by stating below a simple procedure for this operation:

1. $u(y) := C^+(\beta, y)$
2. $v(y) := C^-(\beta, y)$
3. $c(y) := \frac{u(y)+v(y)}{2} + \frac{-u(y)+v(y)}{2}\, 2^N$

To further describe this operation and, later on, in order to discuss its cache complexity and parallelization, we specify the data layout. From the definition of the prime number sequence $(p_1, \ldots, p_e)$, we can assume that each coefficient of the bivariate polynomials $C^+(x,y)$, $C^-(x,y)$ can be encoded within e machine words. Thus, we assume that $C^+(x,y)$ (resp. $C^-(x,y)$) is represented by an array of $(2d-1)\,K\,e$ machine words such that, for $0 \le i \le 2d-2$ and $0 \le j \le K-1$, the coefficient $c^+_{i,j}$ (resp. $c^-_{i,j}$) is written between the positions $(K\,i+j)e$ and $(K\,i+j)e+e-1$, inclusively. Thus, this array can be regarded as the encoding of a 2-D matrix whose i-th row is $c^+_i(x)$ (resp. $c^-_i(x)$). Now, we write

$$u(y) \coloneqq \sum_{i=0}^{2d-2} u_i y^i \quad \text{and} \quad v(y) \coloneqq \sum_{i=0}^{2d-2} v_i y^i \tag{4.24}$$

thus, from the definition of $u(y)$, $v(y)$, for $0 \le i \le 2d-2$, we have

$$u_i = \sum_{j=0}^{K-1} c^+_{i,j}\, 2^{M\,j} \quad \text{and} \quad v_i = \sum_{j=0}^{K-1} c^-_{i,j}\, 2^{M\,j}. \tag{4.25}$$

Denoting by H^+, H^- the largest absolute value of a coefficient in $C^+(x,y)$, $C^-(x,y)$, we deduce

$$|u_i| \;\le\; H^+ \frac{\left((2^M)^K - 1\right)}{2^M - 1} \quad \text{and} \quad |v_i| \;\le\; H^- \frac{\left((2^M)^K - 1\right)}{2^M - 1}. \tag{4.26}$$

From the discussion justifying Relation (4.16), we have

$$H^+,\, H^- \;\le\; 2\,d\,K\,2^{2M}, \tag{4.27}$$

and with (4.26) we derive

$$|u_i|,\, |v_i| \;\le\; 2\,d\,K\,2^{M+N} \tag{4.28}$$

Indeed, recall that $N = K\,M$ holds. We return to the question of data layout. Since each of $c^+_{i,j}$ or $c^-_{i,j}$ is a signed integer fitting within e machine words, it follows from (4.26) that each of the coefficients u_i, v_i can be encoded within

$$f \;:=\; \lceil N/w \rceil + e \tag{4.29}$$

machine words. Hence, we store each of the polynomials $u(y), v(y)$ in an array of $(2d-1) \times f$ machine words such that the coefficient in degree i is located between position $f\,i$ and position $f\,(i+1)-1$. Finally, we come to the computation of $c(y)$. We have

$$c_i \;=\; \frac{u_i + v_i}{2} \;+\; 2^N\;\frac{v_i - u_i}{2}, \tag{4.30}$$

which implies

$$|c_i| \;\leq\; 2\,d\,K\,2^{M+N}(1+2^N). \tag{4.31}$$

Relation (4.31) implies that the polynomial $c(y)$ can be stored within an array of $(2d-1) \times 2f$ machine words. Of course, a better bound than (4.31) can be derived by simply expanding the product $a(y)\,b(y)$, leading to

$$|c_i| \;\leq\; d\,2^{2N-2}. \tag{4.32}$$

The ratio between the two bounds given by (4.31) and (4.32) tells us that the extra amount of space required by our algorithm is $O(\log_2(K)+M)$ bits per coefficient of $c(y)$. Recall that, in practice, we have $M \leq w$. Hence, this extra space amount can be regarded as small and thus satisfactory.

4.2.2 The algorithm in pseudo-code

With the procedures that were defined in this section, we are ready to state our algorithm for integer polynomial multiplication.

Input: $a(y), b(y) \in \mathbb{Z}[y]$ and z a small integer such that $2 \leq z \leq w$ and $d := \max(\deg(a), \deg(b)) + 1$.
Output: the product $a(y)\,b(y)$

1: $(N, K, M) := \mathsf{DetermineBase}(a(y), b(y), z)$
2: $A(x,y) := \mathsf{BivariateRepresentation}(a(y), N, K, M)$
3: $B(x,y) := \mathsf{BivariateRepresentation}(b(y), N, K, M)$
4: $p_1, \ldots, p_e := \mathsf{RecoveryPrimes}(d, K, M)$
5: **for** $i \in 1 \cdots e$ **do**

6: $C_i^-(x,y) := \mathsf{CyclicConvolution}(A, B, K, p_i)$

7: $C_i^+(x,y) := \mathsf{NegacyclicConvolution}(A, B, K, p_i)$

8: **end do**

9: $C^+(x,y) := \mathsf{CombineBivariate}(C_i^+(x,y), p_i, i = 1 \cdots e)$

10: $C^-(x,y) := \mathsf{CombineBivariate}(C_i^-(x,y), p_i, i = 1 \cdots e)$

11: $c(y) := \mathsf{RecoveringProduct}(C^+(x,y), C^-(x,y), 2^M)$

12: **return** $c(y)$

In order to analyze the complexity of our algorithm, it remains to specify the data layout for $a(y)$, $b(y)$, $A(x,y)$, $B(x,y)$, $C_1^+(x,y), \ldots, C_e^+(x,y)$, $C_1^-(x,y), \ldots, C_e^-(x,y)$. Note that this data layout question was handled for $C^-(x,y)$, $C^+(x,y)$ and $c(y)$ in Section 4.2.1.

In the sequel, we view $a(y)$, $b(y)$ as *dense* in the sense that each of their coefficients is assumed to be of essentially the same size. Hence, from the definition of N, see Relation (4.3), we assume that each of $a(y)$, $b(y)$ is stored within an array of $d \times \lceil N/w \rceil$ machine words such that the coefficient in degree i is located between positions $\lceil N/w \rceil i$ and $\lceil N/w \rceil (i+1) - 1$.

Finally, we assume that each of the bivariate integer polynomials $A(x,y)$, $B(x,y)$ is represented by an array of $d \times K$ machine words whose $(K \times i + j)$-th coefficient is $a_{i,j}$, $b_{i,j}$ respectively, for $0 \le i \le d-1$ and $0 \le j \le K-1$. The same row-major layout is used for each of the $d \times K$ machine word arrays representing the bivariate modular polynomials $C_1^+(x,y), \ldots, C_e^+(x,y)$, $C_1^-(x,y), \ldots, C_e^-(x,y)$.

4.2.3 Parallelization

One of the design goals of our algorithm is to take advantage of the parallel FFT-based routines for multiplying dense multivariate polynomials over finite fields that have been proposed in [46, 47]. To be precise, these routines provide us with a parallel implementation of the procedure CyclicConvolution, from which we easily derive a parallel implementation of NegacyclicConvolution.

Lines **1** and **4** can be ignored in the analysis of the algorithm. Indeed, as we shall explain in Section 4.4, one can simply implement DetermineBase and RecoveryPrimes by look-up in precomputed tables. For instance, Table B.1 in Appendix B is being used for determining the base using 2 primes.

For parallelizing Lines **2** and **3**, it is sufficient in practice to convert concurrently all the coefficients of $a(y)$ and $b(y)$ to univariate polynomials of $\mathbb{Z}[y]$. Similarly, for parallelizing Lines **9** and **10**, one can view each of the polynomials $C^-(x,y)$, $C^+(x,y)$ (and

their modular images) as polynomials in y, then processing their coefficients concurrently. For parallelizing Line **11** it is sufficient again to compute concurrently the coefficients of $u(y)$, $v(y)$ and then those of $c(y)$. Finally, the parallel for-loop of **5** can be converted into a parallel for-loop, while each call to CyclicConvolution and NegacyclicConvolution relies on parallel FFT as mentioned.

4.3 Complexity analysis

In this section, we analyze the algorithm stated in Section 4.2.2. We estimate its *work* and *span* as defined in the *fork-join concurrency model* introduced in Section 2.1.1. Since the fork-join model has no primitive constructs for defining parallel for-loops, each of those loops is simulated by a divide-and-conquer procedure for which non-terminal recursive calls are forked, see [43] for details. Hence, in the fork-join model, the bit-wise comparison of two vectors of size n has a span of $O(\log(n))$ bit operations. This is actually the same time estimate as in the Exclusive-Read-Exclusive-Write PRAM [60, 31] model, but for a different reason.

We shall also estimate the *cache complexity* [22] of the serial counterpart of our algorithm for an ideal cache of Z words and with L words per cache line. Note that the ratio work to cache complexity indicates how an algorithm is capable of re-using cached data. Hence the larger is the ratio, the better.

We denote by W_i, S_i, Q_i the work, span and cache complexity of Line **i** in the algorithm stated in Section 4.2.2. As mentioned before, we can ignore the costs of Lines **1** and **4**. Moreover, we can use W_2, S_2, Q_2 as estimates for W_3, S_3, Q_3, respectively. Similarly, we can use the estimates of Lines **6** and **9** for the costs of Lines **7** and **10**, respectively. Thus, we only analyze the costs of Lines **2**, **6**, **9** and **11**.

Analysis of BivariateRepresentation$(a(y), N, K, M)$**.** Converting each coefficient of $a(y)$ to a univariate polynomial of $\mathbb{Z}[x]$ requires $O(N)$ bit operations thus

$$W_2 \in O(d\,N) \quad \text{and} \quad S_2 \in O(\log(d)\,N). \tag{4.33}$$

In the latter, the $\log(d)$ factor comes from the fact that the parallel for-loop corresponding to "for each coefficient of $a(y)$" is executed as a recursive function with $O(\log(d))$ nested recursive calls. Considering now the cache complexity, and taking into account the data layout specified in Section 4.2.2, one observes that converting $a(y)$ to $A(x, y)$ leads to $O(\lceil dN/wL \rceil + 1)$ cache misses for reading $a(y)$ and $O(\lceil dK/L \rceil + 1)$ cache misses for writing

$A(x,y)$. Therefore, we have

$$Q_2 \in O(\lceil dN/wL \rceil + \lceil dK/L \rceil + 1). \tag{4.34}$$

Analysis of CyclicConvolution(A, B, K, p_i)**.** Following the techniques developed in [46, 47], we compute $A(x,y)B(x,y) \mod \langle x^K - 1, p_i \rangle$ by 2-D FFT of format $K \times (2d-1)$. In the direction of y, we use van der Hoeven's TFT algorithm [63], thus the only constraint on d is the fact that fact 2^r divides $p-1$ where $r = \lceil \log_2(2d-1) \rceil$. In the direction of x, the convolutions (i.e. products in $\mathbb{Z}[x]/\langle x^K - 1, p_i \rangle$) require to compute (non-truncated) FFTs of size K. Using the Algorithm of Wang and Zhu [67], this can be done for all K dividing $p_i - 1$, but such a convolution requires $O(K\log^2(K))$ operations in $\mathbb{Z}/p_i$ instead of the more desirable $O(K\log(K))$. Alternatively, one can assume that K is highly composite so as to take advantage of faster FFT algorithms such as Cooley-Tukey's algorithm [15]. The Cooley-Tukey factorization used with small radices can be argued to have cache-oblivious locality benefits on systems with hierarchical memory [20]. For this reason, we further assume that N, and thus K, are z-smooth integers where z is small, say $z \leq w$. In Section 4.3.1, we explain the reduction to this hypothesis. Consequently, we apply the complexity results of [46], leading to a work (resp. a span) of $O(s\log(s))$ (resp. $O(\log(s))$) operations in $\mathbb{Z}/p_i$, where $s = (2d-1)K$. Since p_i is of machine word size, we deduce

$$W_6 \in O(dK\log(dK)) \quad \text{and} \quad S_6 \in O(\log(dK)) \tag{4.35}$$

bit operations and, from the results of [22], we have

$$Q_6 \in O(1 + (dK/L)(1 + \log_Z(dK))). \tag{4.36}$$

Analysis of CombineBivariate$(C_1^+(x,y), \ldots, C_e^+(x,y), p_1, \ldots, p_e)$**.** Applying subproduct tree techniques (see Section 5) each coefficient of $C^+(x,y)$ is obtained from the corresponding coefficients of $C_1^+(x,y), \ldots, C_e^+(x,y)$ within $O(\mathsf{M}(\log_2(\mu))\,\mathrm{loglog}_2\,(\mu))$ word operations, where μ is the product of the prime numbers $p_1, \ldots, p_e$ and $\ell \longmapsto \mathsf{M}(\ell)$ is a multiplication time for multiplying two integers with at most ℓ bits. For this multiplication time, we use the estimates of Fürer [24] or De, Kurur, Saha and Saptharishi [17]. From (4.17), we can assume $\log_2(\mu) \in \Theta(\log_2(dK) + 2M)$. Since $C^+(x,y)$ has (at most) $2d-1$ coefficients along y and K coefficients along x, and since $2M$ is less than the

number of bits of a machine word, we obtain

$$W_9 \in O(dK\,\mathsf{M}(\log_2(d\,K))\,\mathrm{loglog}_2(d\,K)) \text{ and } S_9 \in O(\log_2(d)\,K\,\mathsf{M}(\log_2(d\,K))\,\mathrm{loglog}_2(d\,K)) \tag{4.37}$$

bit operations. Recall, indeed, that we only parallelize computations along y, since parallelizing along x would be practically counterproductive due to parallelization overheads (scheduling costs, tasks migration). Turning to the cache complexity of Line **9**, we observe that, once a cache-line from each of the polynomials $C_1^+(x,y),\ldots,C_e^+(x,y),C^+(x,y)$ has been loaded into the cache, then L recombinations of coefficients can be performed without evicting any of those cache-lines from the cache. Indeed, accommodating e coefficients from $C_1^+(x,y),\ldots,C_e^+(x,y)$ (taking into account that each coefficient is a word on one cache-line of L words) requires $Z_e = eL$ words of cache. Next, accommodating e consecutive coefficients from $C^+(x,y)$, (taking into account possible misalignments in memory) requires $Z^+ = L(\lceil e/L\rceil + 1) \leq e + L + 1$ words of cache. Recall that we choose $p_1,\ldots,p_e$ such that $\mu = p_1,\ldots,p_e \geq 4dK2^{2M}$ and, thus,

$$\begin{array}{rcl} e & = & \lfloor \log_2(\mu)/w \rfloor + 1 \\ & \leq & 2 + \lceil \log_2(dK) \rceil + 2M + 1. \end{array} \tag{4.38}$$

Hence, we require $(4+\lceil\log_2(dK)\rceil+2M)(L+1)$ words of cache for storing those coefficients. Since, for our algorithm, the quantity $4+\lceil\log_2(dK)\rceil+2M$ represents a few machine words (typically 2 for the largest examples tested on a computer with 1/4 Tera bytes of RAM) our assumption is in fact clearly covered by the standard *tall cache assumption* [22], namely $Z \in \Omega(L^2)$. Under this assumption, elementary calculations leads to

$$\begin{array}{rcl} Q_9 & \in & O(e(\lceil dK/L\rceil + 1) + \lceil (2d-1)\times K \times e/L\rceil + 1) \\ & \in & O((\lceil \log_2(dK)\rceil + 2M)(\lceil dK/L\rceil + 1)). \end{array} \tag{4.39}$$

Analysis of $\mathsf{RecoveringProduct}(C^+(x,y),C^-(x,y),2^M)$**.** Converting each coefficient of $u(y)$ and $v(y)$ from the corresponding coefficients $C^+(x,y)$ and $C^-(x,y)$ requires $O(K\,e)$ bit operations. Then, computing each coefficient of $c(y)$ requires $O(N + e\,w)$ bit operations. Thus we have

$$W_{11} \in O(d\,(K\,e + N + e)) \text{ and } S_{11} \in O(\log(d)\,(K\,e + N + e)) \tag{4.40}$$

word operations. Converting $C^+(x,y)$, $C^-(x,y)$ to $u(y)$, $v(y)$ leads to $O(\lceil d\,K\,e/L\rceil + 1)$ cache misses for reading $C^+(x,y)$, $C^-(x,y)$ and $O(\lceil d(N/w + e)/L\rceil + 1)$ cache misses for writing $u(y)$, $v(y)$. This second estimate holds also for the total number of cache misses

for computing $c(y)$ from $u(y)$, $v(y)$. Thus, we have

$$Q_{11} \in O(\lceil d\,(K\,e + N + e)/L \rceil + 1). \tag{4.41}$$

We note that the quantity $K\,e + N + e$ can be replaced in above asymptotic upper bounds by $K(\log_2(d\,K) + 3\,M)$.

4.3.1 Smooth integers in short intervals

We review a few facts about smooth integers and refer to Andrew Granville's survey [33] for details and proofs. Let $S(x, y)$ be the set of integers up to x, all of whose prime factors are less or equal to y (such integers are called "y-smooth"), and let $\Psi(x, y)$ be the number of such integers. It is a remarkable result that for any fixed $u \geq 1$, the proportion of the integers up to x, that only have prime factors less or equal to $x^{1/u}$, tends to a nonzero limit as x escapes to infinity. This limit, denoted by $\rho(u)$ is known as the Dickman-de Bruijn ρ-function. To be precise, Dickman's result states the following asymptotic equivalence

$$\Psi(x, y) \;\sim\; x\,\rho(u) \;\text{ as }\; x \longrightarrow \infty \tag{4.42}$$

where $x = y^u$. Unfortunately, one cannot write down a useful, simple function that gives the value of $\rho(u)$ for all u. Therefore, upper bounds and lower bounds for $\Psi(x, y)$ are of interest, in particular for the purpose of analyzing algorithms where smooth integers play a key role, as for the multiplication algorithm of Section 4.2. A simple lower bound is given by

$$\Psi(x, y) \;\geq\; x^{1-\log(\log(x))/\log(y)} \tag{4.43}$$

for all $x \geq y \geq 2$ and $x \geq 4$, from which, one immediately deduces

$$\Psi(x, \log^2(x)) \;\geq\; \sqrt{x}. \tag{4.44}$$

However, it is desirable to obtain statements which could imply inequalities like $\Psi(x + z, y) - \Psi(x, y) > 0$ for all $x \geq z$ and for y arbitrary small. Indeed, such inequality would mean that, for all x, y, a "short interval" around x contains at least one y-smooth integer. In fact, the following relation is well-known for all $x \geq z \geq x/y^{1-o(1)}$:

$$\Psi(x + z, y) - \Psi(x, y) \;\sim\; \frac{z}{x}\Psi(x, y) \;\sim\; z\rho(u). \tag{4.45}$$

but does not meet our needs since it does allow us to make y arbitrary small independently of x and z, while implying $\Psi(x + z, y) - \Psi(x, y) > 0$.

Nevertheless, in 1999, Ti Zuo Xuan [68] proved that, under the Riemann Hypothesis (RH), for any $\varepsilon > 0,\ \delta > 0$ there exists x_0 such that for all $x \geq x_0$ the interval $(x, x + y]$, for $\sqrt{x}(\log(x))^{1+\epsilon} \leq y \leq x$, contains an integer having no prime factors exceeding x^{δ}. Moreover, in 2000, Granville has conjectured that for all $\alpha > 0$ there exists x_0 such that for all $x \geq x_0$, we have

$$\Psi(x + \sqrt{x}, x^{\alpha}) - \Psi(x, x^{\alpha}) > 0. \tag{4.46}$$

In the sequel of this section, we shall assume that either RH or Granville's conjecture holds. In fact, we have verified Relation (4.46) experimentally for all $x \leq 2^{23}$. This is by far sufficient for the purpose of implementing our multiplication algorithm and verifying the properties of the positive integers N, K, M stated in Theorem 1. Moreover, for all the values x that we have tested, the following holds

$$\Psi(2x, 23) - \Psi(x, 23) > 0. \tag{4.47}$$

4.3.2 Proof of Theorem 1

The fact that one can find positive integers N, K, M with $N = K\,M$ and $M \leq w$, such that the integer N is w-smooth and $N_0 < N \leq N_0 + \sqrt{N_0}$ follows from the discussion of Section 4.3.1. Next, recall that analyzing our algorithm reduces to analyzing Lines **2**, **6**, **9** and **12**, noting that **6** is the block of a for loop with e iterations. Recall also that Relation (4.38) gives an upper bound for e.

Based on the results obtained above for W_2, W_6, W_9, W_{11} with Relations (4.33), (4.35), (4.37), (4.40) respectively, it follows that the estimate for the work of the whole algorithm is given by $e\,W_6$, leading to the result in Theorem 1. Meanwhile the span of the whole algorithm is given by S_9. Neglecting the double logarithmic factor $\mathrm{loglog}_2(d\,K))$ and the iterated logarithmic factor from $\mathsf{M}(\log_2(d\,K))$, one obtains the result in Theorem 1. Finally, the cache complexity estimate in Theorem 1 comes from adding up Q_2, $e \times Q_6$, Q_9, Q_{11} and simplifying.

4.4 Implementation

We have implemented the proposed algorithm using CilkPlus targeting multi-cores. The code for Convert-in and Convert-out steps can be found in Appendix A.

4.5 Experimentation

We use the multi-threaded language `CilkPlus` [43] and compiled our code with the `CilkPlus` branch of GCC[3]. Our experimental results were obtained on an 48-core AMD Opteron 6168, running at 900Mhz with 256 GB of RAM and 512KB of L2 cache. Table 4.1 gives running times for the five multiplication algorithms that we have implemented:

- KS_s stands for Kronecker's substitution combined with Schönhage & Strassen algorithm [58]; (see Section 3.2) note this is a serial implementation, run on 1 core,
- CVL_p^2 is our algorithm in Section 4.2, run on 48 cores,
- DnC_p is a straightforward parallel implementation of plain multiplication, run on 48 cores, see Section 3.3,
- Toom_p^4 is a parallel implementation of 4-way Toom-Cook, run on 48 cores, see Section 3.5.2,
- Toom_p^8 is a parallel implementation of 8-way Toom-Cook, run on 48 cores, see Section 3.5.3.

In addition, Table 4.1 gives running times for integer polynomial multiplication performed with FILNT 2.4.3 [37] and Maple 18. which both rely on the same algorithm as our KS_s. In each example of Table 4.1, for each input polynomial, the degree d is equal to the coefficient bit size N. The input polynomials $a(y), b(y)$ are random and dense.

Table 4.2 gathers running times for CVL_p^2, FILNT 2.4.3 and Maple 18 on examples for which the coefficient bit size N is larger than the degree d by a factor around 45.

From these tables, we see that our method CVL_p^2 outperforms its counterparts, both the serial ones and parallel ones on sufficiently large input data.

Note that the implementation of the algorithms described in Chapter 3 relies entirely on the GMP library. This latter is recognized to be about twice slower than its FLINT counterpart.

d, N	CVL_p^2	DnC_p	Toom_p^4	Toom_p^8	KS_s	FLINT_s	Maple_s^{18}
2^{10}	0.139	0.11	0.046	0.059	0.057	0.016	0.06
2^{11}	0.196	0.17	0.17	0.17	0.25	0.067	0.201
2^{12}	0.295	0.58	0.67	0.64	1.37	0.42	0.86
2^{13}	0.699	2.20	2.79	2.73	5.40	1.671	3.775
2^{14}	1.927	8.26	10.29	8.74	20.95	7.178	17.496
2^{15}	9.138	30.75	35.79	33.40	92.03	32.112	84.913
2^{16}	33.04	122.1	129.4	115.9	*Err.	154.69	445.67

Table 4.1: Polynomial multiplication timings with $d = N$.

[3] `http://gcc.gnu.org/svn/gcc/branches/cilkplus/`

d	N	CVL_p^2	FLINT_s	Maple_s^{18}
1024	49152	0.293	1.224	2.65
2048	96256	1.089	4.744	11.82
4096	188416	3.398	23.967	51.414
8192	368640	13.01	102.95	241.072
16384	720896	51.383	496.785	1318.125

Table 4.2: Polynomial multiplication timings with $d \ll N$.

	CVL_s^2 sub-steps			CVL_p^2 sub-steps		
d, N	I	II	III	I	II	III
2048	0.006	0.544	0.055	0.036	0.121	0.045
4096	0.02	2.203	0.22	0.057	0.145	0.086
8192	0.075	9.458	0.832	0.056	0.315	0.263
16384	0.299	40.45	3.274	0.128	1.253	0.647
32768	1.209	174.426	12.852	0.439	4.802	4.102
65536	4.842	777.463	51.404	1.796	21.035	11.211

Table 4.3: Profiling information for CVL_s^2 and CVL_p^2.

Size	$\frac{\text{Work}(\text{CVL}_p^2)}{\text{Span}(\text{CVL}_p^2)}$	$\frac{\text{Work}(\text{CVL}_p^2)}{\text{Work}(\text{KS}_s)}$
4096	402.7	1.76
8192	607.36	1.88
16384	756.03	2.09
32768	865.78	2.06

Table 4.4: `Cilkview` analysis of CVL_p^2 and KS_s.

Table 4.3 provides profiling information for CVL_p^2 and CVL_s^2 (its serial counterpart): Stages I, II, III refers respectively to Lines 1-4, Lines 5-8, Lines 9-11 in the pseudo-code in Section 4.2.2. This shows that the conversions between univariate and multivariate representations contribute to a minor portion of the total running times. This is important since data conversions cannot provide large speedup factors on multicore architectures. Moreover, the fact that Stage II dominates implies that improving our implementation further reduces to use better code for modular FFTs, which is work in progress jointly with the SPIRAL project [49].

Table 4.4 shows that the work overhead (measured by `Cilkview`, the performance analysis tool of `CilkPlus`) of CVL_p^2 w.r.t. to a method based on Schönhage & Strassen algorithm is only around 2 (Column 3) whereas CVL_p^2 provides large amount of parallelism (Column 2).

Turning to parallel univariate real root isolation, we have integrated our parallel integer polynomial multiplication into the algorithm proposed in [11]. To this end, we perform the Taylor Shift operation, that is, the map $x \longmapsto f(x+1)$, by means of Algorithm (E) in [65], which reduces calculations to integer polynomial multiplication in large degrees and the algorithm of [11] in small degrees. In Tables 4.5, we call URRI this adaptive algorithm combining FFT-based arithmetic (via Algorithm (E)) and plain arithmetic (via [11]).

We run these two parallel real root algorithms, URRI and CMY [11], which are both implemented in `CilkPlus`, against Maple 18 serial *realroot* command, which implements a state-of-the-art algorithm. Table 4.5 shows the running times (in secs) of well-known three test problems, including `Bnd`, `Cnd` and `Chebycheff`. Moreover, for each test problem, the degree of the input polynomial varies in a range.

	Size	URRI	CMY [11]	*realroot*	#Roots
Bnd	16384	43.498	127.412	159.245	1
	32768	176.351	609.513	1011.872	1
Cnd	16384	5.086	25.296	109.420	1
	32768	18.141	125.902	816.134	1
Chebycheff	2048	608.738	594.82	1378.444	2047
	4096	8194.06	10014	¿ 18000	4095

Table 4.5: Running time for Bnd, Cnd and Chebycheff

Comparing these results to the correspondence in Table 3.5 for the classical algorithms, shines the worthiness of the effort for this new method. The reason that in our benchmarks in Table 4.1, the new approach is not superior compared to Toom-Cook's is that we are executing the algorithms on a machine having a small number of proces-

sors (say 12-24), and Toom-Cooks which has static parallelism (say 15) performs well. Whereas, when running the new algorithm on an ideal machine, the new algorithm beats Toom-Cook with a significantly high speed-up.

4.6 Conclusion

We have presented a parallel FFT-based method for multiplying dense univariate polynomials with integer coefficients. Our approach relies on two convolutions (cyclic and negacyclic) of bivariate polynomials which allow us to take advantage of the row-column algorithm for 2D FFTs. The data conversions between univariate polynomials over $\mathbb{Z}$ and bivariate polynomials over $\mathbb{Z}/p\mathbb{Z}$ are highly optimized by means of low-level "bit hacks" thus avoiding software multiplication of large integers. In fact, our code relies only and directly on machine word operations (addition, shift and multiplication).

Our experimental results show this new algorithm has a high parallelism and scale better than its competitor algorithms.

The source of the algorithms discussed in this chapter are freely available at the web site of *Basic Polynomial Algebra Subprograms* (BPAS-Library) [4].

[4] `http://bpaslib.org/`

Chapter 5

Subproduct tree techniques on many-core GPUs

We propose parallel algorithms for operations on univariate polynomials (multi-point evaluation, interpolation) based on subproduct tree techniques. We target implementation on many-core GPUs. On those architectures, we demonstrate the importance of adaptive algorithms, in particular the combination of parallel plain arithmetic and parallel FFT-based arithmetic. Experimental results illustrate the benefits of our algorithms.

This chapter is a joint work with S. A. Haque and M. Moreno Maza.

5.1 Introduction

We investigate the use of Graphics Processing Units (GPUs) in the problems of evaluating and interpolating polynomials. Many-core GPU architectures were considered in [61] and [64] in the case of numerical computations, with the purpose of obtaining better support, in terms of accuracy and running times, for the development of polynomial system solvers.

Our motivation, in this work, is also to improve the performance of polynomial system solvers. However, we are targeting symbolic, thus exact, computations. In particular, we aim at providing GPU support for solvers of polynomial systems with coefficients in finite fields, such as the one reported in [54]. This case handles as well problems from cryptography and serves as a base case for the so-called modular methods [16], since those methods reduce computations with rational number coefficients to computations with finite field coefficients.

Finite fields allow the use of asymptotically fast algorithms for polynomial arithmetic, based on Fast Fourier Transforms (FFTs) or, more generally, subproduct tree techniques.

Chapter 10 in the landmark book [28] is an overview of those techniques, which have the advantage of providing a more general setting than FFTs. More precisely, evaluation points do not need to be successive powers of a primitive root of unity. Evaluation and interpolation based on subproduct tree techniques have "essentially" (i.e. up to log factors) the same algebraic complexity estimates as their FFT-based counterparts. However, their implementation is known to be challenging.

In this chapter, we report on the first GPU implementation (using CUDA [55]) of subproduct tree techniques for multi-point evaluation and interpolation of univariate polynomials. The parallelization of those techniques raises the following challenges on hardware accelerators:

1. The divide-and-conquer formulation of operations on subproduct-trees is not sufficient to provide enough parallelism and one must also parallelize the underlying polynomial arithmetic operations, in particular polynomial multiplication.

2. Algorithms based on FFT (such as subproduct tree techniques) are memory bound since the ratio of work to memory access is essentially constant, which makes those algorithms not well suited for multi-core architectures.

3. During the course of the execution of a subproduct tree operation (construction, evaluation, interpolation) the degrees of the involved polynomials vary greatly, thus so does the work load of the tasks, which makes those algorithms complex to implement on many-core GPUs.

The contributions of this work are summarized below. We propose parallel algorithms for performing subproduct tree construction, evaluation and interpolation. We also report on their implementation on many-core GPUs. See Sections 5.3, 5.5 and 5.6, respectively. We enhance the traditional algorithms for polynomial evaluation and interpolation based on subproduct tree techniques, by introducing the data-structure of a *subinverse tree*, which we use to implement both evaluation and interpolation, see Section 5.4. For subproduct tree operations targeting many-core GPUs, we demonstrate the importance of *adaptive algorithms*. That is, algorithms that adapt their behavior according to the available computing resources. In particular, we combine *parallel plain arithmetic* and *parallel fast arithmetic*. For the former we rely on [36] and, for the latter we extend the work of [53]. The span and parallelism overhead of our algorithm are measured considering *many-core machine model* stated in Section 2.4. To evaluate our implementation, we measure the effective memory bandwidth of our GPU code for parallel multi-point evaluation and interpolation on a card with a theoretical maximum memory bandwidth

of 148 GB/S, our code reaches peaks at 64 GB/S. Since the arithmetic intensity of our algorithms is high, we believe that this is a promising result.

All implementation of subproduct tree techniques that we are aware of are serial only. This includes [9] for $GF(2)[x]$, the FLINT library[38] and the `Modpn` library [44]. Hence we compare our code against probably the best serial C code (namely the FLINT library) for the same operations. For sufficiently large input data and on NVIDIA Tesla C2050, our code outperforms its serial counterpart by a factor ranging between 20 to 30. Experimental data are provided in Section 5.7. Our code is available in source as part of the project *CUDA Modular Polynomial* (CUMODP) whose web site is `http://www.cumodp.org`.

5.2 Background

We review various notions related to subproduct tree techniques (see [28, Chapter 10] for details). We also specify costs for the underlying polynomial arithmetic used in our implementation. Notations and hypotheses introduced in this section are used throughout this chapter. Let $n = 2^k$ for some positive integer k and let $\mathbb{K}$ be a finite field. Let $u_0, \ldots, u_{n-1} \in \mathbb{K}$. Define $m_i = x - u_i$, for $0 \leq i < n$. We assume that each $u_i \in \mathbb{K}$ can be stored in one machine word.

Subproduct tree. The subproduct tree $M_n := \mathsf{SubproductTree}(u_0, \ldots, u_{n-1})$ is a complete binary tree of height $k = \log_2 n$. The j-th node of the i-th level of M_n is denoted by $M_{i,j}$, where $0 \leq i \leq k$ and $0 \leq j < 2^{k-i}$, and is defined as follows:

$$M_{i,j} = m_{j \cdot 2^i} \cdot m_{j \cdot 2^i + 1} \cdots m_{j \cdot 2^i + (2^i - 1)} = \prod_{0 \leq \ell < 2^i} m_{j \cdot 2^i + \ell}.$$

Note that each of $M_{i,j}$ can be defined recursively as follows:

$$M_{0,j} = m_j \quad \text{and} \quad M_{i+1,j} = M_{i,2j} \cdot M_{i,2j+1}.$$

Observe that the i-th level of M_n has 2^{k-i} polynomials with degree of 2^i. Since each element of $\mathbb{K}$ fits within a machine word, then storing the subproduct tree M_n requires at most $n \log_2 n + 3n - 1$ words.

Let us split the point set $U = \{u_0, \ldots, u_{n-1}\}$ into two halves of equal cardinality and proceed recursively with each half until it becomes a singleton. This leads to a binary tree of depth $\log_2 n$ having the points $u_0, \ldots, u_{n-1}$ as leaves, depicted on Figure 5.1. Note that the j-th node from the left at level i is labeled by $M_{i,j}$. Algorithm 9 generates the

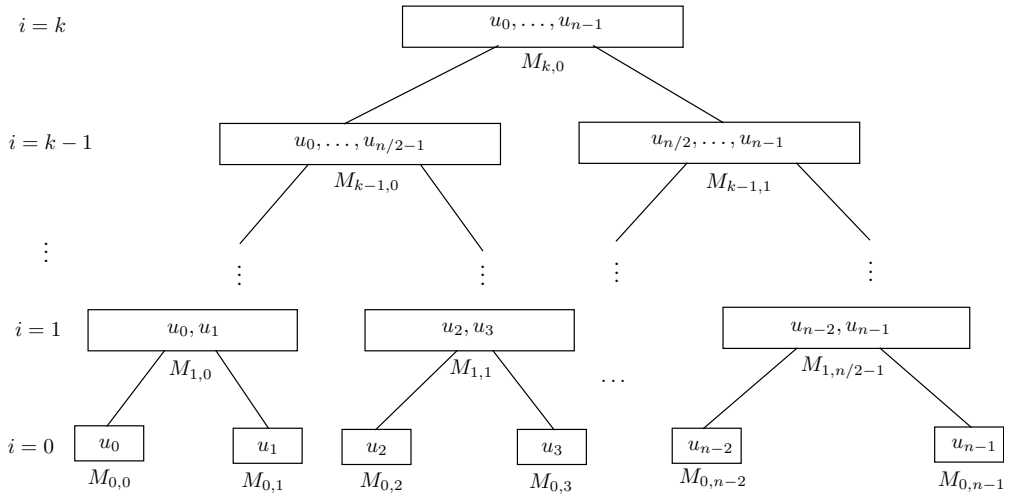


Figure 5.1: Subproduct tree associated with the point set $U = \{u_0, \ldots, u_{n-1}\}$.

polynomials $M_{i,j}$ in an efficient manner, discussed in Section 5.3.

Algorithm 9: SubproductTree$(m_0, \ldots, m_{n-1})$

Input: $m_0 = (x - u_0), \ldots, m_{n-1} = (x - u_{n-1}) \in \mathbb{K}[x]$ with $u_0, \ldots, u_{n-1} \in \mathbb{K}$ and $n = 2^k$ for $k \in \mathbb{N}$.

Output: The subproduct-tree M_n, that is, the polynomials $M_{i,j} = \prod_{0 \le \ell < 2^i} m_{j \cdot 2^i + \ell}$ for $0 \le i \le k$ and $0 \le j < 2^{k-i}$.

```
for j = 0 to n − 1 do
    M_{0,j} = m_j;
for i = 1 to k do
    for j = 0 to 2^{k−i} − 1 do
        M_{i,j} = M_{i−1,2j} M_{i−1,2j+1};
return M_n;
```

Multi-point evaluation and interpolation. Given a univariate polynomial $f \in \mathbb{K}[x]$ of degree less than n, we define $\chi(f) = (f(u_0), \ldots, f(u_{n-1}))$. The map χ is called the *multi-point evaluation map* at $u_0, \ldots, u_{n-1}$. Let $m = \prod_{0 \le i < n}(x - u_i)$. When $u_0, \ldots, u_{n-1}$ are pairwise distinct, then

$$\chi : \begin{array}{ccc} \mathbb{K}[x]/\langle m \rangle & \longrightarrow & \mathbb{K}^n \\ f & \longmapsto & (f(u_0), \ldots, f(u_{n-1})) \end{array}$$

realizes an isomorphism of $\mathbb{K}$-vector spaces $\mathbb{K}[x]/\langle m \rangle$ and $\mathbb{K}^n$. The inverse map χ^{-1} can be computed via Lagrange interpolation. Given values $v_0, \ldots, v_{n-1} \in \mathbb{K}$, the unique polynomial $f \in \mathbb{K}[x]$ of degree less than n which takes the value v_i at the point u_i for all

$0 \leq i < n$ is: $f = \sum_{i=0}^{n-1} v_i s_i m/(x - u_i)$ where $s_i = \prod_{i \neq j, 0 \leq j < n} 1/(u_i - u_j)$. We observe that $\mathbb{K}[x]/\langle m \rangle$ and $\mathbb{K}^n$ are vector spaces of dimension n over $\mathbb{K}$. Moreover, χ is a $\mathbb{K}$-linear map, which is a bijection when the evaluation points $u_0, \ldots, u_{n-1}$ are pairwise distinct.

Complexity measures. Since we are targeting GPU implementation, our parallel algorithms are analyzed using an appropriate model of computation introduced in Section 2.4. The complexity measures are the *work* (i.e. algebraic complexity estimate) the *span* (i.e. running time on infinitely many processors) and the *parallelism overhead*. This latter is the total amount of time for transferring data between the global memory and the local memories of the streaming multi-processors (SMs).

Notation 3 *The number of arithmetic operations for multiplying two polynomials with degree less than d using the* plain (schoolbook) multiplication *is* $\mathsf{M}_{\text{plain}}(d) = 2d^2 - 2d + 1$. *In our GPU implementation, when d is small enough, each polynomial product is computed by a single thread block and thus within the local memory of a single SM. In this case, we use $2d + 2$ threads for one polynomial multiplication. Each thread copies one coefficient from global memory to the local memory. Each of these threads, except one, is responsible for computing one coefficient of the output polynomial and writes that coefficient back to global memory. So the span and parallelism overhead are $d + 1$ and $2U$ respectively, where $1/U$ is the throughput measured in word per second, see Section 2.4.*

Notation 4 *The number of operations for multiplying two polynomials with degree less than d using* Cooley-Tukey's FFT *algorithms is* $\mathsf{M}_{\text{FFT}}(d) = 9/2\ d' \log_2(d') + 4d'$ *[48]. Here* $d' = 2^{\lceil \log_2(2d-1) \rceil}$. *In our GPU implementation, which relies on Stockham FFT algorithm, this number of operations becomes:* $\mathsf{M}_{\text{FFT}}(d) = 15d' \log_2(d') + 2d'$, *see [53]. The span and parallelism overhead of our FFT-based multiplication are* $15d' + 2d'$ *and* $(36d' - 21)U$ *respectively* Section 2.4.

Notation 5 *Given $a, b \in \mathbb{K}[x]$, with* $\deg(a) \geq \deg(b)$ *we denote by* $\mathsf{Remainder}(a, b)$ *the remainder in the* Euclidean division *of a by b. The number of arithmetic operations for computing* $\mathsf{Remainder}(a, b)$, *by plain division, is* $(\deg(b) + 1)(\deg(a) - \deg(b) + 1)$. *In our GPU implementation, we perform plain division for small degree polynomials, in which case a, b are stored into the local memory of an SM. For larger polynomials, we use an FFT-based algorithm to be discussed later. Returning to plain division, we use* $\deg(b) + 1$ *threads to implement this operation. Each thread reads one coefficient of b and at most* $\lceil \frac{\deg(a)+1}{\deg(b)+1} \rceil$ *coefficients of a from the global memory. For the output, at most* $\deg(b)$ *threads write the coefficients of the remainder to the global memory. The span and parallelism overhead are* $2(\deg(a) - \deg(b) + 1)$ *and* $(2 + \lceil \frac{\deg(a)+1}{\deg(b)+1} \rceil)U$.

Notation 6 *For $f \in \mathbb{K}[x]$ of degree $d > 0$ and an $k \geq d$, the* reversal *of order k of f is the polynomial denoted by $\mathsf{rev}_k(f)$ and defined as $\mathsf{rev}_k(f) = x^k f(1/x)$. In our implementation, we use one thread for each coefficient of the input and output. So the span and overhead are 1 and $2U$, respectively.*

Notation 7 *Adding two polynomials of degree at most d is done within $d+1$ coefficient operations. In our implementation, we use one thread per coefficient operation. So the span and overhead are 1 and $3U$, respectively.*

Notation 8 *For $f \in \mathbb{K}[x]$, with $f(0) = 1$, and $\ell \in \mathbb{N}$ the* modular inverse *of f modulo x^ℓ is denoted by $\mathsf{Inverse}(f, \ell)$ and is uniquely defined by $\mathsf{Inverse}(f, \ell)\, f \equiv 1 \mod (x^\ell)$. Algorithm 10 computes $\mathsf{Inverse}(f, \ell)$ using* Newton iteration*. Observe that, this algorithm has $\lceil \log_2 \ell \rceil$ dependent sequential steps. As a result, the for-loop cannot be turned into a parallel loop.*

Algorithm 10: $\mathsf{Inverse}(f, \ell)$

Input: $f \in \mathbb{R}[x]$ such that $f(0) = 1$ and $\ell \in \mathbb{N}$.
Output: $g_r \in \mathbb{R}[x]$ such that $f g_r \equiv 1 \mod x^\ell$.

```
g_0 = 1;
r = ⌈log_2 ℓ⌉;
for i = 1...r do
    g_i = (2 g_{i-1} − f g_{i-1}^2) mod x^{2^i};
return g_r;
```

Remark 12 *To help the reader following the complexity analysis presented in the sequel of this paper, a* Maple *worksheet is available at* `http://cumodp.org/links.html`. *Therein, we compute estimates for space allocation, work (total of number of arithmetic operations), span (parallel running time) and parallelism overhead for constructing sub-product tree and sub-inverse tree (our proposed data structure). Recall that parallelism overhead measures the time spent in transferring data between the global memory of the devices and shared memories of the SMs. Note that the estimates computed by this* Maple *worksheet are based on our CUDA implementation available at* `http://cumodp.org`.

5.3 Subproduct tree construction

In this section, we study an adaptive algorithm for constructing the subproduct tree $M_n := \mathsf{SubproductTree}(u_0, \ldots, u_{n-1})$ as defined in Section 5.2. Recall that $n = 2^k$ holds for

some positive integer k and that we have $u_0, \ldots, u_{n-1} \in \mathbb{K}$. Both polynomial evaluation and interpolation use the subproduct tree technique which depends highly on polynomial multiplication. This brings several implementation challenges.

First of all, it is well-known that, for univariate polynomials of low degrees, FFT-based multiplication is more expensive than plain multiplication in the sense of the number of arithmetic operations. For this reason, we apply plain multiplication for constructing the nodes of levels $1, \ldots, H$ of the subproduct tree M_n, where $0 < H \leq k$ is a prescribed *threshold*. Then, we use FFT-based multiplication for the nodes of higher level.

A second challenge follows from the following observation. At level i of the subproduct tree, each polynomial has degree 2^i and thus $2^i + 1$ coefficients in a dense representation. This is not a favorable case for FFT-based multiplication. Fortunately, the leading coefficient of any such polynomial in the subproduct tree is 1. So, it is possible to create $M_{i,j}$ from $M_{i-1,2j}$ and $M_{i-1,2j+1}$, even if we do not store the leading coefficients of the latter two polynomials.

As we will see in Section 5.7 our implementation still has room for improvements regarding polynomial multiplication. For instance, we could consider using an "intermediate" algorithm for polynomials with degree in a "middle range". Such an algorithm could be the one of Karatsuba or one of its variants. However, it is known that these algorithms are hard to parallelize [12].

Definition 1 *Let H be a fixed integer with $1 \leq H \leq k$. We call the following procedure an* adaptive algorithm for computing M_n with threshold H*:*

1. *for each level $1 \leq h \leq H$, the nodes are computed using plain multiplication,*

2. *for each level $H+1 \leq h \leq k$, the nodes are computed using FFT-based multiplication.*

This algorithm is adaptive in the sense that it takes into account the amount of available resources, as well as the input data size. Indeed, as specified in Section 5.2, each plain multiplication is performed by a single SM, while each FFT-based multiplication is computed by a kernel call, thus using several SMs. In fact, this kernel computes a number of FFT-based products concurrently.

Before analyzing this adaptive algorithm, we recall that, if the subproduct tree M_n is computed by means of a single multiplication algorithm, with multiplication time[1] $\mathsf{M}(n)$, Lemma 10.4 in [28] states that the total number of operations for constructing M_n is at most $\mathsf{M}(n) \log_2 n$ operations in $\mathbb{K}$. Lemma 8 below prepares to the study of our adaptive algorithm.

[1]This notion is defined in [28, Chapter 8]

Lemma 8 *Let $0 \le h_1 < h_2 \le k$ be integers. Assume that level h_1 of M_n has already been constructed. The total number of operations in $\mathbb{K}$ for constructing levels $h_1+1, \ldots, h_2$ in M_n is at most $\sum_{i=h_1+1}^{h_2} 2^{k-i} M(2^{i+1})$.*

PROOF. Recall that $M(d)$ is an upper bound on the number of operations in $\mathbb{K}$ for multiplying two univariate polynomials of degree less than d. Let $h_1 < i \le h_2$ be an index. To construct the i-th level, we need 2^{k-i} multiplications of degree less than 2^{i+1}. So the total cost to construct for level i is upper bounded $2^{k-i}M(2^{i+1})$. □

We can have an immediate consequence from Lemma 8 by setting $h_1 = 0$ and $h_2 = k$.

Corollary 10 *The number of operations for constructing the M_n is $\sum_{i=1}^{k} 2^{k-i} M(2^{i+1})$.*

Remark 13 *We do not store the leading coefficient of polynomials in M_n of levels $H+1, \ldots, k-1$. So, the length of a polynomial becomes 2^i at level i. The objective of this technique is to reduce the computation time for FFT based multiplication. As the leading coefficient is always 1, we proceed as follows.*

Let $a, b \in \mathbb{K}[x]$ be two monic and univariate polynomials. Let $\deg(a) = \deg(b) = d = 2^e$ for some $e \in \mathbb{N}$. Let $a' = a - x^d$ and $b' = b - x^d$. Then, we have $ab = x^{2d} + a'b' + (a' + b')x^d$.

If we were to compute $a\,b$ directly the cost would be $O(\mathsf{M}_{\mathrm{FFT}}(2d))$. But if compute it from $a'\,b'$ using the above formula, then the cost will be reduced to $O(\mathsf{M}_{\mathrm{FFT}}(d) + d)$. On the RAM model, this technique saves almost half of the computational time. On a many-core machine, though the cost is not significant in theory, it saves $O(d)$ memory space and also saves about half of the work. In fact, this has a significant impact on the computational time, as we could observe experimentally.

Another implementation trick is the so-called *FFT doubling*. At a level $H+2 \le i \le k$, for $0 \le j \le 2^{k-i} - 1$, consider how to compute $M_{i,j}$ from $M_{i-1,2j}$ and $M_{i-1,2j+1}$. Since the values of $M_{i-1,2j}$ and $M_{i-1,2j+1}$ at 2^{i-1} points have already been computed (via FFT), it is sufficient, in order to determine $M_{i,j}$, to evaluate $M_{i-1,2j}$ and $M_{i-1,2j+1}$ at 2^{i-1} additional points. To do this, we write $f \in \{M_{i-1,2j}, M_{i-1,2j+1}\}$ as $f = f_0 + x^{2^{i-2}} f_1$, with $\deg(f_0) < 2^{i-2}$, and evaluate each of f_0, f_1 at those 2^{i-1} additional points. While this trick brings savings in terms of work, it increases memory footprint, in particular parallelism overheads. Integrating this trick in our implementation is work in progress and, in the rest of this paper, the theoretical and experimental results do not rely on it.

Proposition 12 *The number of arithmetic operations of the adaptive algorithm for computing M_n with threshold H is*

$$n\left(\frac{15}{2}\log_2(n)^2 + \frac{19}{2}\log_2(n) + 2^H - \frac{15}{2}H^2 - \frac{17}{2}H - \frac{1}{2^H}\right).$$

PROOF. We compute the number of arithmetic operations for constructing M_n with threshold H from Corollary 10. We rely on the cost of polynomial multiplication given in Notations 3 and 4. Note that, we apply the technique described in Remark 13 for FFT-based multiplication to create the polynomials of level $H+1,\ldots,k$ of M_n. The total number of arithmetic operations for computing levels $0,1,\ldots,H$ of M_n using plain arithmetic is $\frac{n}{2}\left(19\log_2(n)+15\log_2(n)^2-19\,H-15\,H^2\right)$ coefficient operations. For levels $H+1,\ldots,k$ of M_n, the cost is $n(H+2^H-\frac{1}{2^H})$ coefficient operations. We obtain the algebraic complexity estimates for constructing M_n by adding these two quantities. □

Proposition 13 *The number of machine words required for storing M_n, with threshold H is given below*

$$n\left(\log_2(n)-H+5\right)+(-H-2)\left(n+\frac{n}{2^{H+1}}\right)+2nH\left(1+\frac{1}{2^{H+2}}\right)$$

PROOF. Following our adaptive algorithm, we distinguish the nodes at levels $0,\ldots H$ from those at levels $H+1,\ldots,k$. At level $i\in\{0,\ldots H\}$, the number of coefficients of each polynomial of M_n is 2^i+1 and all those coefficients are stored. The total number of coefficients over all polynomials in M_n for level $\{0,\ldots,H\}$, which is $(-H-2)\left(n+\frac{n}{2^{H+1}}\right)+2nH\left(1+\frac{1}{2^{H+2}}\right)+5\,n$.

At level $i\in\{H+1,\ldots,k\}$, we use the implementation technique described in Remark 13, that is, leading coefficients of each polynomial are not stored. So a polynomial at level i requires 2^i words of storage. From the same worksheet, we compute the total number of words required to store polynomials at level $\{H+1\ldots,k\}$, which is $n\left(\log_2(n)-H\right)$. □

Proposition 14 *Span and overhead of Algorithm 9 for constructing M_n with threshold H using our adaptive method are* span_{M_n} *and* $\mathsf{overhead}_{M_n}$ *respectively, where*

$$\mathsf{span}_{M_n}=\frac{15}{2}\left(\log_2(n)+1\right)^2-\frac{7}{2}\log_2(n)+2^{H+1}-\frac{15}{2}\left(H+1\right)^2+\frac{9}{2}H-2$$

and

$$\mathsf{overhead}_{M_n}=\left(\left(18\left(\log_2(n)+1\right)^2-35\log_2(n)-18\left(H+1\right)^2+35\,H\right)+2\,H\right)U.$$

PROOF. Let us fix i with $0\le i<H$. At level i, our implementation uses plain multiplication in order to compute the polynomials at level $i+1$. Following Notation 3, the span and the parallelism overhead of this process are $H-2+2^{H+1}$ and $2\,HU$, respectively. For

level $H \leq i < k$, each thread is participating to one FFT-based multiplication and two coefficient additions (in order to implement the trick of Remark 13) With Notation 4 and 7, we obtain the span and overhead for this step from Maple worksheet as $\frac{15}{2}\left(\log_2(n)+1\right)^2 - \frac{7}{2}\log_2(n) - \frac{15}{2}\left(H+1\right)^2 + \frac{7}{2}H$ and $\left(18\left(\log_2(n)+1\right)^2 - 35\log_2(n) - 18\left(H+1\right)^2 + 35H\right)U$ respectively. □

Propositions 12 and 14 imply that for a fixed a H, the parallelism (ratio work to span) is in $\Theta(n)$ which is very satisfactory. We stress the fact that this result could be achieved because both our plain and FFT-based multiplications are parallelized. Observe also that, for a fixed n, parallelism overhead decreases as H increases: that is, plain multiplication suffers less parallelism overheads than FFT-based multiplication on GPUs.

It is natural to ask how to choose H so as to minimize work and span. Elementary calculations, using our MAPLE worksheet suggest $6 \leq H \leq 7$. However, in degrees 2^6 and 2^7, parallelism overhead is too high for FFT-based multiplication and, experimentally, the best choice appeared to be $H = 8$.

5.4 Subinverse tree construction

For $f \in \mathbb{K}[x]$ of degree less than n, evaluating f on the point set $\{u_0, \ldots, u_{n-1}\}$ is done by Algorithm 11 by calling $\mathsf{TopDownTraverse}(f, k, 0, M_n, F)$. An array F of length n is passed to this procedure such that $F[i]$ receives $f(u_i)$ for $0 \leq i \leq n-1$. The function call $\mathsf{Remainder}(f, M_{i,j})$ relies on plain division whenever $i < H$ holds, where H is the threshold of Section 5.3. Fast division is applied when polynomials are large enough and, actually,

Algorithm 11: $\mathsf{TopDownTraverse}(f, i, j, M_n, F)$

Input: $f \in \mathbb{K}[x]$ with $\deg(f) < 2^i$, i and j are integers such that $0 \leq i \leq k$, $0 \leq j < 2^{k-i}$ and F is an array of length n.

```
if i == 0 then
    F[j] = f;
    return;
f_0 = Remainder(f, M_{i-1,2j});
f_1 = Remainder(f, M_{i-1,2j+1});
TopDownTraverse(f_0, i-1, 2j, M_n, F);
TopDownTraverse(f_1, i-1, 2j+1, M_n, F);
```

can not be stored within the local memory of a streaming multiprocessor.

Fast division requires computing $\mathsf{Inverse}(\mathsf{rev}_{2^i}(M_{i,j}), 2^i)$, for $H \leq i \leq k$ and $0 \leq j < 2^{k-i}$, see Chapter 9 in [28]. However, this latter calculation has, in principle, to be done via Newton iteration. As mentioned in Section 5.2, this latter provides little opportunities

for concurrent execution. To overcome this performance issue, we introduce a strategy that relies on a new data structure called *subinverse tree*. In this section, we first define subinverse trees and describe their implementation. Then, we analyze the complexity of constructing a subinverse tree.

Definition 2 *For the subproduct tree* $M_n := \mathsf{SubproductTree}(u_0, \ldots, u_{n-1})$*, the* subinverse tree *associated with* M_n*, denoted by* InvM_n*, is a complete binary tree of the same format as* M_n*, defined as follows. For* $0 \le i \le k$*, for* $0 \le j < 2^{k-i}$*, the* j*-th node of level* i *in* InvM_n *contains the univariate polynomial* $\mathsf{InvM}_{i,j}$ *of less than degree* 2^i *and defined by*

$$\mathsf{InvM}_{i,j}\,\mathsf{rev}_{2^i}(M_{i,j}) \;\equiv\; 1 \mod x^{2^i}.$$

Remark 14 *We do not store the polynomials of subinverse tree* InvM_n *below level* H*. Indeed, for those levels, we rely on plain division for the function calls* $\mathsf{Remainder}(f, M_{i,j})$ *in Algorithm 11.*

Proposition 15 *Let* InvM_n *be the subinverse tree associated with a subproduct tree* M_n*, with the threshold* $H < k$*. Then, the amount of space required for storing* InvM_n*, is* $(k - H)n$*.*

Proof. From the Definition 2, we realize the length of $\mathsf{InvM}_{i,j}$ is 2^i. As the total number of polynomials at level i in InvM_n is 2^{k-i}, we need 2^k, that is, n machine words to store all polynomials of level i. Here, we are not considering the root of InvM_n to be stored, because in evaluation or interpolation of a univariate polynomial, we do not need this. Plus, as it is mentioned in Remark 14, we do not store for levels below H. □

The following lemma is a simple observation from which we derive Proposition 16 and, thus, the principle of subinverse tree construction.

Lemma 9 *Let* $\mathbb{R}$ *be a commutative ring with identity element. Let* $a, b, c \in \mathbb{R}[x]$ *be univariate polynomials such that* $c = a\,b$ *and* $a(0) = b(0) = 1$ *hold. Let* $d = \deg(c) + 1$*. Then, we have* $c(0) = 1$ *and* $\mathsf{Inverse}(c, d) \mod x^d$ *can be computed from* a *and* b *as follows:*

$$\mathsf{Inverse}(c, d) \;\equiv\; \mathsf{Inverse}(a, d) \cdot \mathsf{Inverse}(b, d) \mod x^d.$$

Proposition 16 *Let* $\mathsf{InvM}_{i,j}$ *be the* j^{th} *polynomial (from left to right) of the subinverse tree at level* i*, where* $0 < i < k$ *and* $0 \le j < 2^{k-i}$*. We have the following:*

$$\mathsf{InvM}_{i,j} \equiv \mathsf{Inverse}(\mathsf{rev}_{2^{i-1}}(M_{i-1,2j}), 2^i) \cdot \mathsf{Inverse}(\mathsf{rev}_{2^{i-1}}(M_{i-1,2j+1}), 2^i) \mod x^{2^i}$$

where $\mathsf{InvM}_{i,j} = \mathsf{Inverse}(\mathsf{rev}_{2^i}(M_{i,j}), 2^i)$ *from Definition 2.*

The key observation is that computing $\mathsf{InvM}_{i,j}$ requires $\mathsf{Inverse}(\mathsf{rev}_{2^{i-1}}(M_{i-1,2j}), 2^i)$ and $\mathsf{Inverse}(\mathsf{rev}_{2^{i-1}}(M_{i-1,2j+1}), 2^i)$. However, at level $i-1$, the nodes $\mathsf{InvM}_{i-1,2j}$ and $\mathsf{InvM}_{i-1,2j+1}$ are $\mathsf{Inverse}(\mathsf{rev}_{2^{i-1}}(M_{i-1,2j}), 2^{i-1})$ and $\mathsf{Inverse}(\mathsf{rev}_{2^{i-1}}(M_{i-1,2j+1}), 2^{i-1})$ respectively. To take advantage of this observation, we call $\mathsf{OneStepNewtonIteration}(\mathsf{rev}_{2^{i-1}}(M_{i-1,2j}), \mathsf{InvM}_{i-1,2j}, i-1)$ and $\mathsf{OneStepNewtonIteration}(\mathsf{rev}_{2^{i-1}}(M_{i-1,2j+1}), \mathsf{InvM}_{i-1,2j+1}, i-1)$, see Algorithm 12, so as to obtain $\mathsf{Inverse}(M_{i-1,2j}, 2^i)$ and $\mathsf{Inverse}(M_{i-1,2j+1}, 2^i)$ respectively. Algorithm 12 performs a single iteration of *Newton iteration*'s algorithm. Finally, we perform one truncated polynomial multiplication, as stated in Proposition 16, to obtain $\mathsf{InvM}_{i,j}$. We apply this technique to compute all the polynomials of level i of the subinverse tree, for $H+1 \leq i \leq k$.

Since we do not store the leading coefficients of the polynomials in the subproduct tree, our implementation relies on a modified version Algorithm 12, namely Algorithm 13.

Algorithm 12: $\mathsf{OneStepNewtonIteration}(f, g, i)$

Input: $f, g \in \mathbb{R}[x]$ such that $f(0) = 1$, where $\deg(g) =\leq 2^i$ and $fg \equiv 1 \mod x^{2^i}$.
Output: $g' \in \mathbb{R}[x]$ such that $fg' \equiv 1 \mod x^{2^{i+1}}$.
$g' = (2g - fg^2) \mod x^{2^{i+1}}$;
return g';

Let $f = \mathsf{rev}_{2^i}(M_{i,j})$ and $g = \mathsf{InvM}_{i,j}$. From Definition 2, we have $fg \equiv 1 \bmod x^{2^i}$. Note that $\deg(fg) \leq 2^{i+1} - 1$ holds. Let $e' = -fg + 1$. Thus e' is a polynomial of degree at most $2^{i+1} - 1$. Moreover, from the definition of a subinverse tree, we know its least significant 2^i coefficients are zeros. Let $e = e'/x^{2^i}$. So $\deg(e) \leq 2^i - 1$. In Algorithm 12, we have $g' \equiv g \mod x^{2^i}$. We can compute g' from eg and g. The advantage of working with e instead of e' is that the degree of e' is twice the degree of e.

In Algorithm 13, we compute e as follows

$$e = -\mathsf{rev}_{2^i}(M_{i,j} \cdot \mathsf{rev}_{2^i-1}(\mathsf{InvM}_{i,j}) - x^{2^{i+1}-1})$$

by means of one convolution and three more polynomial operations. Since we do not store the leading coefficient of $M_{i,j}$, we need to do these three additional operations.

Middle product technique is used in the implementations of Algorithms 10 and 12. This improves the computational time significantly [34]. However, we do not apply middle product technique directly in constructing subinverse tree, since it only works well when the intermediate inverse polynomial g_i has smaller degree than polynomial f (in Algorithm 10).

For a given i, with $H < i \leq k$, and for $0 \leq j < 2^{k-i}$, Algorithm 14 computes the polynomial $\mathsf{InvM}_{i,j}$. Algorithm 14 calls Algorithm 13 twice to increase the accuracy of

Algorithm 13: EfficientOneStep($M'_{i,j}$,InvM$_{i,j}$, i)

Input: $M'_{i,j} = M_{i,j} - x^{2^i}$, InvM$_{i,j}$.
Output: g, such that g rev$_{2^i}(M_{i,j}) \equiv 1 \bmod x^{2^{i+1}}$.
a =rev$_{2^i-1}$(InvM$_{i,j}$);
$b = a - x^{2^i-1}$;
c =convolution$(a, M'_{i,j}, 2^i)$;
d =rev$_{2^i}(c + b)$;
$e = -d$;
$h = e$ InvM$_{i,j} \bmod x^{2^i}$;
$g = hx^{2^i}$+InvM$_{i,j}$;
return g;

Algorithm 14: InvPolyCompute(M_n,InvM, i, j)

Input: M_n and InvM are the subproduct tree and subinverse tree respectively.
Output: c such that c rev$_{2^i}(M_{i,j}) \equiv 1 \bmod x^{2^i}$.
$M'_{i-1,2j} = M_{i-1,2j} - x^{2^{i-1}}$;
$M'_{i-1,2j+1} = M_{i-1,2j+1} - x^{2^{i-1}}$;
a = EfficientOneStep($M'_{i-1,2j}$,InvM$_{i-1,2j}, i-1$) ;
b = EfficientOneStep($M'_{i-1,2j+1}$,InvM$_{i-1,2j+1}, i-1$) ;
$c = ab \bmod x^{2^i}$;
return c;

Algorithm 15: SubinverseTree(M_n, H)

Input: M_n is the subproduct tree and $H \in \mathbb{N}$.
Output: the subinverse tree InvM$_n$
for $j = 0 \ldots 2^{k-H} - 1$ **do**
    InvM$_{H,j}$ = Inverse($M_{H,j}$, deg($M_{H,j}$));
for $i = (H+1) \ldots k$ **do**
    for $j = 0 \ldots 2^{k-i} - 1$ **do**
        InvM$_{i,j}$ =InvPolyCompute(M_n,InvM$_{i,j}$);
return InvM$_n$;

$\mathsf{InvM}_{i-1,2j}$ and $\mathsf{InvM}_{i-1,2j+1}$ to x^{2i}. Then it multiplies those latter polynomials and applies a mod operation. Algorithm 15 is the top level algorithm which creates the subinverse tree InvM_n using a bottom-up approach and calling Algorithm 14 for computing each node $\mathsf{InvM}_{i,j}$ for $H \le i \le k$ and $0 \le j < 2^{k-i}$.

Propositions 17 and 18 imply that for a fixed a H, the parallelism (ratio work to span) is in $\Theta(n)$ which is very satisfactory.

Proposition 17 *For the subproduct tree M_n, with threshold H, the number of arithmetic operations for constructing the subinverse tree InvM_n using Algorithm 15 is:*

$$n\left(10\left(3\log_2(n)^2+\log_2(n)-3\,H^2-7\,H-4\right)+\frac{16\,4^{2^H}}{3\cdot 2^H}+2-\frac{1}{3\cdot 2^H}-\frac{2}{2^{H-2^H}}\right).$$

Proof. At level H, we need to compute 2^{k-H} polynomials. For each polynomials, we need to call Algorithm 10, having the corresponding polynomial of subproduct tree at level H, whose degree is 2^H. So the loop in this algorithm runs H times. We apply plain multiplications for this step using the idea of middle product technique to make the implementation fast. In middle product technique, we require convolution to compute some coefficients of a polynomial multiplications. In plain arithmetic, we can do the same in a direct way. For example, in the i-th iteration of the for-loop in Algorithm 10 for $i = 2, \ldots, H$, we need to compute 2^{i-1} coefficients of g_i. We can treat both f and g_{i-1} as polynomials of degree less than 2^{i-1}. Thus this multiplication cost can be expressed as $\mathsf{M}_{\text{plain}}(2^{i-1})$ We need two polynomial multiplications of this type in each iteration. We also need some polynomial subtraction operations. Observe that computing g_0 and g_1 is trivial in Algorithm 10. Based on our implementation described with Notation 3 to compute the total number of coefficient operations for construction the H-th level of InvM_n, the total number of operation is given below:

$$-\frac{n}{3\cdot 2^H}+2n+\frac{16\,n\,4^{2^H}}{3\cdot 2^H}-\frac{2n}{2^{H-2^H}}.$$

After level H, each polynomial in InvM_n is computed by the equation given in Proposition 16. Note that, when we are constructing the i-th level of subinverse tree, it is assumed that all of the polynomials at level $i-1$ are precomputed. All polynomial multiplications in these levels are FFT-based. From Algorithm 13 and Proposition 16 along with Notation 4, we compute the total number of operations required to compute the polynomials from level $H+1$ to $k-1$ which is given below:

$$10\left(\log_2(n)+3\log_2(n)^2-7\,H-3\,H^2-4\right)n.$$

We sum up these two complexity estimates and we get the result. □

Proposition 18 *For the subproduct tree M_n with threshold H, the span and overhead of constructing the subinverse tree InvM_n by Algorithm 15 are $\mathsf{span}_{\mathsf{InvM}_n}$ and $\mathsf{overhead}_{\mathsf{InvM}_n}$ respectively, where*

$$\mathsf{span}_{\mathsf{InvM}_n} = \frac{75}{2}\log_2(n)^2 - \frac{107}{2}\log_2(n) + 2\cdot 4^H + 4\cdot 2^H - \frac{75}{2}H^2 - \frac{43}{2}H + 14$$

and

$$\mathsf{overhead}_{\mathsf{InvM}_n} = U\left(90\log_2(n)^2 - 255\log_2(n) + 2^{H+1} - 90H^2 + 75H + 166\right).$$

Proof. The construction of InvM_n can be divided into two steps. First, we compute the polynomials at level H using plain arithmetic by Algorithm 10. During this step, we assign one thread to compute one polynomial of InvM_n. So its span is equal to the complexity of Newton iteration algorithm that computes inverse of a polynomial of degree 2^H modular x^{2^H}. One kernel call is enough to compute this. Moreover each thread is responsible to copy one polynomial at level H of the subproduct tree from global memory to local memory. The span and overhead that is computed for this step in our Maple worksheet are $4\cdot 2^H - 2 + 2\cdot 4^H$ and $(2^{H+1}+1)U$ respectively.

Second, we construct level $H+1,\ldots,(k-1)$ of InvM_n. As mentioned before, we do not construct the root of the subinverse tree. For a level above H, each thread participates in three FFT-based multiplications and five other coefficient operations (involving shifting, addition, copying). For each of the operations, except FFT-based multiplication, each thread requires accessing at most three times in global memory. So the span and overhead for this step is computed from Notation 4 and 7 are $\frac{75}{2}\log_2(n)^2 - \frac{107}{2}\log_2(n) - \frac{75}{2}H^2 - \frac{43}{2}H + 16$ and $15\left(6\log_2(n)^2 - 17\log_2(n) - 6H^2 + 5H + 11\right)U$ respectively. □

5.5 Polynomial evaluation

Multi-point evaluation of polynomial $f \in \mathbb{K}[x]$ of degree less than n, for points in $\{u_0,\ldots,u_{n-1}\}$ can be done by *Horner's* rule in $O(n^2)$ time. If we consider parallel architecture to solve this problem, the span becomes $O(n)$ by doing each point-evaluations in parallel trivially. Subproduct tree based multi-point evaluation has better time complexity and span than that.

Algorithm 11 solves the multi-point evaluation problem using subproduct tree technique. To do so, we construct the subproduct tree $M_n := \mathsf{SubproductTree}(u_0,\ldots,u_{n-1})$

with threshold H and the corresponding subinverse tree InvM_n. Then, we run Algorithm 11, which requires polynomial division. We implement both plain and fast division. For the latter, we rely on the subinverse tree, as described in Section 5.4

Proposition 19 *For the subproduct tree M_n with threshold H and its corresponding subinverse tree InvM_n, the number of arithmetic operations of Algorithm 11 is:*

$$30\,n\log_2(n)^2 + 106\,n\log_2(n) + n\,2^{H+1} - 30\,n\,H^2 - 46\,n\,H + 74\,n + 16\,\frac{n}{2^H} - 8.$$

PROOF. Our adaptive algorithm has two steps. First, we need to call Algorithm 16 for computing the remainder for $k' = k, \ldots, (H+1)$. We do not need to compute the *inverses of polynomials* as we have InvM_n. All of the multiplications in this algorithm are FFT-based. We need two multiplications and four other operations (polynomial reversals and subtraction). Following Notations 4, 6 and 8, the number of arithmetic operations complexity of this step will be:

$$106\,n\log_2(n) + 30\,n\log_2(n)^2 - 30\,nH^2 - 46\,Hn + 76\,n + 16\,\frac{n}{2^H} - 8.$$

Second, when $k' = H, \ldots, 1$ in Algorithm 16, we use plain division algorithm described in Notation 5. The total number of operation for this step is $n\,2^{H+1} - 2\,n$. □

Remark 15 *In [52], the algebraic complexity estimate for performing* multi-point evaluation *(which only considers multiplication cost and ignores other coefficient operations) is $7M(n/2)\,\log_2(n) + O(M(n))$. Considering for $M(n)$ a multiplication time like the one based on Cooley-Tukey's algorithm (see Section 5.2) the running time estimate of [52] becomes similar to the estimate of Proposition 19. Since our primary goal is paralllelization, we view this comparison as satisfactory. Furthermore, Propositions 19 and 20 imply that for a fixed a H, the parallelism (ratio work to span) is in $\Theta(n)$ which is satisfactory as well.*

In our proposed method, for constructing one polynomial in subinverse tree, we require two polynomial convolutions and two polynomial multiplications between corresponding polynomials (children of that polynomial) from subproduct tree and subinverse tree and one polynomial multiplication. We require two polynomial multiplications in each call to Algorithm 16, if subinverse tree is given. In addition, we need one multiplication to create one polynomial in subproduct tree. So in total, the algebraic complexity estimates for solving multi-point evaluation *using our proposed method is $3M(n/2)(\log_2(n) - 1) + (2M(n/2) + 4M(n/4) + 4\mathsf{CONV}(n/4)(\log_2(n) - 2))$ considering only the polynomial multiplication and ignoring all other coefficient operations. It should be noted, we start*

computing the subinverse tree from level H. Each leaf of the subinverse tree has a constant polynomial that is 1*. Now if we compare our complexity estimate with that in [52], (converting the convolution time to an appropriate multiplication time), we might not see any significant differences. In practice, our proposed method should performs better on parallel machine. We do not compare these techniques. We keep it for future work.*

Considering our adaptive strategy, we compute the exact number of operations in solving multi-point evaluation *problem by adding the algebraic complexity estimates of constructing M_n along with the corresponding* InvM_n *and Algorithm 11 found in Proposition 12, 17 and 19 respectively.*

Proposition 20 *Given a subproduct tree M_n with threshold H and the corresponding subinverse tree* InvM_n*, span and overhead of Algorithm 11 are* $\mathsf{span}_{\mathsf{eva}}$ *and* $\mathsf{overhead}_{\mathsf{eva}}$ *respectively, where*

$$\mathsf{span}_{\mathsf{eva}} = 15\log_2(n)^2 + 23\log_2(n) + 6\times 2^H - 15\,H^2 - 22\,H - 2$$

and

$$\mathsf{overhead}_{\mathsf{eva}} = \left(36\log_2(n)^2 + 3\log_2(n) - 36\,H^2 + 2\,H\right)U.$$

Proof. From the proof of Proposition 19, we can compute the span and overhead of Algorithm 11, when the value of $k' = k, \ldots, (H+1)$, using Notations 4, 6 and 8 as

$$15\log_2(n)^2 + 23\log_2(n) - 15\,H^2 - 23\,H$$

and

$$3\,U\left(12\log_2(n)^2 + \log_2(n) - 12\,H^2 - H\right)$$

respectively. Once the Algorithm 11 depends on plain arithmetic for division that means when $k' = H, \ldots, 1$, the span and overhead can be computed using Notation 5 as $H - 2 + 6\,2^H$ and $5\,HU$ respectively. □

5.6 Polynomial interpolation

As recalled in Section 5.2, we rely on Lagrange interpolation. Our interpolation procedure, inspired by the recursive algorithm in [28, Chapter 10.9], relies on Algorithm 17 below, which proceeds in a bottom-up traversal fashion.

Algorithm 17 computes a binary tree such that the j-th node from the left at level i is a polynomial $I_{i,j}$ of degree $2^i - 1$, for $0 \le i \le k$, $0 \le j \le 2^{k-i} - 1$. The root $I_{k,0}$ is the desired

Algorithm 16: FastRemainder(a, b)

Input: $a, b \in \mathbb{R}[x]$ with $b \neq 0$ monic.
Output: (q, r) such that $a = bq + r$ and $\deg(r) < \deg(b)$

```
t = deg(a);
s = deg(b);
if t < s then
    q = 0;
r = a;
else
    f = rev_s(b);
    g = inverse(f, t − s + 1);
    q = rev_t(a)g  mod  x^{t−s+1};
    /* rev_t(a) means to replace x by 1/x in a and then multiply a with
       x^t.                                                              */
    q = rev_{t−s}(q);
    r = a − bq;
return (q, r);
```

Algorithm 17: LinearCombination$(M_n, c_0, \ldots, c_{n-1})$

Input: Precomputed subproduct tree M_n for the evaluation points $u_0, \ldots, u_{n-1}$, and $c_0, \ldots, c_{n-1} \in \mathbb{K}$, with $n = 2^k$ for $k \in \mathbb{N}$
Output: $\sum_{0 \leq i < n} c_i m/(x - u_i) \in \mathbb{K}[x]$, where $m = \prod_{0 \leq i < n}(x - u_i)$

```
for j = 0 to n − 1 do
    I_{0,j} = c_j;
for i = 1 to k do
    for j = 0 to 2^{k−i} − 1 do
        I_{i,j} = M_{i−1,2j} I_{i−1,2j+1} + M_{i−1,2j+1} I_{i−1,2j};
return I_{k,0};
```

polynomial. We use the same threshold H as for the construction of the subproducttree tree:

1. for each node $I_{i,j}$ where $1 \le i \le H$ and $0 \le j < 2^{k-i}$, we compute $I_{h,j}$ using plain multiplication.

2. for each node $I_{i,j}$, with $H+1 \le i \le k$, we compute the $I_{i,j}$ using FFT-based multiplication.

In Theorem 10.10 in [28], the complexity estimates of the *Linear Combination* is stated as $(\mathsf{M}(n)+O(n))\log(n)$. In Proposition 21, we present a more precise estimate.

Proposition 21 *For the subproduct tree M_n with threshold H, the number of arithmetic operations Algorithm 17 is given below*

$$15\,n\log_2(n)^2 + 20\,n\log_2(n) + 11\,n + 13\,n\,H - 15\,n\,H^2 + n\,2^{H+1} - n\,2^{1-H}.$$

Proof. Each polynomial $I_{i,j}$ for $0 \le i < k$ and $0 \le j < 2^{k-i}$ is obtained by two polynomial multiplications and one polynomial addition. For level $i = 0, \ldots, H$, by plain multiplication and addition, Using Notation 3 and 7, we compute the total number of operations as $3\,Hn + 6\,n + n\,2^{H+1} - n\,2^{1-H}$. For the other levels, we apply FFT-based multiplication. Using Notation 4 and 7, we obtain the total number of operations as $5\,n\left(4\log_2(n) + 1 + 3\log_2(n)^2 + 2\,H - 3\,H^2\right)$. □

Proposition 22 *For the subproduct tree M_n with threshold H and the corresponding subinverse tree InvM_n, the span and overhead of Algorithm 17 are $\mathsf{span}_{\mathsf{lc}}$ and $\mathsf{overhead}_{\mathsf{lc}}$ respectively, where*

$$\mathsf{span}_{\mathsf{lc}} = \frac{15}{2}\log_2(n)^2 + \frac{25}{2}\log_2(n) + 2^{H+1} - \frac{15}{2}H^2 - \frac{21}{2}H - 2$$

and

$$\mathsf{overhead}_{\mathsf{lc}} = 18\log_2(n)^2 + \log_2(n) - 18\,H^2 + 4\,H.$$

Proof. At level i for $0 \le i \le H$, this algorithm does 2^{k-i} polynomial plain multiplications and 2^{k-i-1} polynomial additions. So each thread participates in one coefficient multiplication and one addition. Thus using Notation 3 and 7, we compute the span and overhead as $2\,H - 2 + 2^{H+1}$ and $5\,HU$ respectively.

For a level,i $(i > H)$ we have same number of polynomial multiplications. But each of the multiplication is done by FFT. As we do not store the leading coefficients for both

$M_{i,j}$, we need one more polynomial addition. So a thread participates in one FFT-based multiplication and two coefficient additions. We compute the span and overhead for this step as

$$\frac{15}{2}\log_2(n)^2 + \frac{25}{2}\log_2(n) - \frac{15}{2}H^2 - \frac{25}{2}H$$

and

$$U\left(18\log_2(n)^2 + \log_2(n) - 18H^2 - H\right)$$

respectively by using Notation 4 and 7. □

Finally we use Algorithm 18 in which we first compute $c_0, \ldots, c_{n-1}$, and then we call Algorithm 17. Algorithm 18 is adapted from Algorithm 10.11 in [28].

Algorithm 18: FastInterpolation$(u_0, \ldots, u_{n-1}, v_0, \ldots, v_{n-1})$

Input: $u_0, \ldots, u_{n-1} \in \mathbb{K}$ such that $u_i - u_j$ is a unit for $i \neq j$, and $v_0, \ldots, v_{n-1} \in \mathbb{K}$, and $n = 2^k$ for $k \in \mathbb{N}$

Output: The unique polynomial $P \in \mathbb{K}[x]$ of degree less than n such that $P(u_i) = v_i$ for $0 \leq i < n$

$M_n :=$ SubproductTree$(u_0, \ldots, u_{n-1})$;
Let m be the root of M_n;
Compute $m'(x)$ the derivative of m;
InvM$_n :=$ SubinverseTree(M_n, H);
TopDownTraverse$(m'(x), i, j, M_n, F)$;
return LinearCombination$(M_n, v_0/F[0], \ldots, v_{n-1}/F[n-1])$;

The number of arithmetic operations of *polynomial interpolation* by Algorithm 18 is stated in Remark 16.

Remark 16 *In Algorithm 18, we need to compute* multi-point evaluation *followed by Algorithm 17 for linear combination. In between these two major steps, we compute the* derivation *of the root of the subproduct tree, which can be done by n coefficient operations. So the number of arithmetic operations of Algorithm 18 is the summation of that of polynomial evaluation (from Remark 15), Proposition 21 and n for the derivation.*

5.7 Experimentation

In Table 5.1 we compare the running time of our *multi-point evaluation* CUDA code (for polynomials with different degrees) against the running time of our FFT-based polynomial multiplication CUDA code. We see that the ratio between these running time varies in the range $[2.24, 4.02]$ on a GPU card NVIDIA Tesla C2050, while n [7] the same

ratio is estimated to be $\frac{3}{2}$. We believe that this observation as promising for our implementation. One of the major factors of performance in GPU applications is of memory

K	T_{eva}	T_{mul}	$T_{eva}/T_{mul} * k$
10	0.11	0.0049	2.24
11	0.17	0.0051	3.03
12	0.21	0.0060	2.91
13	0.28	0.0061	3.53
14	0.36	0.0069	3.72
15	0.42	0.0070	4.00
16	0.56	0.0087	4.02
17	0.70	0.0111	3.70
18	1.01	0.0163	3.44
19	1.50	0.0256	3.08
20	2.52	0.0438	2.80
21	4.61	0.0862	2.54
22	9.08	0.1654	2.49
23	18.83	0.3416	2.39

Table 5.1: Computation time for random polynomials with different degrees (2^K) and points. T_{eva} and T_{mul} are running times for polynomial evaluation, an polynomial multiplication (FFT-based) respectively. All of the times are in seconds.

k	Evaluation	Interpolation
4	0.0012	0.0013
5	0.0025	0.0026
6	0.0042	0.0045
7	0.0050	0.0060
8	0.0021	0.0029
9	0.0192	0.0318
10	0.0877	0.1228
11	0.2554	0.3403
12	0.5596	0.7054
13	1.2947	1.6182
14	2.5838	3.1445
15	5.2702	6.3464
16	9.6193	11.4143
17	16.4358	18.7800
18	22.6172	26.7590
19	32.3230	38.7674
20	40.4644	49.0012
21	46.7343	57.0978
22	50.8830	62.4516
23	52.9413	64.2464

Table 5.2: Effective memory bandwidth in (GB/S). k is $\log_2$ of the size of the input polynomial ($n = 2^k$).

bandwidth. For our algorithm this factor is presented for various input degrees in the Table 5.2. The maximum memory bandwidth for our GPU is 148 GB/S. Since our code has a high arithmetic intensity, we believe that our experimental are promising, while leaving room for improvement.

In Table 5.3 and Figure 5.3, we compare two implementations of FFT-based polynomial multiplication. The first one is implemented with CUDA [53]. The other one is from the FLINT library [37]. We executed our CUDA codes on a NVIDIA Tesla M2050 GPU and the other code on the same machine with an Intel Xeon X5650 CPU at 2.67GHz. From the experimental data, it is clear that, our CUDA code for FFT-based multiplication outperforms its FLINT counterpart only in degree larger than 2^{13}. This tells us that we need to implement another multiplication algorithm to have better performance in low-to-average degrees. This is work in progress.

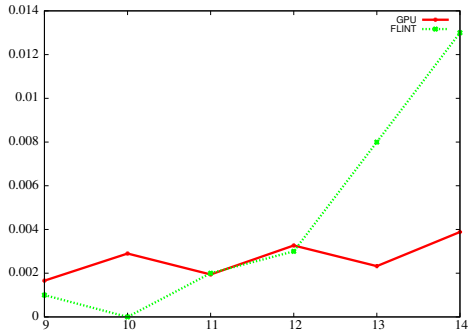


Figure 5.2: Our GPU implementation versus FLINT for FFT-based polynomial multiplication.

k	GPU (s)	FLINT (s)	Speed-Up
9	0.001	0.001	0.602
10	0.0029	0	0
11	0.0019	0.002	1.029
12	0.0032	0.003	0.917
13	0.0023	0.008	3.441
14	0.0039	0.013	3.346
15	0.0032	0.023	7.216
16	0.0065	0.045	6.942
17	0.0084	0.088	10.475
18	0.0122	0.227	18.468
19	0.0198	0.471	23.738
20	0.0266	1.011	27.581
21	0.0718	2.086	29.037
22	0.1451	4.419	30.454
23	0.3043	9.043	29.717

Table 5.3: Execution times of our FFT-based polynomial multiplication of polynomials with the size 2^k comparing with FLINT library.

In Table 5.4 we compare our implementation of polynomial evaluation and interpolation with that of the FLINT library. We found that our implementation does not perform well until degree 2^{15}. In degree 2^{23}, we achieve a 21 times speedup factor w.r.t. FLINT. We believe that by improving our multiplication routine for polynomials of degrees 2^9 to 2^{13}, we would have better performance in both polynomial evaluation and interpolation in these ranges.

5.8 Conclusion

We discussed fast multi-point evaluation and interpolation of univariate polynomials over a finite field on GPU architectures. We have combined algorithmic techniques like subproduct trees, subinverse trees, plain polynomial arithmetic, FFT-based polynomial arithmetic. Up to our knowledge, this is the first report on a parallel implementation of subproduct tree techniques. The source code of our algorithms is freely available in CUMODP-Library website [2].

[2]http://cumodp.org/

	Evaluation			Interpolation		
k	GPU (s)	FLINT (s)	Speed-Up	GPU (s)	FLINT (s)	Speed-Up
10	0.0843	0	0	0.0968	0.01	0.1032
11	0.1012	0.01	0.0987	0.1202	0.01	0.0831
12	0.1361	0.02	0.1468	0.1671	0.03	0.1794
13	0.1580	0.07	0.4429	0.1963	0.09	0.4584
14	0.2034	0.17	0.8354	0.2548	0.22	0.8631
15	0.2415	0.41	1.6971	0.3073	0.53	1.7242
16	0.3126	0.99	3.1666	0.4026	1.26	3.1294
17	0.4285	2.33	5.4375	0.5677	2.94	5.1780
18	0.7106	5.43	7.6404	0.9034	6.81	7.5379
19	1.0936	12.63	11.5484	1.3931	15.85	11.3768
20	1.9412	29.2	15.0420	2.4363	36.61	15.0268
21	3.6927	67.18	18.1923	4.5965	83.98	18.2702
22	7.4855	153.07	20.4486	9.2940	191.32	20.5851
23	15.796	346.44	21.9321	19.6923	432.13	21.9441

Table 5.4: Execution times of our polynomial evaluation and interpolation where the size of polynomial is 2^k compared with those of FLINT library.

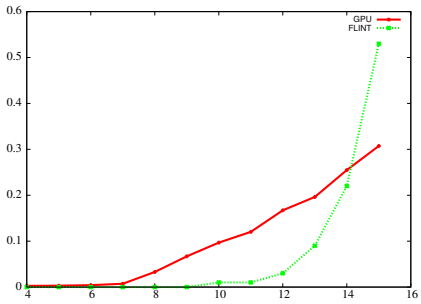


Figure 5.3: Interpolation lower degrees

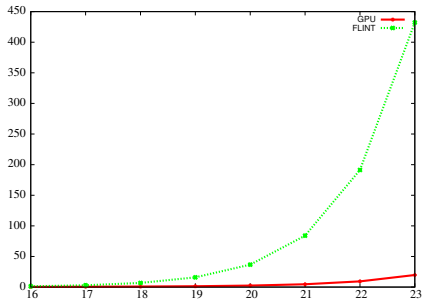


Figure 5.4: Interpolation higher degrees

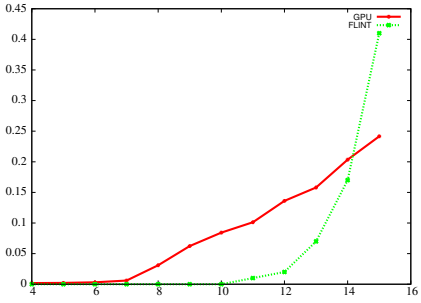


Figure 5.5: Evaluation lower degrees

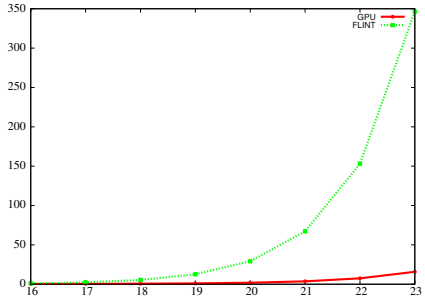


Figure 5.6: Evaluation higher degrees

The experimental results are very promising. Room for improvement, however, still exists, in particular for efficiently multiplying polynomials in the range of degrees from 2^9 to 2^{13}. Filling this gap is work in progress.

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] M. Bodrato and A. Zanoni. What about Toom-Cook matrices optimality ? Technical Report 605, Centro "Vito Volterra", Università di Roma "Tor Vergata", October 2006. `http://bodrato.it/papers/#CIVV2006`.

[5] M. Bodrato and A. Zanoni. Integer and polynomial multiplication: towards optimal toom-cook matrices. In Wang [66], pages 17–24.

[6] W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 24(3-4):235–265, 1997. Computational algebra and number theory (London, 1993).

[7] A. Bostan and É. Schost. Polynomial evaluation and interpolation on special sets of points. *J. Complexity*, 21(4):420–446, 2005.

[8] R. Brent and P. Zimmermann. *Modern Computer Arithmetic*. Cambridge University Press, New York, NY, USA, 2010.

[9] R. P. Brent, P. Gaudry, E. Thomé, and P. Zimmermann. Faster multiplication in gf(2)[x]. In *Proceedings of the 8th international conference on Algorithmic number theory*, ANTS-VIII'08, pages 153–166, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] C. Chen, F. Mansouri, M. Moreno Maza, N. Xie, and Y. Xie. Parallel multiplication of dense polynomials with integer coefficients. 2014.

[11] C. Chen, M. Moreno Maza, and Y. Xie. Cache complexity and multicore implementation for univariate real root isolation. *J. of Physics: Conference Series*, 341, 2011.

[12] M. F. I. Chowdhury, M. Moreno Maza, W. Pan, and É. Schost. Complexity and performance results for non fft-based univariate polynomial multiplication. In *Proceedings of Advances in mathematical and computational methods: addressing modern of science, technology, and society, AIP conference proceedings*, volume 1368, pages 259–262, 2011.

[13] S. A. Cook. *On the minimum computation time of functions*. PhD thesis, 1966. URL: `http://cr.yp.to/bib/entries.html#1966/cook`.

[14] J. Cooley and J. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comp.*, 19:297–301, 1965.

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

[16] X. Dahan, M. Moreno Maza, É. Schost, W. Wu, and Y. Xie. Lifting techniques for triangular decompositions. In *Proceedings of the 2005 international symposium on Symbolic and algebraic computation*, ISSAC '05, pages 108–115, New York, NY, USA, 2005. ACM.

[17] A. De, P. P. Kurur, C. Saha, and R. Saptharishi. Fast integer multiplication using modular arithmetic. *SIAM J. Comput.*, 42(2):685–699, 2013.

[18] R. J. Fateman. Can you save time in multiplying polynomials by encoding them as integers?

[19] 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.

[20] M. Frigo and S. G. Johnson. The design and implementation of fftw3. 93(2):216–231, 2005.

[21] 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.

[22] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms*, 8(1):4, 2012.

[23] 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.

[24] M. Fürer. Faster integer multiplication. *SIAM J. Comput.*, 39(3):979–1005, 2009.

[25] M. Gastineau. Parallel operations of sparse polynomials on multicores: I. multiplication and poisson bracket. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, PASCO '10, pages 44–52, New York, NY, USA, 2010. ACM.

[26] M. Gastineau and J. Laskar. Development of trip: Fast sparse multivariate polynomial multiplication using burst tries. In Vassil N. Alexandrov, G. Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *International Conference on Computational Science (2)*, volume 3992 of *Lecture Notes in Computer Science*, pages 446–453. Springer, 2006.

[27] M. Gastineau and J. Laskar. Highly scalable multiplication for distributed sparse multivariate polynomials on many-core systems. In Vladimir P. Gerdt, Wolfram Koepf, Ernst W. Mayr, and Evgenii V. Vorozhtsov, editors, *CASC*, volume 8136 of *Lecture Notes in Computer Science*, pages 100–115. Springer, 2013.

[28] J. Gathen and J. Gerhard. *Modern Computer Algebra.* Cambridge University Press, 1999.

[29] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 2 edition, 2003.

[30] P. Gaudry, A. Kruppa, and P. Zimmermann. A gmp-based implementation of schönhage-strassen's large integer multiplication algorithm. In Wang [66], pages 167–174.

[31] P. B. Gibbons. A more practical PRAM model. In *Proc. of SPAA*, pages 158–168, 1989.

[32] T. Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 5.0.5 edition, 2012. `http://gmplib.org/`.

[33] A. Granville. Smooth numbers: computational number theory and beyond. *Algorithmic Number Theory, MSRI Publications*, 44:267–323, 2008.

[34] G. Hanrot, M. Quercia, and P. Zimmermann. The middle product algorithm i. *Appl. Algebra Eng., Commun. Comput.*, 14(6):415–438, March 2004.

[35] S. A. Haque, M. Moreno Maza, and N. Xie. A many-core machine model for designing algorithms with minimum parallelism overheads. *CoRR*, abs/1402.0264, 2014.

[36] S. A. Haque and M. Moreno Maza. Plain polynomial arithmetic on GPU. In *J. of Physics: Conf. Series*, volume 385, page 12014. IOP Publishing, 2012.

[37] W. Hart, F. Johansson, and S. Pancratz. FLINT: Fast Library for Number Theory, 2012. Version 2.3, `http://flintlib.org`.

[38] W. B. Hart. Fast library for number theory: An introduction. In K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, editors, *Mathematical Software - ICMS 2010, Third International Congress on Mathematical Software, Kobe, Japan, September 13-17, 2010. Proceedings*, volume 6327 of *Lecture Notes in Computer Science*, pages 88–91. Springer, 2010.

[39] D. Harvey. Faster polynomial multiplication via multipoint kronecker substitution. *J. Symb. Comput.*, 44(10):1502–1510, October 2009.

[40] J. W. Hong and H. T. Kung. I/o complexity: The red-blue pebble game. In *STOC*, pages 326–333. ACM, 1981.

[41] A. A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7:595–596, 1963. URL: `http://cr.yp.to/bib/entries.html#1963/karatsuba`.

[42] D. E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[43] C. E. Leiserson. The cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[44] 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.

[45] C. Van Loan. *Computational frameworks for the fast Fourier transform*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[46] 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.

[47] M. Moreno Maza and Y. Xie. Balanced dense polynomial multiplication on multi-cores. *Int. J. Found. Comput. Sci.*, 22(5):1035–1055, 2011.

[48] M. Moreno Maza and Y. Xie. Balanced dense polynomial multiplication on multi-cores. *Int. J. Found. Comput. Sci.*, 22(5):1035–1055, 2011.

[49] L. Meng, Y. Voronenko, J. R. Johnson, M. Moreno Maza, F. Franchetti, and Y. Xie. Spiral-generated modular fft algorithms. In M. Moreno Maza and Jean-Louis Roch, editors, *PASCO*, pages 169–170. ACM, 2010.

[50] M. Monagan and R. Pearce. Poly: A new polynomial data structure for maple 17 .

[51] M. B. Monagan and R. Pearce. Parallel sparse polynomial multiplication using heaps. In Jeremy R. Johnson, Hyungju Park, and Erich Kaltofen, editors, *ISSAC*, pages 263–270. ACM, 2009.

[52] P. L. Montgomery. *An FFT Extension of the Elliptic Curve Method of Factorization*. PhD thesis, University of California Los Angeles, USA, 1992.

[53] M. Moreno Maza and W. Pan. Fast polynomial arithmetic on a gpu. *J. of Physics: Conference Series*, 256, 2010.

[54] M. Moreno Maza and W. Pan. Solving bivariate polynomial systems on a gpu. *J. of Physics: Conference Series*, 341, 2011.

[55] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[56] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, June 2011.

[57] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[58] A. Schönhage and V. Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7(3-4):281–292, 1971.

[59] 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.

[60] L. J. Stockmeyer and U. Vishkin. Simulation of parallel random access machines by circuits. *SIAM J. Comput.*, 13(2):409–422, 1984.

[61] S. Tanaka, T. Chou, B. Y. Yang, C. M. Cheng, and K. Sakurai. Efficient parallel evaluation of multivariate quadratic polynomials on gpus. In *WISA*, pages 28–42, 2012.

[62] A. L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Mathematics Doklady*, 3:714–716, 1963.

[63] J. van der Hoeven. The truncated fourier transform and applications. In Jaime Gutierrez, editor, *ISSAC*, pages 290–296. ACM, 2004.

[64] J. Verschelde and G. Yoffe. Evaluating polynomials in several variables and their derivatives on a gpu computing processor. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, IPDPSW '12, pages 1397–1405, Washington, DC, USA, 2012. IEEE Computer Society.

[65] J. von zur Gathen and J. Gerhard. Fast algorithms for taylor shifts and certain difference equations. In B. W. Char, P. S. Wang, and W. Küchlin, editors, *ISSAC*, pages 40–47. ACM, 1997.

[66] D. Wang, editor. *Symbolic and Algebraic Computation, International Symposium, ISSAC 2007, Waterloo, Ontario, Canada, July 28 - August 1, 2007, Proceedings*. ACM, 2007.

[67] Y. Wang and X. Zhu. A fast algorithm for the fourier transform over finite fields and its vlsi implementation. *IEEE J.Sel. A. Commun.*, 6(3):572–577, September 2006.

[68] T. Z. Xuan. On smooth integers in short intervals under the Riemann hypothesis. *Acta Arithmetica*, 88:327–332, 1999.

[69] A. Zanoni. Toom-cook 8-way for long integers multiplication. In S. M. Watt, V. Negru, T. Ida, T. Jebelean, D. Petcu, and D. Zaharie, editors, *SYNASC*, pages 54–57. IEEE Computer Society, 2009.

[70] A. Zanoni. Iterative toom-cook methods for very unbalanced long integer multiplication. In *Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*, ISSAC '10, pages 319–323, New York, NY, USA, 2010. ACM.

Appendix A

Converting

A.1 Convert-in

```
/**
 * Convert a univariate polynomial with large coefficients to
 * a bivariate polynomial representation with relatively
 * small coefficients.
 *
 * univariate --> a1    * y ^ 0 + a2    * y ^ 1 + ... + ad    * y ^ (d-1)
 * bivariate  --> A1(x) * y ^ 0 + A2(x) * y ^ 1 + ... + Ad(x) * y ^ (d-1)
 *
 * Ai(x) = b1 * x ^ 0 + b2 * x ^ 1 + ... + bK * x ^ (K-1)
 *
 * @param coeff the coefficients of large integers
 * @param d partial degree of the univariate polynomial plus one
 */
BivariatePolynomial * MulSSA::ToBivarTwoMod(mpz_class *coeff, int d,
                                            int p1, int p2) {

    BivariatePolynomial *biA = new BivariatePolynomial(d, K, M, 2);
    int *A2 = biA->getCoefficients() + biA->getSize();

    //convert a large int coeff[i] to a univar poly of degree K-1
    //by a base of 2^M
    //store the coefficients of the univar poly to the coeff of y^i
    #pragma cilk_grainsize = 8192;
```

```
    cilk_for(int i=0; i<d; ++i){
        if (coeff[i]!=0){
            sfixn ci = i*K;
            sfixn *Ai = biA->getCoefficients() + ci;
            sfixn *A2i = A2 + ci;
            mpzToPolyTwoMod((coeff[i]).get_mpz_t(), M, p1, p2, Ai, A2i);
        }
    }

    return biA;
}
```

A.2 Convert-out

```
/**
 * for both C^+ (ncc) and C^- (cc), for each coefficient of y^i,
 * which is a large integer encoded as a polynomial in x, i.e.
 * ncc1_i(x), ncc2_i(x), ncc3_i(x), cc1_i(x), cc2_i(x), cc3(x)
 * (1) for each coefficient of x, apply CRT for the three prime numbers,
 *     get ncc_i and cc_i
 * (2) convert ncc_i and cc_i to GMP, get u_i and v_i
 * (3) compute c_i = (u_i+v_i)/2+(-u_i+v_i)/2*2^N
 *
 * Output:
 * @c: An array storing big integer coefficients, that is c = a * b
 *
 * Input:
 * @ncc1: An array storing ncc1_i(x), ncc2_i(x), ncc3_i(x), cc1_i(x), cc2_i(x), cc3(x
 * @d1: Degree of polynomial a
 * @d2: Degree of polynomial b
 **/
void MulSSA::CRT_ToGMP_Recovering(sfixn *ncc1, int d1, int d2, mpz_class *c) {
    int csize = d1 + d2 - 1;
    int fullsize = K * csize;
    sfixn *ncc2 = ncc1 + fullsize;
    sfixn *ncc3 = ncc2 + fullsize;
```

```
    sfixn *cc1 = ncc3 + fullsize;
    sfixn *cc2 = cc1 + fullsize;
    sfixn *cc3 = cc2 + fullsize;

    #pragma cilk_grainsize = 1024;
    cilk_for (int i = 0; i < csize; ++i) {
        int Ks = i * K;
        mpz_t u, v;
        CRTtoMPZ(u, ncc1+Ks, ncc2+Ks, ncc3+Ks, K);
        CRTtoMPZ(v, cc1+Ks, cc2+Ks, cc3+Ks, K);

        mpz_add (c[i].get_mpz_t(), u, v);
        c[i] >>= 1;
        mpz_sub (v, v, u);
        c[i] += mpz_class (v) << (N - 1);

        mpz_clear(u);
        mpz_clear(v);
    }
}
```

Appendix B

Good N Table

N	k	M
2	1	1
4	1	2
6	1	3
8	1	4
10	1	5
12	1	6
14	1	7
16	1	8
18	1	9
20	1	10
22	1	11
24	1	12
26	1	13
28	1	14
30	1	15
32	1	16
34	1	17
36	1	18
38	1	19
40	1	20
42	1	21
44	1	22
46	1	23
48	1	24

N	k	M
50	1	25
52	1	26
54	1	27
56	1	28
58	1	29
60	1	30
62	1	31
64	1	32
66	1	33
68	1	34
70	1	35
72	1	36
74	1	37
76	1	38
78	1	39
80	1	40
82	1	41
84	1	42
86	1	43
88	1	44
90	1	45
92	1	46
94	1	47
96	1	48

N	k	M
98	1	49
100	1	50
102	1	51
104	1	52
106	1	53
108	1	54
110	1	55
112	1	56
114	1	57
116	2	29
120	2	30
124	2	31
128	2	32
132	2	33
136	2	34
140	2	35
144	2	36
148	2	37
152	2	38
156	2	39
160	2	40
164	2	41
168	2	42
172	2	43

N	k	M
176	2	44
180	2	45
184	2	46
188	2	47
192	2	48
196	2	49
200	2	50
204	2	51
208	2	52
212	2	53
216	2	54
220	2	55
224	2	56
232	3	29
240	3	30
248	3	31
256	3	32
264	3	33
272	3	34
280	3	35
288	3	36
296	3	37
304	3	38
312	3	39

Table B.1: Good $N, k = \log_2 K, M$ using two 62-bits primes.

N	k	M
320	3	40
328	3	41
336	3	42
344	3	43
352	3	44
360	3	45
368	3	46
376	3	47
384	3	48
392	3	49
400	3	50
408	3	51
416	3	52
424	3	53
432	3	54
440	3	55
448	4	28
464	4	29
480	4	30
496	4	31
512	4	32
528	4	33
544	4	34
560	4	35
576	4	36
592	4	37
608	4	38
624	4	39
640	4	40
656	4	41
672	4	42
688	4	43
704	4	44
720	4	45
736	4	46
752	4	47
768	4	48
784	4	49
800	4	50
816	4	51
832	4	52
848	4	53
864	4	54
896	5	28

N	k	M
928	5	29
960	5	30
992	5	31
1024	5	32
1056	5	33
1088	5	34
1120	5	35
1152	5	36
1184	5	37
1216	5	38
1248	5	39
1280	5	40
1312	5	41
1344	5	42
1376	5	43
1408	5	44
1440	5	45
1472	5	46
1504	5	47
1536	5	48
1568	5	49
1600	5	50
1632	5	51
1664	5	52
1696	5	53
1728	6	27
1792	6	28
1856	6	29
1920	6	30
1984	6	31
2048	6	32
2112	6	33
2176	6	34
2240	6	35
2304	6	36
2368	6	37
2432	6	38
2496	6	39
2560	6	40
2624	6	41
2688	6	42
2752	6	43
2816	6	44
2880	6	45

N	k	M
2944	6	46
3008	6	47
3072	6	48
3136	6	49
3200	6	50
3264	6	51
3328	6	52
3456	7	27
3584	7	28
3712	7	29
3840	7	30
3968	7	31
4096	7	32
4224	7	33
4352	7	34
4480	7	35
4608	7	36
4736	7	37
4864	7	38
4992	7	39
5120	7	40
5248	7	41
5376	7	42
5504	7	43
5632	7	44
5760	7	45
5888	7	46
6016	7	47
6144	7	48
6272	7	49
6400	7	50
6528	7	51
6656	8	26
6912	8	27
7168	8	28
7424	8	29
7680	8	30
7936	8	31
8192	8	32
8448	8	33
8704	8	34
8960	8	35
9216	8	36
9472	8	37

N	k	M
9728	8	38
9984	8	39
10240	8	40
10496	8	41
10752	8	42
11008	8	43
11264	8	44
11520	8	45
11776	8	46
12032	8	47
12288	8	48
12544	8	49
12800	8	50
13312	9	26
13824	9	27
14336	9	28
14848	9	29
15360	9	30
15872	9	31
16384	9	32
16896	9	33
17408	9	34
17920	9	35
18432	9	36
18944	9	37
19456	9	38
19968	9	39
20480	9	40
20992	9	41
21504	9	42
22016	9	43
22528	9	44
23040	9	45
23552	9	46
24064	9	47
24576	9	48
25088	9	49
25600	10	25
26624	10	26
27648	10	27
28672	10	28
29696	10	29
30720	10	30
31744	10	31

Table B.2: Good $N, k = \log_2 K, M$ using two 62-bits primes. (Continue)

N	k	M	N	k	M	N	k	M	N	k	M
32768	10	32	98304	12	24	335872	13	41	1179648	15	36
33792	10	33	102400	12	25	344064	13	42	1212416	15	37
34816	10	34	106496	12	26	352256	13	43	1245184	15	38
35840	10	35	110592	12	27	360448	13	44	1277952	15	39
36864	10	36	114688	12	28	368640	13	45	1310720	15	40
37888	10	37	118784	12	29	376832	14	23	1343488	15	41
38912	10	38	122880	12	30	393216	14	24	1376256	15	42
39936	10	39	126976	12	31	409600	14	25	1409024	15	43
40960	10	40	131072	12	32	425984	14	26	1441792	16	22
41984	10	41	135168	12	33	442368	14	27	1507328	16	23
43008	10	42	139264	12	34	458752	14	28	1572864	16	24
44032	10	43	143360	12	35	475136	14	29	1638400	16	25
45056	10	44	147456	12	36	491520	14	30	1703936	16	26
46080	10	45	151552	12	37	507904	14	31	1769472	16	27
47104	10	46	155648	12	38	524288	14	32	1835008	16	28
48128	10	47	159744	12	39	540672	14	33	1900544	16	29
49152	10	48	163840	12	40	557056	14	34	1966080	16	30
51200	11	25	167936	12	41	573440	14	35	2031616	16	31
53248	11	26	172032	12	42	589824	14	36	2097152	16	32
55296	11	27	176128	12	43	606208	14	37	2162688	16	33
57344	11	28	180224	12	44	622592	14	38	2228224	16	34
59392	11	29	184320	12	45	638976	14	39	2293760	16	35
61440	11	30	188416	12	46	655360	14	40	2359296	16	36
63488	11	31	196608	13	24	671744	14	41	2424832	16	37
65536	11	32	204800	13	25	688128	14	42	2490368	16	38
67584	11	33	212992	13	26	704512	14	43	2555904	16	39
69632	11	34	221184	13	27	720896	14	44	2621440	16	40
71680	11	35	229376	13	28	753664	15	23	2686976	16	41
73728	11	36	237568	13	29	786432	15	24	2752512	16	42
75776	11	37	245760	13	30	819200	15	25	2883584	17	22
77824	11	38	253952	13	31	851968	15	26	3014656	17	23
79872	11	39	262144	13	32	884736	15	27	3145728	17	24
81920	11	40	270336	13	33	917504	15	28	3276800	17	25
83968	11	41	278528	13	34	950272	15	29	3407872	17	26
86016	11	42	286720	13	35	983040	15	30	3538944	17	27
88064	11	43	294912	13	36	1015808	15	31	3670016	17	28
90112	11	44	303104	13	37	1048576	15	32	3801088	17	29
92160	11	45	311296	13	38	1081344	15	33	3932160	17	30
94208	11	46	319488	13	39	1114112	15	34	4063232	17	31
96256	11	47	327680	13	40	1146880	15	35	4194304	17	32

Table B.3: Good $N, k = \log_2 K, M$ using two 62-bits primes. (Continue)

Curriculum Vitae

Name: **Farnam Mansouri**

Education and Degrees:

The University of Western Ontario
London, Ontario, Canada
M.Sc. in Computer Science, April 2014

Amirkabir University of Technology
Tehran, Iran
B.S. in Computer Engineering, May 2012

Work Experience:

Research Assistant, Teaching Assistant
University of Western Ontario, London, Canada
January 2013 - April 2014

Research Intern (Mitacs-Accelerate)
Maplesoft Inc., Waterloo, Ontario, Canada.
July 2013 - October 2013

Software Solution Provider
Caspian Company, Tehran, Iran
September 2010 - September 2012