3-19-2021 1:30 PM

Parallel Arbitrary-precision Integer Arithmetic

Davood Mohajerani, *The University of Western Ontario*

Supervisor: Moreno Maza, Marc, *The University of Western Ontario*

A thesis submitted in partial fulfillment of the requirements for the Doctor of Philosophy degree in Computer Science

Abstract

Arbitrary-precision integer arithmetic computations are driven by applications in solving systems of polynomial equations and public-key cryptography. Such computations arise when high precision is required (with large input values that fit into multiple machine words), or to avoid coefficient overflow due to intermediate expression swell. Meanwhile, the growing demand for faster computation alongside the recent advances in the hardware technology have led to the development of a vast array of many-core and multi-core processors, accelerators, programming models, and language extensions (e.g., CUDA and OpenCL for GPUs, and OpenMP and Cilk for multi-core CPUs). The massive computational power of parallel processors makes them attractive targets for carrying out arbitrary-precision integer arithmetic. At the same time, developing parallel algorithms, followed by implementing and optimizing them as multi-threaded parallel programs imposes a set of challenges. This work explains the current state of research on parallel arbitrary-precision integer arithmetic on GPUs and CPUs, and proposes a number of solutions for some of the challenging problems related to this subject.

Keywords: Arbitrary-precision integer arithmetic, FFT, GPU, Multi-core

Summary

Arbitrary-precision integer arithmetic computations are driven by applications in solving systems of polynomial equations and public-key cryptography. Such computations arise when high precision is required. Meanwhile, the growing demand for faster computation alongside the recent advances in the hardware technology have led to the development of a vast array of many-core and multi-core processors, accelerators, programming models, and language extensions. The massive computational power of parallel processors makes them attractive targets for carrying out arbitrary-precision integer arithmetic. At the same time, developing parallel algorithms, followed by implementing and optimizing them as multi-threaded parallel programs imposes a set of challenges. This work explains the current state of research on parallel arbitrary-precision integer arithmetic on GPUs and CPUs, and proposes a number of solutions for some of the challenging problems related to this subject.

Co-Authorship Statement

- **Chapter 2** is a joint work with Liangyu Chen, Svyatoslav Covanov, and Marc Moreno Maza. This work is published in [1]. The contributions of the thesis author include: a new GPU implementation, optimization of arithmetic operations on GPU, experimental results (tables, diagrams, profiling information) for comparing the presented algorithms against computationally equivalent solutions on CPUs and GPUs.
- **Chapter 3** is a joint work with Svyatoslav Covanov, Marc Moreno Maza, and Linxiao Wang. This work is published in [2]. The contributions of the thesis author include: adaptation of convolution code (originally developed by Svyatoslav Covanov as part of BPAS library) for multiplying arbitrary elements of big prime field, specialized arithmetic for the CRT, implementation and tuning of base-case DFT functions, parallelization of the code using Cilk, and collection of experimental results.
- **Chapter 4** is a joint work with Alexander Brandt, Marc Moreno Maza, Jeeva Paudel, and Linxiao Wang. A preprint of this work is published in [3]. The contributions of the thesis author include: processing of the annotated CUDA kernel code, generation of instrumented binary code using the LLVM Pass Framework, development of a customized profiler using NVIDIA's EVENT API within the CUPTI API, and collection of experimental results.
- **Chapter 5** is a joint work with Marc Moreno Maza. The contributions of the thesis author include: algorithm design and analysis, implementation, code optimization and collection of experimental results.

Acknowlegements

First and foremost, I would like to thank my supervisor Professor Marc Moreno Maza. I am grateful for his advice, support, and providing me the opportunity to work on multiple fascinating problems. Through our discussions, I have had the chance to learn a handful of lessons, including but not limited to, the emphasis on the first principles, clear and concise expression of ideas, and trying to systematically model the problems with algebraic structures. There were multiple times that I assumed a problem could not be further studied, or optimized and more than once I was proven wrong. This in essence has taught me to distinguish the real barriers versus the ones that are a creation of my thinking. Everything that I have worked on has taught me another important lesson that I must always remember and that is what Voltaire once said it best: *"Perfect is the enemy of the good."*

I am grateful for the insightful comments and questions of thesis examiners Professor Michael Bauer, Professor Mark Daley, Professor Arash Reyhani-Masoleh, and Professor Éric Schost.

I would like to thank the members of Ontario Research Center for Computer Algebra (ORCCA) and the Computer Science Department of the University of Western Ontario. I am thankful to my colleagues and co-authors; I have learned a little bit of what to do and what not to do from each of you. Specifically, I am thankful to Svyatoslav Covanov for laying the theoretical foundation and further contributions to our joint work on big prime field FFT. I also would like to thank Alexander Brandt for his help with proofreading chapter 5 of this thesis as well as our ISSAC 2019 paper (FFT on multi-core CPUs) although he was not a co-author of that work.

Last but not least, I am very thankful to my family and friends for their endless support.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background and motivation

Arbitrary-precision integer arithmetic is driven by applications in solving systems of polynomial equations and cryptography. Those arithmetic calculations arise when high precision is required either because of large input values that fit into multiple machine words, or because of possible coefficient overflow due to intermediate expression swell. The main difficulty with the implementation of arbitrary-precision arithmetic is to sharply control hardware resources, which translates in scheduling and parallelization challenges. Meanwhile, the growing demand for faster computation alongside the recent advances in the hardware technology have led to the development of a vast array of many-core and multi-core processors, accelerators, programming models, and language extensions (e.g., CUDA and OpenCL for GPUs, and OpenMP and Cilk for multi-core CPUs). The massive computational power of the parallel processors, specially GPUs, makes them viable targets for carrying out arbitrary-precision integer arithmetic.

At the same time, developing parallel algorithms, followed by implementing and optimizing them as multi-threaded parallel programs imposes a set of challenges. This thesis explains the current state of research on a number of a problems in arbitrary-precision arithmetic on CUDA-enabled GPUs as well as multi-core CPUs, also, proposes a number of solutions for some of the challenging problems related to this subject. The solutions include parallel algorithms, complexity analysis, experimental results, and finally, critical implementation tricks for each problem. Combining the solutions together, the goal is to maximize the performance of arbitrary-precision integer arithmetic on parallel hardware. This work is inspired by the previous research papers, algorithms and software libraries in code generation and optimization such as SPIRAL [5] and FFTW [6, 7], auto-tuning such as ATLAS [8], and the mathematical libraries such as GMP [9], FLINT [10], and NTL [11].

Note that the emphasis on *arbitrary-precision* arithmetic is to distinguish the proposed solutions from the ones for *fixed multi-precision* arithmetic, where the implementation is specifically tuned for numbers that fit in s machine words, where s is a prescribed and small power of 2, typically between 1 and 8. In our work, this number s of machine words is

- either prescribed in advance but the value of s can be arbitrary large, or

- not prescribed in advance, thus implying that, for an arithmetic operation, input and output numbers may use different values of s.

In the former case, our arithmetic operations take place in a prime field $\mathbb{Z}/p\mathbb{Z}$ where p fits into multiple machine words. Meanwhile, in the latter case, we work over the ring $\mathbb{Z}$ of integers.

1.2 Challenges and objectives

In this section, first, we review the common objectives among the subjects that we have studied. Then, we provide a brief summary of the problems and the proposed solutions.

Common objectives

The primary objective is an end-to-end optimization effort for better use of the hardware resources. To put it another way, maximizing the performance by minimizing the running time throughout the entire system. To be more specific, the main focus in each of the studied problems is to provide a set of ideas, implementation tricks, and experimental results in the following order of priority:

- to design new algorithms, or to adapt existing ones for parallel architectures,
- to use memory hierarchy efficiently in order to minimize the communication overhead, and finally,
- to apply device-specific optimizations to reach to the peak performance on a device; this includes but not limited to loop unrolling, kernel decomposition, using inline assembly, writing code with respect to the way the hardware works (e.g., taking into account the scheduler, instruction-level parallelism, and pipelining features of the device).

Big prime field FFT on GPUs

We consider prime fields of large characteristic $\mathbb{Z}/p\mathbb{Z}$ where p fits on k machine words and k is a power of 2. When the characteristic of these fields is restricted to a subclass of the generalized Fermat numbers, we show that arithmetic operations in such fields offer attractive performance, both in terms of algebraic complexity and parallelism. In particular, these operations can be vectorized, leading to efficient implementation of fast Fourier transforms on graphics processing units. This work demonstrates the potential of GPUs and their huge computational capacity for tackling an essential computational algebra problem, that is, directly computing FFT over large prime fields as a competitive alternative to modular computation of FFT based on the Chinese Remainder Theorem (CRT). We explain more details in Chapter 2.

Big prime field FFT on Multi-core

This work extends the previous study realized on GPUs to multi-core processors. In this new context, we overcome the less fine control of hardware resources by successively using FFT in support of the multiplication in those fields. We obtain favorable speedup factors (up to 6.9x on a 6-core, 12 threads node, and 4.3x on a 4-core, 8 threads node) of our parallel implementation compared to the serial implementation for the overall application thanks to the low memory footprint and the sharp control of arithmetic instructions of our implementation of generalized Fermat prime fields. We explain more details in Chapter 3.

KLARAPTOR: A Tool for Dynamically Finding Optimal Kernel Launch Parameters Targeting CUDA Programs

We present KLARAPTOR (Kernel LAunch parameters RAtional Program estimaTOR), a tool built on top of the LLVM Pass Framework and NVIDIA CUPTI API to dynamically determine the optimal values of kernel launch parameters of a CUDA kernel. We describe a technique to build at the compile-time of a CUDA program a so-called rational program. The rational program, based on some performance prediction model, and knowing particular data and hardware parameters at runtime, can be executed to automatically and dynamically determine the values of launch parameters for the CUDA program that will yield nearly optimal performance. Our underlying technique could be applied to parallel programs in general, given a performance prediction model which accounts for program and hardware parameters. We have implemented and tested our technique in the context of GPU kernels written in CUDA. We explain more details in Chapter 4.

Arbitrary-precision Integer Multiplication on GPUs

In this work, we propose a new fine-grained parallel algorithm for multiplying arbitrary-precision integers of k digits on. This solution is based on classical $O(k^2)$ algorithm. We explain more details in Chapter 5.

2 Big Prime Field FFT on GPUs

2.1 Introduction

Prime field arithmetic plays a central role in computer algebra by supporting computation in Galois fields. The prime fields that are used in computer algebra systems, in particular in the implementation of modular methods, are often of small characteristic, that is, based on prime numbers that fit in a machine word. Increasing precision beyond the machine word size can be done via the Chinese Remainder Theorem (CRT) or Hensel Lemma. However, using machine-word size, thus small, prime numbers yields major issues in certain modular methods, in particular for solving systems of non-linear equations. Indeed, in such circumstances, the so-called *unlucky primes* are to be avoided, see for instance [12, 13] as well as Section 2.9. This makes using larger primes desirable.

We consider prime fields of large characteristic, typically fitting on k machine words, where k is a power of 2. In practice, k typically ranges from 2 to 1024. When the characteristic of these fields is restricted to a subclass of the generalized Fermat numbers, we show that arithmetic operations in such fields offer attractive performance both in terms of algebraic complexity and parallelism. In particular, these operations can be vectorized, leading to efficient implementation of fast Fourier transforms on graphics processing units (GPUs).

We present algorithms for arithmetic operations in a "big" prime field $\mathbb{Z}/p\mathbb{Z}$, where p is a generalized Fermat number of the form $p = r^k + 1$ where r fits a machine-word and k is a power of 2. We report on a GPU implementation of those algorithms as well as a GPU implementation of a Fast Fourier Transform (FFT) over such a big prime field. Our experimental results show that

1. computing an FFT of size N, over a big prime field for p fitting on k 64-bit machine-words, and
2. computing $2k$ FFTs of size N, over a small prime field (that is, where the prime fits a 32-bit half-machine-word) followed by a combination (i.e. CRT-like) of those FFTs

are two competitive approaches in terms of running time. Since the former approach has the advantage of reducing the occurrence of unlucky primes when applying modular methods (in particular in the area of polynomial system solving), we view this experimental observation as a promising result.

The reasons for a GPU implementation are as follows. First, the model of computations and the hardware performance provide interesting opportunities for big prime field arithmetic, in particular in terms of vectorization of the program code. Secondly, highly optimized FFTs over small prime fields have been implemented on GPUs by Wei Pan [14, 15] in the CUMODP library, see www.cumodp.org, and we use them in our experimental comparison.

Section 3.5 reports on various comparative experimentations. First, a comparison of the above two approaches implemented on GPU, exhibiting an advantage for the FFT over a big prime field. Second, a comparison between the two same approaches implemented on a single-core CPU, exhibiting an advantage for the CRT-based FFT over small prime fields. Third, from the two previous comparisons, one deduces a comparison of the FFT over a big prime field (resp. the CRT-based FFT over small prime fields) implemented on GPU and CPU, exhibiting a clear advantage for the GPU implementations. Overall, the big prime field FFT on the GPU is the best approach.

A discrete Fourier transform (DFT) over $\mathbb{Z}/p\mathbb{Z}$, when p is a generalized Fermat prime, can be seen as a generalization of the FNT (Fermat number transform), which is a specific case of the NTT (number theoretic transform). However, the computation of a DFT over $\mathbb{Z}/p\mathbb{Z}$ implies additional considerations, which are not taken into account in the literature on NTT or FNT computations [16, 17].

The computation of a NTT can be done via various methods used for a DFT, among them is the radix-2 Cooley-Tukey, for example. However, the final complexity depends on the way a given DFT is computed. It appears that, in the context of generalized Fermat primes, there is a better choice than the radix-2 Cooley-Tukey. The method used in the present paper is related to the article [18], which is derived from Fürer's algorithm [19] for the multiplication of large integers. The practicality of this latter algorithm is an open question. And, in fact, the work reported in our paper is a practical contribution responding to this open question.

The paper [16] discusses the idea of using Fermat number transform for computing convolutions, thus working modulo numbers of the form $F = 2^b + 1$, where b is a power of 2. This is an effective way to avoid round-off error caused by twiddle factor multiplication in computing DFT over the field of complex numbers. The paper [17] considers *generalized Fermat Mersenne* (GFM) prime numbers that prime of the form $(q^{pn} - 1)/(q - 1)$ where, typically, q is 2 and both p and n are small. These numbers are different from the primes used in our paper, which have the form $r^k + 1$ where r is typically machine-word long and k is a power of 2 so that r is a $2k$-th primitive root of unity, see Section 2.3.

2.2 Complexity analysis

Consider a prime field $\mathbb{Z}/p\mathbb{Z}$ and N, a power of 2, dividing $p - 1$. Then, the finite field $\mathbb{Z}/p\mathbb{Z}$ admits an N-th primitive root of unity (see Section 2.4 for this notion). Denote by ω such an element. Let $f \in \mathbb{Z}/p\mathbb{Z}[x]$ be of degree at most $N - 1$. Then, computing the DFT of f at ω via

an FFT, following the standard 2-way divide-and-conquer algorithm, (see Chapter 8 in [20]) amounts to:

1. $N \log(N)$ additions in $\mathbb{Z}/p\mathbb{Z}$,
2. $(N/2) \log(N)$ multiplications by a power of ω in $\mathbb{Z}/p\mathbb{Z}$.

If the bit-size of p is k machine words, then

1. each addition in $\mathbb{Z}/p\mathbb{Z}$ costs $O(k)$ machine-word operations,
2. each multiplication by a power of ω costs $O(\mathsf{M}(k))$ machine-word operations,

where $n \longmapsto \mathsf{M}(n)$ is a multiplication time as defined in [20]. Therefore, multiplication by a power of ω becomes a bottleneck as k grows. To overcome this difficulty, we consider the following trick proposed by Martin Fürer in [19, 21]. We assume that $N = K^e$ holds for some "small" K, say $K = 32$ and an integer $e \geq 2$. Further, we define $\eta = \omega^{N/K}$, with $J = K^{e-1}$ and assume that multiplying an arbitrary element of $\mathbb{Z}/p\mathbb{Z}$ by η^i, for any $i = 0, \ldots, K-1$, can be done within $O(k)$ machine-word operations. Consequently, every arithmetic operation (addition, multiplication) involved in a DFT of size K, using η as a primitive root, amounts to $O(k)$ machine-word operations. Therefore, such DFT of size K can be performed with $O(K \log(K)\, k)$ machine-word operations. As we shall see in Section 2.3, this latter result holds whenever p is a so called *generalized Fermat number*.

Returning to the DFT of size N at ω and using the factorization formula of Cooley and Tukey, we have

$$\mathrm{DFT}_{JK} \;=\; (\mathrm{DFT}_J \otimes I_K) D_{J,K} (I_J \otimes \mathrm{DFT}_K) L_J^{JK}, \tag{2.1}$$

see Section 2.4. Hence, the DFT of f at ω is essentially performed by:

1. K^{e-1} DFT's of size K (that is, DFT's on polynomials of degree at most $K-1$),
2. N multiplications by a power of ω (coming from the diagonal matrix $D_{J,K}$) and
3. K DFT's of size K^{e-1}.

Unrolling Formula (2.1) so as to replace DFT_J by DFT_K and the other linear operators involved (the diagonal matrix D and the permutation matrix L) one can see that a DFT of size $N = K^e$ reduces to:

1. $e\, K^{e-1}$ DFT's of size K, and
2. $(e-1)\, N$ multiplication by a power of ω.

Recall that the assumption on the cost of a multiplication by η^i, for $0 \leq i < K$, makes the cost for one DFT of size K to $O(K \log_2(K)\, k)$ machine-word operations. Hence, all the DFT's of size K together amount to $O(e\, N \log_2(K) k)$ machine-word operations. That is, $O(N \log_2(N)\, k)$ machine-word operations. Meanwhile, the total cost of the multiplication by a power of ω is $O(e\, N\, \mathsf{M}(k))$ machine-word operations, that is, $O(N \log_K(N)\, \mathsf{M}(k))$ machine-word operations. Indeed, multiplying an arbitrary element of $\mathbb{Z}/p\mathbb{Z}$ by an arbitrary power of ω requires $O(\mathsf{M}(k))$ machine-word operations. Therefore, under our assumption, a

DFT of size N at ω amounts to

$$O(N\ \log_2(N)\, k\ +\ N\ \log_K(N)\, \mathsf{M}(k)) \tag{2.2}$$

machine-word operations. When using generalized Fermat primes, we have $K = 2k$ and the above estimate becomes

$$O(N\ \log_2(N)\, k\ +\ N\ \log_k(N)\, \mathsf{M}(k)) \tag{2.3}$$

The second term in the big-O notation dominates the first one. However, we keep both terms for reasons that will appear shortly.

Without our assumption, as discussed earlier, the same DFT would run in $O(N\ \log_2(N)\, \mathsf{M}(k))$ machine-word operations. Therefore, using generalized Fermat primes brings a speedup factor of $\log(K)$ w.r.t. the direct approach using arbitrary prime numbers.

At this point, it is natural to ask what would be the cost of a comparable computation using small primes and the CRT. To be precise, let us consider the following problem. Let $p_1, \ldots, p_k$ be pairwise different prime numbers of machine-word size and let m be their product. Assume that N divides each of $p_1 - 1, \ldots, p_k - 1$ such that each of fields $\mathbb{Z}/p_1\mathbb{Z}, \ldots, \mathbb{Z}/p_k\mathbb{Z}$ admits an N-th primitive roots of unity, $\omega_1, \ldots, \omega_k$. Then, $\omega = (\omega_1, \ldots, \omega_k)$ is an N-th primitive root of $\mathbb{Z}/m\mathbb{Z}$. Indeed, the ring $\mathbb{Z}/p_1\mathbb{Z} \otimes \cdots \otimes \mathbb{Z}/p_k\mathbb{Z}$ is a direct product of fields. Let $f \in \mathbb{Z}/m\mathbb{Z}[x]$ be a polynomial of degree $N - 1$. One can compute the DFT of f at ω in three steps:

1. Compute the images $f_1 \in \mathbb{Z}/p_1\mathbb{Z}[x], \ldots, f_k \in \mathbb{Z}/p_k\mathbb{Z}[x]$ of f.
2. Compute the DFT of f_i at ω_i in $\mathbb{Z}/p_i\mathbb{Z}[x]$, for $i = 1, \ldots, k$,
3. Combine the results using CRT so as to obtain a DFT of f at ω.

The first and the third above steps will run within $O(N \times \mathsf{M}(k) \log_2(k))$ machine-word operations meanwhile the second one amount to $O(k \times N \log(N))$ machine-word operations, yielding a total of

$$O(N\ \log_2(N)\, k\ +\ N\, \mathsf{M}(k) \log_2(k)) \tag{2.4}$$

These estimates yield a running-time ratio between the two approaches of $\log(N)/\log_2^2(k)$, which suggests that for k large enough the big prime field approach may outperform the CRT-based approach. We believe that this analysis is part of the explanation for the observation that the two approaches are, in fact, competitive in practice, as we shall see in Section 3.5.

We conclude this section by observing that, in the above, we have focused our discussion on algebraic complexity, thus not considering the question of cache complexity. We note that for small prime fields, FFTs that are optimal in terms of cache complexity can be derived easily from the results of [22]. For big prime fields, the same results could be adapted to derive cache complexity optimal FFTs. We leave that for future work. Nevertheless, we can already observe that when the big prime is large enough for one multiplication in the big prime field to fully occupy the L1 cache, then the naive 2-way divide-and-conquer FFT becomes essentially cache complexity optimal.

2.3 Generalized Fermat numbers

The n-th Fermat number, denoted by F_n, is given by $F_n = 2^{2^n} + 1$. This sequence plays an important role in number theory and, as mentioned in the introduction, in the development of asymptotically fast algorithms for integer multiplication [23, 21].

Arithmetic operations modulo a Fermat number are simpler than modulo an arbitrary positive integer. In particular 2 is a 2^{n+1}-th primitive root of unity modulo F_n. Unfortunately, F_4 is the largest Fermat number which is known to be prime. Hence, when computations require the coefficient ring be a field, Fermat numbers are no longer interesting. This motivates the introduction of other family of Fermat-like numbers, see, for instance, Chapter 2 in the text book *Guide to elliptic curve cryptography* [24].

Numbers of the form $a^{2^n} + b^{2^n}$ where $a > 1$, $b \geq 0$ and $n \geq 0$ are called *generalized Fermat numbers*. An odd prime p is a generalized Fermat number if and only if p is congruent to 1 modulo 4. The case $b = 1$ is of particular interest and, by analogy with the ordinary Fermat numbers, it is common to denote the generalized Fermat number $a^{2^n} + 1$ by $F_n(a)$. So 3 is $F_0(2)$. We call a the *radix* of $F_n(a)$. Note that, Landau's fourth problem asks if there are infinitely many generalized Fermat primes $F_n(a)$ with $n > 0$.

In the finite ring $\mathbb{Z}/F_n(a)\mathbb{Z}$, the element a is a 2^{n+1}-th primitive root of unity. However, when using binary representation for integers on a computer, arithmetic operations in $\mathbb{Z}/F_n(a)\mathbb{Z}$ may not be as easy to perform as in $\mathbb{Z}/F_n\mathbb{Z}$. This motivates the following.

Definition 1 *We call* sparse radix generalized Fermat number*, any integer of the form $F_n(r)$ where r is either $2^w + 2^u$ or $2^w - 2^u$, for some integers $w > u \geq 0$. In the former case, we denote $F_n(r)$ by $F_n^+(w, u) = 2^w + 2^u$ and in the latter by $F_n^-(w, u) = 2^w - 2^u$.*

Table 2.1 lists sparse radix generalized Fermat numbers (SRGFNs) that are prime. For each such number p, we give the largest power of 2 dividing $p - 1$, that is, the maximum length N of a vector to which a radix-K FFT algorithm where K is an appropriate power of 2.

Notation 1 *In the sequel, we consider $p = F_n(r)$, a fixed SRGFN. We denote by 2^e the largest power of 2 dividing $p - 1$ and we define $k = 2^n$, so that $p = r^k + 1$ holds.*

As we shall see in the sequel of this section, for any positive integer N which is a power of 2 such that N divides $p - 1$, one can find an N-th primitive root of unity $\omega \in \mathbb{Z}/p\mathbb{Z}$ such that multiplying an element $x \in \mathbb{Z}/p\mathbb{Z}$ by $\omega^{i(N/2k)}$ for $0 \leq i < 2k$ can be done in linear time w.r.t. the bit size of x. Combining this observation with an appropriate factorization of the DFT transform on N points over $\mathbb{Z}/p\mathbb{Z}$, we obtain an efficient FFT algorithm over $\mathbb{Z}/p\mathbb{Z}$.

Table 2.1: SRGFNs of practical interest.

p	$\max\{2^e \text{ s.t. } 2^e \mid p-1\}$
$(2^{63}+2^{53})^2+1$	2^{106}
$(2^{64}-2^{50})^4+1$	2^{200}
$(2^{63}+2^{34})^8+1$	2^{272}
$(2^{62}+2^{36})^{16}+1$	2^{576}
$(2^{62}+2^{56})^{32}+1$	2^{1792}
$(2^{63}-2^{40})^{64}+1$	2^{2560}
$(2^{64}-2^{28})^{128}+1$	2^{3584}

2.3.1 Representation of $\mathbb{Z}/p\mathbb{Z}$

We represent each element $x \in \mathbb{Z}/p\mathbb{Z}$ as a vector $\vec{x} = (x_{k-1}, x_{k-2}, \ldots, x_0)$ of length k and with non-negative integer coefficients such that we have

$$x \equiv x_{k-1}\, r^{k-1} + x_{k-2}\, r^{k-2} + \cdots + x_0 \mod p. \tag{2.5}$$

This representation is made unique by imposing the following constraints

1. either $x_{k-1} = r$ and $x_{k-2} = \cdots = x_1 = 0$,
2. or $0 \le x_i < r$ for all $i = 0, \ldots, (k-1)$.

We also map x to a univariate integer polynomial $f_x \in \mathbb{Z}[T]$ defined by $f_x = \sum_{i=0}^{k-1} x_i t^i$ such that $x \equiv f_x(r) \mod p$.

Now, given a non-negative integer $x < p$, we explain how the representation $\vec{x}$ can be computed. The case $x = r^k$ is trivially handled, hence we assume $x < r^k$. For a non-negative integer z such that $z < r^{2^i}$ holds for some positive integer $i \le n = \log_2(k)$, we denote by $\mathrm{vec}(z, i)$ the unique sequence of 2^i non-negative integers $(z_{2^i-1}, \ldots, z_0)$ such that we have $0 \le z_j < r$ and $z = z_{2^i-1} r^{2^i-1} + \cdots + z_0$. The sequence $\mathrm{vec}(z, i)$ is obtained as follows:

1. if $i = 1$, we have $\mathrm{vec}(z, i) = (q, s)$,
2. if $i > 1$, then $\mathrm{vec}(z, i)$ is the concatenation of $\mathrm{vec}(q, i-1)$ followed by $\mathrm{vec}(s, i-1)$,

where q and s are the quotient and the remainder in the Euclidean division of z by $r^{2^{i-1}}$. Clearly, $\mathrm{vec}(x, n) = \vec{x}$ holds.

We observe that the sparse binary representation of r facilitates the Euclidean division of a non-negative integer z by r, when performed on a computer. Referring to the notations in Definition 1, let us assume that r is $2^w + 2^u$, for some integers $w > u \ge 0$. (The case $2^w - 2^u$ would be handled in a similar way.) Let z_{high} and z_{low} be the quotient and the remainder in

the Euclidean division of z by 2^w. Then, we have

$$z = 2^w z_{\text{high}} + z_{\text{low}} = r z_{\text{high}} + z_{\text{low}} - 2^u z_{\text{high}}. \tag{2.6}$$

Let $s = z_{\text{low}} - 2^u z_{\text{high}}$ and $q = z_{\text{high}}$. Three cases arise:

(S1) if $0 \leq s < r$, then q and s are the quotient and remainder of z by r,
(S2) if $r \leq s$, then we perform the Euclidean division of s by r and deduce the desired quotient and remainder,
(S3) if $s < 0$, then (q, s) is replaced by $(q + 1, s + r)$ and we go back to Step (S1).

Since the binary representations of r^2 can still be regarded as sparse, a similar procedure can be done for the Euclidean division of a non-negative integer z by r^2. For higher powers of r, we believe that Montgomery multiplication [25] is the way to go, though this remains to be explored.

2.3.2 Finding primitive roots of unity in $\mathbb{Z}/p\mathbb{Z}$

Notation 2 *Let N be a power of 2, say 2^ℓ, dividing $p-1$ and let $g \in \mathbb{Z}/p\mathbb{Z}$ be an N-th primitive root of unity.*

Recall that such an N-th primitive root of unity can be obtained by a simple probabilistic procedure. Write $p = qN + 1$. Pick a random $\alpha \in \mathbb{Z}/p\mathbb{Z}$ and let $\omega = \alpha^q$. Little Fermat theorem implies that either $\omega^{N/2} = 1$ or $\omega^{N/2} = -1$ holds. In the latter case, ω is an N-th primitive root of unity. In the former, another random $\alpha \in \mathbb{Z}/p\mathbb{Z}$ should be considered. In our various software implementation of finite field arithmetic [26, 27, 28], this procedure finds an N-th primitive root of unity after a few tries and has never been a performance bottleneck.

In the following, we consider the problem of finding an N-th primitive root of unity ω such that $\omega^{N/2k} = r$ holds. The intention is to speed up the portion of FFT computation that requires to multiply elements of $\mathbb{Z}/p\mathbb{Z}$ by powers of ω.

Proposition 1 *In $\mathbb{Z}/p\mathbb{Z}$, the element r is a $2k$-th primitive root of unity. Moreover, the following algorithm computes an N-th primitive root of unity $\omega \in \mathbb{Z}/p\mathbb{Z}$ such that we have $\omega^{N/2k} = r$ in $\mathbb{Z}/p\mathbb{Z}$.*

Proof Since $g^{N/2k}$ is a $2k$-th root of unity, it is equal to r^{i_0} (modulo p) for some $0 \leq i_0 < 2k$ where i_0 is odd. Let j be a non-negative integer. Observe that we have

$$g^{j2^\ell/2k} = (g^i\, g^{2\,k\,q})^{2^\ell/2k} = g^{i2^\ell/2k} = r^{i\ i_0}, \tag{2.7}$$

where q and i are quotient and the remainder of j in the Euclidean division by $2k$. By definition of g, the powers $g^{i2^\ell/2k}$, for $0 \leq i < 2k$, are pairwise different. It follows from Formula (2.7) that the elements $r^{i\ i_0}$ are pairwise different as well, for $0 \leq i < 2k$. Therefore, one of

Algorithm 1 Find a primitive N-th root of unity $\omega \in \mathbb{Z}/p\mathbb{Z}$ such that $\omega^{N/2k} = r$.

input:

- Exponent N.
- Radix r and exponent k from $p = r^k + 1$.
- An N-th root of unity $g \in \mathbb{Z}/p\mathbb{Z}$.

output:

- An N-th primitive root of unity $\omega \in \mathbb{Z}/p\mathbb{Z}$ such that $\omega^{N/2k} = r$.

procedure PrimitiveRootAsRootOf(N, r, k, g)
  $\alpha := g^{N/2k}$
  $\beta := \alpha$
  $j := 1$
  while $\beta \neq r$ **do**
    $\beta := \alpha\beta$
    $j := j + 1$
  end while
  $\omega := g^j$
  return (ω)
end procedure

those latter elements is r itself. Hence, we have j_1 with $0 \leq j_1 < 2k$ such that $g^{j_1 N/2k} = r$. Then, $\omega = g^{j_1}$ is as desired and Algorithm 1 computes it. □

2.3.3 Addition and subtraction in $\mathbb{Z}/p\mathbb{Z}$

Let $x, y \in \mathbb{Z}/p\mathbb{Z}$ represented by $\vec{x}, \vec{y}$, see Section 2.3.1 for this latter notation. Algorithm 2 computes the representation $\overrightarrow{x + y}$ of the element $(x + y) \mod p$.

Proof At Step (1), $\vec{x}$ and $\vec{y}$, regarded as vectors over $\mathbb{Z}$, are added component-wise. At Steps (2) and (3), the carry, if any, is propagated. At Step (4), there is no carry beyond the leading digit z_{k-1}, hence $(z_{k-1}, \ldots, z_0)$ represents $x + y$. Step (5) handles the special case where $x + y = p - 1$ holds. Step (6) is the *overflow* case which is handled by subtracting $1 \mod p$ to $(z_{k-1}, \ldots, z_0)$, finally producing $\overrightarrow{x + y}$. □

A similar procedure computes the vector $\overrightarrow{x - y}$ representing the element $(x - y) \in \mathbb{Z}/p\mathbb{Z}$. Recall that we explained in Section 2.3.1 how to perform the Euclidean divisions at Step (S3) in a way that exploits the sparsity of the binary representation of r.

In practice, the binary representation of the radix r fits a machine word, see Table 2.1. Consequently, so does each of the "digit" in the representation $\vec{x}$ of every element $x \in \mathbb{Z}/p\mathbb{Z}$. This allows us to exploit machine arithmetic in a sharper way. In particular, the Euclidean divisions at Step (S3) can be further optimized.

Algorithm 2 Computing $x + y \in \mathbb{Z}/p\mathbb{Z}$ for $x, y \in \mathbb{Z}/p\mathbb{Z}$

input:

- Elements $x, y \in \mathbb{Z}/p\mathbb{Z}$ represented by $\vec{x}, \vec{y}$.
- Radix r and exponent k from $p = r^k + 1$.

output:

- Result of addition $x + y$.

procedure BIGPRIMEFIELDADDITION($\vec{x}, \vec{y}, r, k$)

1: compute $z_i = x_i + y_i$ in $\mathbb{Z}$, for $i = 0, \ldots, k-1$,

2: let $z_k = 0$,

3: for $i = 0, \ldots, k-1$, compute the quotient q_i and the remainder s_i in the Euclidean division of z_i by r, then replace (z_{i+1}, z_i) by $(z_{i+1} + q_i, s_i)$,

4: if $z_k = 0$ then return $(z_{k-1}, \ldots, z_0)$,

5: if $z_k = 1$ and $z_{k-1} = \cdots = z_0 = 0$, then let $z_{k-1} = r$ and return $(z_{k-1}, \ldots, z_0)$,

6: let i_0 be the smallest index, $0 \leq i_0 \leq k$, such that $z_{i_0} \neq 0$, then let $z_{i_0} = z_{i_0} - 1$, let $z_0 = \cdots = z_{i_0 - 1} = r - 1$ and return $(z_{k-1}, \ldots, z_0)$.

end procedure

2.3.4 Multiplication by a power of r in $\mathbb{Z}/p\mathbb{Z}$

Before considering the multiplication of two arbitrary elements $x, y \in \mathbb{Z}/p\mathbb{Z}$, we assume that one of them, say y, is a power of r, say $y = r^i$ for some $0 < i < 2k$. Note that the cases $i = 0 = 2k$ are trivial. Indeed, recall that r is a $2k$-th primitive root of unity in $\mathbb{Z}/p\mathbb{Z}$. In particular, $r^k = -1$ in $\mathbb{Z}/p\mathbb{Z}$. Hence, for $0 < i < k$, we have $r^{k+i} = -r^i$ in $\mathbb{Z}/p\mathbb{Z}$. Thus, let us consider first the case where $0 < i < k$ holds. We also assume $0 \leq x < r^k$ holds in $\mathbb{Z}$, since the case $x = r^k$ is easy to handle. From Equation (2.5) we have:

$$\begin{aligned} x r^i &\equiv (x_{k-1}\, r^{k-1+i} + \cdots + x_0\, r^i) \mod p \\ &\equiv \sum_{j=0}^{j=k-1} x_j r^{j+i} \mod p \\ &\equiv \sum_{h=i}^{h=k-1+i} x_{h-i} r^h \mod p \\ &\equiv \Big(\sum_{h=i}^{h=k-1} x_{h-i} r^h - \sum_{h=k}^{h=k-1+i} x_{h-i} r^{h-k}\Big) \mod p \end{aligned}$$

The case $k < i < 2k$ can be handled similarly. Also, in the case $i = k$ we have $x r^i = x(p-1) = -x$ in $\mathbb{Z}/p\mathbb{Z}$. It follows, that for all $0 < i < 2k$, computing the product $x\, r^i$ simply reduces to computing a subtraction. This fact, combined with Proposition 1, motivates the development of FFT algorithms over $\mathbb{Z}/p\mathbb{Z}$.

2.3.5 Multiplication in $\mathbb{Z}/p\mathbb{Z}$

Let again $x, y \in \mathbb{Z}/p\mathbb{Z}$ represented by $\vec{x}, \vec{y}$ and consider the univariate polynomials $f_x, f_y \in \mathbb{Z}[T]$ associated with x, y; see Section 2.3.1 for this notation. To compute the product $x\,y$ in $\mathbb{Z}/p\mathbb{Z}$, we proceed as follows.

Algorithm 3 Computing $xy \in \mathbb{Z}/p\mathbb{Z}$ for $x, y \in \mathbb{Z}/p\mathbb{Z}$

input:

- Polynomials $f_x, f_y \in \mathbb{Z}[T]$ associated with $x, y \in \mathbb{Z}/p\mathbb{Z}$.
- Radix r and exponent k from $p = r^k + 1$.

output:

- Result of multiplication $x\,y$.

procedure BigPrimeFieldMultiplication(f_x, f_y, r, k)

1: We compute the polynomial product $f_u = f_x f_y$ in $\mathbb{Z}[T]$ modulo $T^k + 1$.

2: Writing $f_u = \sum_{i=0}^{k-1} u_i T^i$, we observe that for all $0 \le i \le k-1$ we have $0 \le u_i \le kr^2$ and compute a representation $\overrightarrow{u_i}$ of u_i in $\mathbb{Z}/p\mathbb{Z}$ using the method explained in Section 2.3.1.

3: We compute $u_i r^i$ in $\mathbb{Z}/p\mathbb{Z}$ using the method of Section 2.3.4.

4: Finally, we compute the sum $\sum_{i=0}^{k-1} u_i r^i$ in $\mathbb{Z}/p\mathbb{Z}$ using Algorithm 2.

end procedure

For large values of k, $f_x f_y \mod T^k + 1$ in $\mathbb{Z}[T]$ can be computed by asymptotically fast algorithms (see the paper [29, 18]). However, for small values of k (say $k \le 8$), using plain multiplication is reasonable.

2.4 FFT Basics

We review the Discrete Fourier Transform over a finite field, and its related concepts. See [21] for details.

Primitive and principal roots of unity. Let $\mathcal{R}$ be a commutative ring with units. Let $N > 1$ be an integer. An element $\omega \in \mathcal{R}$ is a *primitive* N-th root of unity if for $1 < k \le N$ we have $\omega^k = 1 \iff k = N$. The element $\omega \in \mathcal{R}$ is a *principal* N-th root of unity if $\omega^N = 1$ and for all $1 \le k < N$ we have

$$\sum_{j=0}^{N-1} \omega^{jk} = 0. \tag{2.8}$$

In particular, if N is a power of 2 and $\omega^{N/2} = -1$, then ω is a principal N-th root of unity. The two notions coincide in fields of characteristic 0. For integral domains every primitive root of unity is also a principal root of unity. For non-integral domains, a principal N-th root of unity is also a primitive N-th root of unity unless the characteristic of the ring $\mathcal{R}$ is a divisor of N.

The discrete Fourier transform (DFT). Let $\omega \in \mathcal{R}$ be a principal N-th root of unity. The *N-point DFT at ω* is the linear function, mapping the vector $\vec{a} = (a_0, \ldots, a_{N-1})^T$ to $\vec{b} = (b_0, \ldots, b_{N-1})^T$ by $\vec{b} = \Omega\vec{a}$, where $\Omega = (\omega^{jk})_{0 \le j,k \le N-1}$. If N is invertible in $\mathcal{R}$, then the N-point DFT at ω has an inverse which is $1/N$ times the N-point DFT at ω^{-1}.

The fast Fourier transform. Let $\omega \in \mathcal{R}$ be a principal N-th root of unity. Assume that N can be factorized to JK with $J, K > 1$. Recall Cooley-Tukey factorization formula [30]

$$\mathrm{DFT}_{JK} \;=\; (\mathrm{DFT}_J \otimes I_K) D_{J,K} (I_J \otimes \mathrm{DFT}_K) L^{JK}_J, \tag{2.9}$$

where, for two matrices A, B over $\mathcal{R}$ with respective dimensions $m \times n$ and $q \times s$, we denote by $A \otimes B$ an $mq \times ns$ matrix over $\mathcal{R}$ called the tensor product of A by B and defined by

$$A \otimes B = [a_{k\ell} B]_{k,\ell} \quad \text{with} \quad A = [a_{k\ell}]_{k,\ell}. \tag{2.10}$$

In the above formula, DFT_{JK}, DFT_J and DFT_K are respectively the N-point DFT at ω, the J-point DFT at ω^K and the K-point DFT at ω^J. The *stride permutation matrix* L^{JK}_J permutes an input vector $\mathbf{x}$ of length JK as follows

$$\mathbf{x}[iJ + j] \mapsto \mathbf{x}[jK + i], \tag{2.11}$$

for all $0 \le j < J$, $0 \le i < K$. If $\mathbf{x}$ is viewed as a $K \times J$ matrix, then L^{JK}_J performs a transposition of this matrix. The *diagonal twiddle matrix* $D_{J,K}$ is defined as

$$D_{J,K} = \bigoplus_{j=0}^{J-1} \mathrm{diag}(1, \omega^j, \ldots, \omega^{j(K-1)}), \tag{2.12}$$

Formula (2.9) implies various divide-and-conquer algorithms for computing DFTs efficiently, often referred to as fast Fourier transforms (FFTs). See the papers [5] and [7] by the authors of the SPIRAL and FFTW projects, respectively. This formula also implies that, if K divides J, then all involved multiplications are by powers of ω^K.

2.5 Blocked FFT on the GPU

In the sequel of this section, let $\omega \in \mathcal{R}$ be a principal N-th root of unity. In the factorization of the matrix DFT_{JK}, viewing the size K as a base case and assuming that J is a power of K,

Formula (2.9) translates into a recursive algorithm. This recursive formulation is, however, not appropriate for generating code targeting many-core GPU-like architectures for which, formulating algorithms iteratively facilitates the division of the work into kernel calls and thread-blocks. To this end, we shall unroll Formula (2.9).

Notation 3 *Assuming $c = 0$, that is, $N = K^e$, we define the following linear operators, for $i = 0, \ldots, e-1$:*

$$\begin{aligned} U_i(\omega) &= \left(I_{K^i} \otimes \mathrm{DFT}_K(\omega^{K^{e-1}}) \otimes I_{K^{e-i-1}}\right) \cdot \\ &\quad \left(I_{K^i} \otimes D_{K,K^{e-i-1}}(\omega^{K^i})\right), \\ V_i(\omega) &= I_{K^i} \otimes L_K^{K^{e-i}}, \\ W_i(\omega) &= I_{K^i} \otimes \left(L_{K^{e-i-1}}^{K^{e-i}} \cdot D_{K,K^{e-i-1}}(\omega^{K^i})\right). \end{aligned} \tag{2.13}$$

Remark 1 *We recall two classical formulas for tensor products of matrices. If A and B are square matrices over $\mathcal{R}$ with respective orders a and b, then we have*

$$A \otimes B = L_a^{ab} \cdot (B \otimes A)\, L_b^{ab}. \tag{2.14}$$

If C and D are two other square matrices over $\mathcal{R}$ with respective orders a and b, then we have

$$(A \otimes B) \cdot (C \otimes D) = (A \cdot C) \otimes (B \cdot D). \tag{2.15}$$

Our GPU implementation reported in Section 2.6 is based on the following two results. We omit the proofs, which can easily be derived from Remark 1 and the Cooley-Tukey factorization formula; see [14]. Computer program code can be generated from Proposition 3 using the techniques of [5].

Proposition 2 *For $i = 0, \ldots, e-1$, we have*

$$U_i(\omega) = V_i(\omega) \left(I_{K^{e-1}} \otimes \mathrm{DFT}_K(\omega^{K^{e-1}})\right) W_i(\omega) \tag{2.16}$$

The following formula reduces the computation of a DFT on K^e points to computing e DFT's on K points.

Proposition 3 *The following factorization of $\mathrm{DFT}_{K^e}(\omega)$ holds:*

$$\mathrm{DFT}_{K^e}(\omega) = U_0(\omega) \cdots U_{e-1}(\omega) V_{e-1}(\omega) \cdots V_0(\omega). \tag{2.17}$$

2.6 Implementation

In this section, we discuss implementation techniques. Our experimental results are reported in Section 3.5. We have realized a GPU implementation in the CUDA language of the algorithms presented in Sections 2.3 and 2.5. We have used the third and the fourth Generalized Fermat primes from Table 1, namely $P_3 := (2^{63} + 2^{34})^8 + 1$ and $P_4 := (2^{62} + 2^{36})^{16} + 1$. We have tested our code and collected the experimental data on three different types of GPU cards.

Parallelization. Performing arithmetic operations on vectors of elements of $\mathbb{Z}/p\mathbb{Z}$ has inherent data parallelism, which is ideal for implementation on GPUs. In our implementation, each arithmetic operation is computed by one thread. An alternative approach would be to use multiple threads for computing one operation. However, this would not improve performance mostly due to overhead of handling carry propagation (in the case of addition and subtraction), or increased latency because of frequent accesses to global memory (in the case of twiddle factor multiplications).

Memory-bound kernels. Performance of our GPU kernels are limited by frequent accesses to memory. Therefore, we have considered solutions for minimizing memory latency, maximizing occupancy (i.e. number of active warps on each streaming multiprocessor) to hide latency, and maximizing IPC (instructions per clock cycle).

Location of data. At execution time, each thread needs to perform computation on at least one element of $\mathbb{Z}/p\mathbb{Z}$, meaning that it will read/write at least k digits of machine-word size. Often, in such a scenario, shared memory is utilized as an auxiliary memory, but this approach has two shortcomings. First, on a GPU, each streaming multiprocessor has a limited amount of shared memory which might not be large enough for allowing each thread to keep at least one element of $\mathbb{Z}/p\mathbb{Z}$ (since the value of k can be quite large). Second, using a huge amount of shared memory will reduce occupancy. At the same time, there is no opportunity for using texture memory or constant memory when computing over $\mathbb{Z}/p\mathbb{Z}$. Conclusively, the only remaining solution is to keep all data on global memory.

Maximizing global memory efficiency. Assume that for a vector of N elements of $\mathbb{Z}/p\mathbb{Z}$, consecutive digits of each element of $\mathbb{Z}/p\mathbb{Z}$ are stored in adjacent memory addresses. Therefore, such a vector can be considered as the row-major layout of a matrix with N rows and k columns. In practice, this data structure will hurt performance due to increased memory overhead, caused by non-coalesced accesses to global memory. In this case, an effective solution is to apply a stride permutation L_k^{kN} on all input vectors (if data is stored in a row-major layout, this permutation is equivalent to transposing the input to a matrix of k rows and N columns). Therefore, all kernels are written with the assumption that consecutive digits of the same element are N steps away from each other in the memory. As a result, accesses to global memory will be coalesced, increasing memory load and store efficiency, and lowering the memory overhead.

Decomposing computation into multiple kernels. Inside a kernel, consuming too many reg-

isters per thread can lower occupancy, or even worse, lead to register spilling. In order to prevent from register spilling, register-intensive kernels are broken into multiple smaller kernels.

Size of thread blocks. Our GPU kernels do not depend on the size of a thread block. So, we choose a configuration for a thread block that will maximize the percentage of occupancy, the value of IPC (instruction per clock cycle), and bandwidth-related performance metrics such as the load and store throughput. We have achieved the best experimental results for thread blocks of 128 threads, or 256 threads.

Effect of GPU instructions on performance. Our current implementation is optimized for the primes $P_3 := (2^{63} + 2^{34})^8 + 1$ and $P_4 := (2^{62} + 2^{36})^{16} + 1$. Therefore, we rely on 64-bit instructions on GPUs. As it is explained in [31], even though 64-bit integer instructions are supported on NVIDIA GPUs, at compile time, all arithmetic and memory instructions will first be converted to a sequence of 32-bit equivalents. This might have a negative impact on the overall performance of our implementation. Specially, compared to addition and subtraction, 64-bit multiplication is computed through a longer sequence of 32-bit instructions. Finally, using 32-bit arithmetic provides more opportunities for optimization such as instruction level parallelism.

2.7 Experimentation

We compare our implementation of FFT over a big prime field against a comparable approach based on FFTs over small prime fields. To be precise, we implement the two approaches discussed in Section 2.2. Recall that the first approach computes an FFT of size N over a big prime field of the form $\mathbb{Z}/p\mathbb{Z}$ where p is a SRGFN of size k machine words. The second approach uses $s = 2\,k$ half-machine word primes $p_1, \ldots, p_s$ and proceeds as follows:

1. ***projection:*** compute the image f_i of f in $\mathbb{Z}/p_1\mathbb{Z}[x], \ldots, \mathbb{Z}/p_k\mathbb{Z}[x]$, for $i = 1, \ldots, k$,
2. ***images:*** compute the DFT of f_i at ω_i in $\mathbb{Z}/p_i\mathbb{Z}[x]$, for $i = 1, \ldots, k$ (using the CUMODP library [14]),
3. ***combination:*** combine the results using CRT so as to obtain a DFT of f at ω.

We use half-machine word primes (instead of machine-word primes as discussed in Section 2.2) because the small prime field FFTs of the CUMODP library impose this choice. Experimental results are gathered in Section 2.7.1.

We also have implemented and tested a sequential, CPU version of both approaches. For the small prime field approach, we use the NTL library [11], supporting FFT modulo machine-word size primes of 60 bits. However, for the big prime field approach, we have implemented our own arithmetic in a sequential C program. Experimental results are gathered in Section 2.7.2.

2.7.1 Big prime vs. small prime on the GPU

The output of the two approaches is the DFT of a vector of size N over a ring R which is either a prime field or a direct product of prime fields, and for which each element spans k machine-words. Hence these two approaches are equivalent building blocks in a modular method. For realizing the benchmark, first, we perform the reduction step, followed by computing $s = 2k$ FFTs of size N over small prime fields. In the small field case, we use the highly optimized implementation of the following FFT algorithms from the CUMODP library (see [14, 15] and [28]): the Cooley-Tukey FFT algorithm (CT), the Cooley-Tukey FFT algorithm with precomputed powers of the primitive root (CT-pow), and the Stockham FFT algorithm. The above codes compute DFTs for input vectors of 2^n elements, where $20 \leq n \leq 26$ is typical.

Our CUDA implementation of the big prime field approach computes DFT over $\mathbb{Z}/p\mathbb{Z}$, for $P_3 := (2^{63} + 2^{34})^8 + 1$ and $P_4 := (2^{62} + 2^{36})^{16} + 1$, and input vectors of size $N = K^e$ where $K = 16$ for P_3, and $K = 32$ for P_4. Furthermore, for P_3, we have $2 \leq e \leq 5$, while for P_4 (due to the limited size of global memory on a GPU card), we have $2 \leq e \leq 4$.

The benchmark is computed on an NVIDIA Geforce GTX 760M (CC 3.0), an NVIDIA Tesla C2075 (CC 2.0), and an NVIDIA Tesla M2050 (CC 2.0). The first card has effective bandwidth of 48 GB/s, with 4 streaming multiprocessor, and the total number of 768 CUDA cores.

Figures 2.1 and 2.2 show the speedup of the big prime field FFT compared to the small prime field approach, measured on the first GPU card. Moreover, Table 2.2 presents the running times of computing the benchmark on the mentioned GPU cards. In each table, the first three columns give the running times for computing the small prime field FFT based on the Cooley-Tukey algorithm, the Cooley-Tukey FFT algorithm with precomputed powers of the primitive root, and the Stockham algorithm, respectively. Meanwhile, the last column presents the running time for computing the big prime field FFT.

As it is reported in [14], the FFT algorithms of the CUMODP library gain speedup factors for vectors of the size 2^{16} and larger, therefore, the input vector should be large enough to keep the GPU device busy, and thus, provide a high percentage of occupancy. This explains the results displayed on Figures 2.1 and 2.2; for both primes P_3 and P_4, when $N = K^2$ and $N = K^3$, our big prime field FFT approach significantly outperforms the small prime field FFT approach.

More importantly, for both primes P_3 and P_4, and with vectors of size $N = K^4$, our experimental results demonstrate that computing the big prime field FFT is competitive with the small prime field approach in terms of running time. For both primes P_3 and P_4, we can compute FFT for an input vector of size $N = K^4$, which is equivalent of 2^{16} and 2^{20} elements, respectively, and is large enough to cover many practical applications.

Eventually, for P_3, and for a vector of size $N = K^5$, the Cooley-Tukey (with precomputation) and Stockham FFT codes are slightly faster than the big prime field FFT. Nevertheless, for each of the tested big primes, there is a bit size range of input vectors over which the big prime

field approach outperforms the small prime approach, which is coherent with the analysis of Section 2.2. For $P_3 := (2^{63} + 2^{34})^8 + 1$, this range is $[2^{12}, 2^{16}]$ while for $P_4 := (2^{62} + 2^{36})^{16} + 1$, this range is $[2^{15}, 2^{20}]$. Our GPU implementation of the big prime field arithmetic is generic and thus can support larger SRGFNs, see Table 1.

Table 2.2: Running time of computing the benchmark for $N = K^e$ on GPU (timings in milliseconds).

Computing the benchmark for $N = K^e$ for $P_3 := (2^{63} + 2^{34})^8 + 1$ ($K = 16$)				
Measured on a NVIDIA GTX-760M GPU				
e	CT	CT-pow	Stockham	Big FFT
2	8.30	2.73	5.29	0.05
3	10.96	6.49	8.55	1.24
4	50.49	30.29	34.37	26.06
5	820.82	444.07	490.72	558.22
Measured on a NVIDIA Tesla C2075 GPU				
e	CT	CT-pow	Stockham	Big FFT
2	9.44	2.93	5.16	0.03
3	11.72	6.27	7.54	0.89
4	31.85	15.57	19.07	17.71
5	418.58	191.57	205.13	371.48
Measured on a NVIDIA Tesla M2050 GPU				
e	CT	CT-pow	Stockham	Big FFT
2	12.92	3.12	5.35	0.03
3	15.35	6.66	8.00	0.88
4	35.59	15.93	19.62	17.41
5	424.98	198.46	206.71	364.88

Computing the benchmark for $N = K^e$ for $P_4 := (2^{62} + 2^{36})^{16} + 1$ ($K = 32$)				
Measured on NVIDIA GTX-760M GPU				
e	CT	CT-pow	Stockham	Big FFT
2	18.30	9.33	12.26	0.37
3	62.98	39.74	46.72	20.40
4	1772.9	974.01	1042.62	971.28
5	N.A.	N.A.	N.A.	N.A.
Measured on NVIDIA Tesla C2075 GPU				
e	CT	CT-pow	Stockham	Big FFT
2	19.82	9.56	11.56	0.27
3	44.50	23.39	27.98	15.16
4	891.35	437.29	464.69	695.02
5	N.A.	N.A.	N.A.	N.A.
Measured on NVIDIA Tesla M2050 GPU				
e	CT	CT-pow	Stockham	Big FFT
2	27.22	9.91	11.62	0.27
3	51.81	23.93	28.60	14.80
4	902.35	449.53	465.51	678.34
5	N.A.	N.A.	N.A.	N.A.

Figure 2.3 shows the percentage of time spent in each operation in order to compute the big prime field FFT on a randomly generated input vector of size $N = K^4$ (measured for both primes and on the first mentioned GPU card). As illustrated, for both primes, computation follows a similar pattern, where multiplication by twiddle factors is the main bottleneck. Finally, Table 2.3 presents the profiling data for computing the base-case DFT_K on a GTX 760M GPU.

Table 2.4 presents the definitions of `nvprof` metrics according to [4].

2.7.2 CPU vs. GPU implementations

Table 2.5 gathers running times for computing FFT sequentially with both small prime and big prime approaches, on three different CPUs (measured in milliseconds). In addition, Table 2.6 shows the speedup range for computing the small and the big prime field approaches on CPU

Table 2.3: Profiling results for computing base-case DFT_K on a GTX 760M GPU (collected using NVIDIA `nvprof`).

Measured on a GTX760M GPU	$P_3 = (2^{63} + 2^{34})^8 + 1$ ($K = 16$)		$P_4 = (2^{62} + 2^{36})^{16} + 1$ ($K = 32$)	
Metric	Mult by r	Add/Sub	Mult by r	Add/Sub
Achieved Occupancy	74%	45%	62%	46%
Executed IPC	0.72	2.41	0.78	4.56
Instruction Replay Overhead	0.47	0.13	0.53	0.028
Global Load Throughput	24.57 GB/s	20.91 GB/s	46.39 GB/s	10.22 GB/s
Global Store Throughput	22.08 GB/s	20.74 GB/s	44.42 GB/s	9.91 GB/s
Global Memory Load Efficiency	90.44%	43.60%	48.70%	98.86%
Global Memory Store Efficiency	94.95%	43.75%	49.35%	99.99%

Table 2.4: Definitions of NVIDIA `nvprof` metrics according to [4].

Metric	Description
Achieved Occupancy	"Ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor."
Executed IPC	"Instructions executed per cycle."
Replayed instructions ratio	$\frac{\text{\#Instructions issued - \#Instructions executed}}{\text{\#Instructions issued}}$
Instruction Replay Overhead	"Average number of replays for each instruction executed."
Global Load Throughput	"Global memory load throughput" (including effects of cache.)
Global Store Throughput	"Global memory store throughput" (including effects of cache.)
Global Memory Load Efficiency	"Ratio of requested global memory load throughput to required global memory load throughput expressed as percentage."
Global Memory Store Efficiency	"Ratio of requested global memory store throughput to required global memory store throughput expressed as percentage."

and GPU. For each prime, the first and the second column show the lowest and the highest running time of the same approach on CPU and GPU, respectively. Also, the last column contains the lowest and the highest speedup ratio of computing the same approach on CPU to its counterpart on GPU .

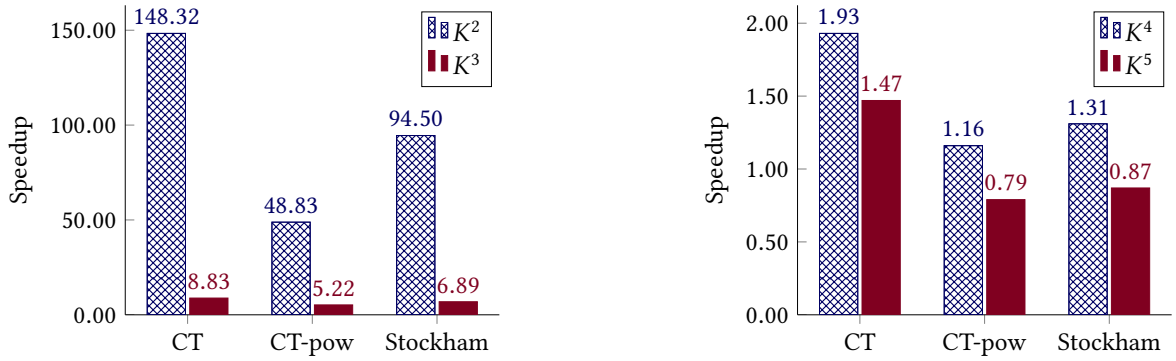

Figure 2.1: Speedup diagram for computing the benchmark for a vector of size $N = K^e$ ($K = 16$) for $P_3 := (2^{63} + 2^{34})^8 + 1$.

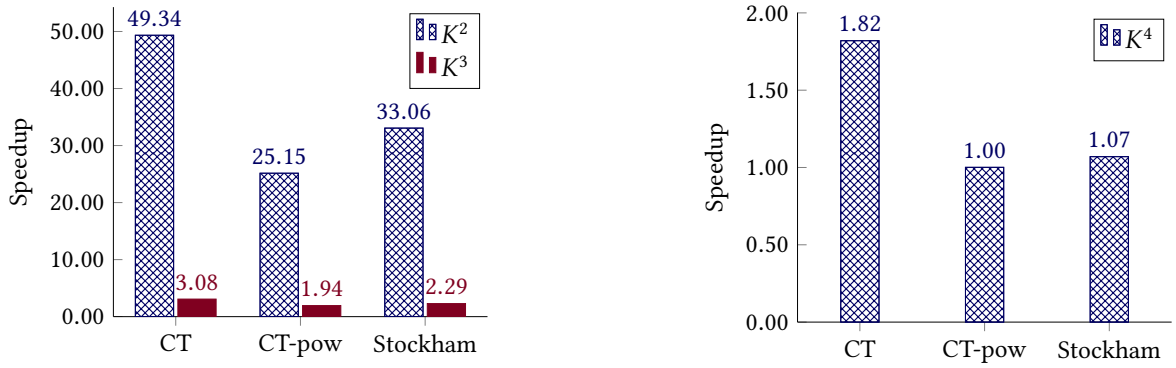

Figure 2.2: Speedup diagram for computing the benchmark for a vector of size $N = K^e$ ($K = 32$) for $P_4 := (2^{62} + 2^{36})^{16} + 1$.

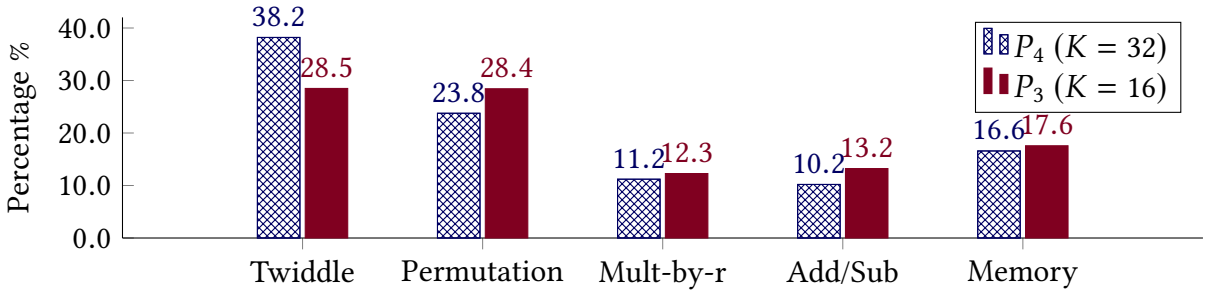

Figure 2.3: Running time of computing DFT_N with $N = K^4$ on a GTX 760M GPU.

Table 2.5: Running time of computing the benchmark for $N = K^e$ using sequential C code on CPU (timings in milliseconds).

Measured on Intel Xeon X5650 @ 2.67GHz CPU		
Computing the benchmark for $N = K^e$ for $P_3 := (2^{63} + 2^{34})^8 + 1$ ($K = 16$)		
e	NTL Small FFT	C Big FFT
2	2.51	1.85
3	23.19	35.08
4	372.19	750.40
Computing the benchmark for $N = K^e$ for $P_4 := (2^{62} + 2^{36})^{16} + 1$ ($K = 32$)		
e	NTL Small FFT	C Big FFT
2	14.94	12.91
3	384.10	692.16
4	11303.76	33351.29

Measured on AMD FX(tm)-8350 @ 2.40GHz CPU		
Computing the benchmark for $N = K^e$ for $P_3 := (2^{63} + 2^{34})^8 + 1$ ($K = 16$)		
e	NTL Small FFT	C Big FFT
2	4.06	4.13
3	16.06	20.01
4	296.00	528.00
Computing the benchmark for $N = K^e$ for $P_4 := (2^{62} + 2^{36})^{16} + 1$ ($K = 32$)		
e	NTL Small FFT	C Big FFT
2	12.00	8.00
3	296.00	396.00
4	10128.00	22992.00

Measured on Intel Core i7-4700HQ @ 2.40GHz CPU		
Computing the benchmark for $N = K^e$ for $P_3 := (2^{63} + 2^{34})^8 + 1$ ($K = 16$)		
e	NTL Small FFT	C Big FFT
2	3.12	0.73
3	14.19	21.06
4	232.76	505.96
Computing the benchmark for $N = K^e$ for $P_4 := (2^{62} + 2^{36})^{16} + 1$ ($K = 32$)		
e	NTL Small FFT	C Big FFT
2	12.48	9.79
3	233.26	496.03
4	7573.65	26089.53

Table 2.6: Speedup ratio ($\frac{T_{\text{CPU}}}{T_{\text{GPU}}}$) for computing the benchmark for $N = K^e$ for P_3 and P_4 (timings in milliseconds).

Computing the benchmark for $N = K^e$ for $P_3 := (2^{63} + 2^{34})^8 + 1$ ($K = 16$) (timings in milliseconds)			
e	NTL Small FFT	Small FFT GPU	Speed-up
2	2.51 - 4.06	2.73 - 12.92	0.19**X** - 1.48**X**
3	14.19 - 23.19	6.27 - 15.35	0.92**X** - 3.69**X**
4	232.76 - 372.19	15.57 - 50.49	4.61**X** - 23.90**X**
e	C Big FFT	BigFFT GPU	Speed-up
2	0.73-4.13	0.03-0.05	14.6**X** - 137.6**X**
3	20.01-35.08	0.88-1.24	16.13**X** - 39.86**X**
4	505.96 - 750.40	17.41-26.06	19.41**X** - 43.10**X**

Computing the benchmark for $N = K^e$ for $P_4 := (2^{62} + 2^{36})^{16} + 1$ ($K = 32$) (timings in milliseconds)			
e	NTL Small FFT	Small FFT GPU	Speed-up
2	12.00 - 14.94	9.33 - 27.22	0.44**X** - 1.60**X**
3	233.26 - 384.10	23.39 - 62.98	3.70**X** - 16.42**X**
4	7573.65 - 11303.76	437.29 - 1772.92	4.27**X** - 25.84**X**
e	C Big FFT	BigFFT GPU	Speed-up
2	8.00 - 12.91	0.27 - 0.37	21.62**X** - 47.81**X**
3	396.00 - 692.16	14.80 - 20.80	19.03**X** - 46.76**X**
4	22992.00 - 33351.29	695.02 - 971.28	23.67**X** - 479.62**X**

2.8 Conclusion

Our results show the advantage of the big prime field approach. To be precise, for a range of vector sizes, one can find a suitable large prime modulo which FFTs outperform the CRT-based approach. The CUDA code presented in this article is part of the CUMODP library freely available at http://www.cumodp.org.

2.9 Appendix: modular methods and unlucky primes

In computer algebra, the so-called *modular methods* are the main application of prime field arithmetic. Let us give a simple example of such methods.

Consider a square matrix A of order n with coefficients in the ring $\mathbb{Z}$ of integers. It is well-

known that $\det(A)$, the determinant of A, can be computed in at most $2n^3$ arithmetic operations in the field $\mathbb{Q}$ of rational numbers, by means of Gaussian elimination. However the cost of each of those operations is not the same and, in fact, depends on the bit size of the rational numbers involved. It can be proved that, if B is the maximum absolute value of a coefficient in A then computing the determinant of A directly (that is, over $\mathbb{Z}$) can be done within $O(n^5\,(\log n + \log B)^2)$ machine-word operations, see the landmark book [20]. If a modular method is used, based on the Chinese Remainder Theorem (CRT), one can reduce the cost to $O(n^4 \log^2(nB)\,(\log^2 n + \log^2 B))$ machine-word operations.

Let us explain how this works. Let d be the determinant of A and let us choose a prime number $p \in \mathbb{Z}$ such that the absolute value $\mid d \mid$ of d satisfies

$$2 \mid d \mid < p.$$

Let r be the determinant of A regarded as a matrix over $\mathbb{Z}/p\mathbb{Z}$ and let us represent the elements of $\mathbb{Z}/p\mathbb{Z}$ within the symmetric range $[-\frac{p-1}{2} \cdots \frac{p-1}{2}]$. Hence we have

$$-\frac{p}{2} < r < \frac{p}{2} \quad \text{and} \quad -\frac{p}{2} < d < \frac{p}{2} \tag{2.18}$$

leading to

$$-p < d - r < p \tag{2.19}$$

Observe that $\det(A)$ is a polynomial expression in the coefficients of A. For instance with $n = 2$ we have

$$\det(A) \;=\; a_{11}\,a_{22} - a_{12}\,a_{21}. \tag{2.20}$$

Denoting by $\overline{x}^p$ the residue class in $\mathbb{Z}/p\mathbb{Z}$ of any $x \in \mathbb{Z}$, we have

$$\overline{x+y}^p \;=\; \overline{x}^p + \overline{y}^p \quad \text{and} \quad \overline{xy}^p \;=\; \overline{x}^p\,\overline{y}^p, \tag{2.21}$$

for all $x, y \in \mathbb{Z}$. It follows for $n = 2$, and using standard notations, that we have

$$\overline{\det(A)}^p \;=\; \overline{a_{11}}^p\,\overline{a_{22}}^p - \overline{a_{12}}^p\,\overline{a_{21}}^p. \tag{2.22}$$

More generally, we have

$$\overline{\det(A)}^p \;=\; \det(A \mod p), \tag{2.23}$$

that is, $d \equiv r \mod p$. This with Relation (2.19) leads to

$$d = r. \tag{2.24}$$

In summary, the determinant of A as a matrix over $\mathbb{Z}$ is equal to the determinant of A regarded as a matrix over $\mathbb{Z}/p\mathbb{Z}$ provided that $2 \mid d \mid < p$ holds. Therefore, the computation of the determinant of A as a matrix over $\mathbb{Z}$ can be done modulo p, which provides a way of controlling expression swell in the intermediate computations.

But if d is what we want to compute, the condition $2 \mid d \mid < p$ is not that helpful for choosing p. However, *Hadamard's inequality* tells us that, if B is the maximum absolute value of an entry of A, then we have

$$\mid d \mid \leq \; n^{n/2} \, B^n. \tag{2.25}$$

One can then choose a prime number p satisfying $2n^{n/2} \, B^n < p$. Of course, such prime may be very large and thus the expected benefit of controlling expression swell may be limited.

An alternative approach is to consider pairwise different prime numbers $p_1, \ldots, p_e$ such that their product exceeds $2n^{n/2} \, B^n$, and each of them fits on a machine-word. Then, computing the determinants of A regarded as a matrix over $\mathbb{Z}/p_1\mathbb{Z}, \ldots, \mathbb{Z}/p_e\mathbb{Z}$ leads to values $r_1, \ldots, r_e$, respectively. Finally, applying the CRT yields d.

The advantage of this alternative approach is that for a prime number p fitting in the machine-word of computer, arithmetic operations modulo p can be implemented efficiently using hardware integer operations.

For an example of a modular method incurring unlucky primes, let us consider the simple problem of computing a Greatest Common Divisor (GCD) of two univariate polynomials with integer coefficients. Let $f = f_n x^n + \cdots + f_0$ and $g = g_m x^m + \cdots + g_0$ be polynomials in x, with respective degrees n and m, and with coefficients in a unique factorization domain (UFD) R. The following matrix is called the *Sylvester matrix* of f and g.

$$\mathrm{Sylv}(f, g) \; = \; \begin{pmatrix} f_0 & & & & & g_0 & & & \\ \vdots & \ddots & & & & \vdots & & & \\ f_{n-i} & & f_0 & & & \vdots & \ddots & & \\ \vdots & \ddots & \vdots & \ddots & & g_{m-i} & \ddots & \ddots & g_0 \\ f_n & & f_{n-i} & & f_0 & \vdots & & \ddots & \vdots \\ & \ddots & \vdots & \ddots & \vdots & g_m & \ddots & & \vdots \\ & & f_n & & f_{n-i} & & & \ddots & g_{m-i} \\ & & & \ddots & \vdots & & & & \vdots \\ & & & & f_n & & & & g_m \end{pmatrix} \tag{2.26}$$

Its determinant is an element of R called the *resultant* of f and g. This determinant is usually denoted by $\mathrm{res}(f, g)$ and enjoys the following property: a GCD h of f and g has degree zero (that is, h is simply an element of R) if and only the $\mathrm{res}(f, g) \neq 0$ holds. In other words, f and g have a non-trivial GCD (that is, a GCD of positive degree) if and only the $\mathrm{res}(f, g) = 0$ holds.

Assume now that R is the ring $\mathbb{Z}$ of the integer numbers and that $\mathrm{res}(f, g) \neq 0$ holds. Suppose that this latter fact is not known and that one is computing a GCD of f and g by means of a modular method based on the CRT. More precisely, we are computing GCDs of f and g modulo sufficiently many prime numbers $p_1, \ldots, p_e$, obtaining polynomials $h_1, \ldots, h_e$ in $\mathbb{Z}/p_1\mathbb{Z}[x], \ldots, \mathbb{Z}/p_e\mathbb{Z}[x]$. If none of the prime numbers $p_1, \ldots, p_e$ divides $\mathrm{res}(f/h, g/h)$,

nor the leading coefficients of f_n and g_m, then combining $h_1, \ldots, h_e$ by CRT yields a GCD of f and g (which, under the assumption $\mathrm{res}(f, g) \neq 0$ turns out to be a constant). However, if one of the prime numbers $p_1, \ldots, p_e$, say p_i, divides $\mathrm{res}(f/h, g/h)$ (even if it does not divide f_n nor g_m) then h_i has a positive degree. It follows that h_i is not a modular image of a GCD of f and g in $\mathbb{Z}[x]$. Therefore, this prime p_i should not be used in our CRT scheme and for this reason is called *unlucky*.

Note that as the coefficients of f and g grow, so will $\mathrm{res}(f, g)$. As a consequence, small primes are likely to be unlucky for input data with large coefficients. While there are tricks to overcome the *noise* introduced by unlucky primes, this can become a serious computational bottleneck. To summarize, certain modular methods, when applied to challenging problems, require the use of prime numbers that do not necessarily fit in a machine-word. This observation motivated the work presented in this chapter.

3 Big Prime Field FFT on Multi-core Processors

3.1 Introduction

Prime field arithmetic plays a central role in computer algebra by supporting computation in Galois fields. The prime fields that are used in computer algebra systems, in particular in the implementation of modular methods, are often of single precision. Increasing precision beyond the machine word size can be done via the Chinese Remainder Theorem (CRT) or Hensel Lemma. However, using machine-word size, thus small, prime numbers has serious inconveniences in certain modular methods, in particular for solving systems of non-linear equations. Indeed, in such circumstances, the so-called unlucky primes are to be avoided, see for instance [12, 13].

We consider prime fields of large characteristic, typically fitting on k machine words, where k is a power of 2. When the characteristic of these fields is restricted to a subclass of the generalized Fermat numbers, the authors of [1] have shown, in an ISSAC 2017 paper, that arithmetic operations in such fields offer attractive performance, both in terms of algebraic complexity and parallelism. In particular, these operations can be vectorized, leading to an efficient implementation of fast Fourier transforms on graphics processing units (GPUs), reported in that same paper.

In the present work, we turn our attention to the most commonly used processors of today's laptops and desktops, namely multi-core processors. These architectures are, in principle, not suitable for fine grained parallelism, in contrast with GPUs. GPUs and multi-core processors differ in memory hierachies as well as communication and synchronization mechanisms between threads. Moreover, GPU architectures offer programmers a finer control of hardware resources than multi-core processors and thus more opportunities to reach high performance. These features of GPU architectures have been essential in the implementation of arithmetic operations of generalized Fermat prime fields. Hence, the implementation techniques developed in [1] can not be easily ported and applied to the context of multi-core processors.

This leads us to a first question: can a serial implementation (written in C programming language) take advantage of the properties of those finite fields towards an implementation

of fast Fourier transform (FFT) over those fields? The answer is yes, however, the route that we took is, of course, quite different than in the GPU case. Instead of performing many batches of arithmetic operations (a natural way of doing things in a GPU implementation) we have focused our effort in optimizing the multiplication between two arbitrary elements of our generalized Fermat prime fields. Consider a generalized Fermat prime number of the form $p = r^k + 1$, where k is a power of 2 and r is of machine-word size. As mentioned in [1], multiplying by a power of r modulo p can be done in $O(k)$ machine-word operations. However, multiplying two arbitrary elements of $\mathbb{Z}/p\mathbb{Z}$ is a non-trivial operation. Note that we encode elements of $\mathbb{Z}/p\mathbb{Z}$ in radix r expansion. Thus, multiplying two arbitrary elements of $\mathbb{Z}/p\mathbb{Z}$ requires computation of the product of two univariate polynomials in $\mathbb{Z}[X]$, of degree less than k, modulo $X^k + 1$. In [1], this is done by using plain multiplication, thus $\Theta(k^2)$ machine-word operations. In Section 3.3, we explain how to multiply two arbitrary elements x, y of $\mathbb{Z}/p\mathbb{Z}$ via FFT.

A second natural question is whether a multi-threaded implementation of big prime field FFT can deliver interesting speedup factors. While obtaining efficient multi-threaded implementation of FFTs with coefficients in single or double precision is a standard research topic [33, 34, 35, 36], the case of higher precision has received little attention so far. With coefficients in the generalized Fermat prime field $\mathbb{Z}/p\mathbb{Z}$, our FFT is in the spirit of the algorithms of Schönhage and Strassen [23] and Fürer [21], where fast multiplication is achieved by "composing" FFTs operating on different vector sizes.

The practicality of Fürer's algorithm is still an open question, a question that we touch in this thesis, without fully addressing it. Several algorithms, similar to Fürer's, have been proposed since. For example, in [37, 38] De et al. gave a similar algorithm which relies on *finite field arithmetic* and achieves the same running time as Fürer's algorithm. Later, Harvey, Van der Hœven and Lecerf proposed, for the integer multiplication, a theoretical improvement to Fürer's algorithm in [39] based on Bluestein's chirp transform. In [40], they also propose a similar algorithm for the multiplication over finite fields, achieving a Fürer-like complexity. This work led to an efficient implementation in [41], using multiplication of polynomials over the special field $\mathbb{F}_{2^{60}}$. In [42], Covanov and Thomé proposed an algorithm based on generalized Fermat primes and the same scheme as Fürer's algorithm, to multiply integers with a Fürer-like complexity.

Returning to our second question, addressing the parallel execution of FFT over big prime fields on multi-cores, the answer is yes. On a 4-core processor and on a 6-core processor, both equipped with hyper-threading technology, we reached nearly linear speedup for the largest input data that we tried.

To measure the benefits of our optimized implementation of the generalized Fermat prime field $\mathbb{Z}/p\mathbb{Z}$, we have realized a naive implementation of the same field, where the radix representation is not used. In this second implementation, the sum $a + b \mod p$ and the product $a \times b \mod p$ are simply computed by calling the modular sum and modular product functions from the GNU Multiple Precision Arithmetic Library (GMP) [9]. The performance of our big prime field FFT degrades substantially with this second implementation of $\mathbb{Z}/p\mathbb{Z}$.

The difference in the performance of the optimized implementation can be attributed to, by our measurements, the sharp management of computing resources (i.e. specialized arithmetic and minimal usage of memory).

The experimental results reported in Section 3.5 support the positive answers to our two questions. Our code is part of the *Basic Polynomial Algebra Subprograms*, also known as the BPAS library [43] and is publicly available at http://www.bpaslib.org/.

3.2 Generalized Fermat prime fields

The residue classes modulo p, where p is a prime number, form a field (unique up to isomorphism) called the *prime field* with p elements, denoted by $\mathrm{GF}(p)$ or $\mathbb{Z}/p\mathbb{Z}$. Single-precision and multi-precision primes are referred to as *small primes* and *big primes*.

Since modular methods for polynomial systems rely on polynomial arithmetic, these large prime numbers must support FFT-based algorithms, such as FFT-based polynomial multiplication. Therefore, we consider the so-called generalized Fermat prime numbers. The detailed introduction of generalized Fermat prime numbers can be found in the previous work of our research group [1].

In this paper, we denote a generalized Fermat prime number p as $p = r^k + 1$, and $\mathbb{Z}/p\mathbb{Z}$ to represent the finite field $\mathrm{GF}(p)$. In particular, in the field $\mathbb{Z}/p\mathbb{Z}$, r is a $2\,k$-th primitive root of unity. Each element $x \in \mathbb{Z}/p\mathbb{Z}$ is represented by a vector $\vec{x} = (x_{k-1}, \ldots, x_0)$ of length k. We can also use a univariate polynomial $f_x \in \mathbb{Z}[R]$ to represent x: we write $f_x = \sum_{i=0}^{k-1} x_i\, R^i$, such that $x \equiv f_x(r) \mod p$. The basic arithmetic algorithms in $\mathbb{Z}/p\mathbb{Z}$ are also introduced in [1] Section 3.

As we have mentioned above, for $p = r^k + 1$, r is a $2\,k$-th primitive root unity in $\mathbb{Z}/p\mathbb{Z}$, Section 3.3 of [1] has provided a very efficient algorithm for multiplication between elements $x, y \in \mathbb{Z}/p\mathbb{Z}$, where one of them is a power of r. We assume that $y = r^i$ for some $0 \le i \le 2k$. The cases $i = 0$ and $i = 2k$ are trivial, since r is a $2k$-th primitive root of unity in $\mathbb{Z}/p\mathbb{Z}$, we have $r^0 = r^{2k} = 1$. Also we have $r^k = -1$ in $\mathbb{Z}/p\mathbb{Z}$, so that for $i = k$, we have $x = -x$ and for $k < i < 2k$, $r^i = -r^{i-k}$ holds. Now let us only consider the case $0 < i < k$, where we have the following equation:

$$\begin{aligned} x r^i &\equiv (x_{k-1}\, r^{k-1+i} + \cdots + x_0\, r^i) \mod p \\ &\equiv \sum_{j=0}^{j=k-1} x_j r^{j+i} \mod p \equiv \sum_{h=i}^{h=k-1+i} x_{h-i} r^h \mod p \\ &\equiv (\sum_{h=i}^{h=k-1} x_{h-i} r^h - \sum_{h=k}^{h=k-1+i} x_{h-i} r^{h-k}) \mod p \end{aligned}$$

We see that for all $0 \le i \le 2k$, the product $x \cdot r^i$ is reduced to a shift and a subtraction. We call this process *cyclic shift*.

The C implementation can be found in the BPAS library [43], we refer to this function as MulPowR in this paper. Our main motivation for using generalized Fermat primes is that, thanks to cyclic shifts, multiplications of elements of $\mathbb{Z}/p\mathbb{Z}$ by a power of r are computationally cheap; this offers the opportunity to reduce the average time spent in multiplication operations during the execution of FFT algorithm over such finite fields. Multiplication between two arbitrary elements in $\mathbb{Z}/p\mathbb{Z}$ can be very complicated and expensive, our previous work [1] gave a theoretical algorithm of computing the product $x\,y \in \mathbb{Z}/p\mathbb{Z}$ using polynomial multiplication (See Algorithm 3 in [1]). In the following section, we will discuss the multiplication between arbitrary elements in more detail, and explain the C implementation.

3.3 Optimizing multiplication in generalized Fermat prime fields

In this section, we discuss how we can efficiently multiply two arbitrary elements in $\mathbb{Z}/p\mathbb{Z}$ (when p is a generalized Fermat prime) using FFT. In Section 3.3.1, we outline an algorithm based on polynomial multiplication via FFT. In Section 3.3.2 we present an implementation of the FFT-based multiplication, then, proceed by explaining each sub-routine.

3.3.1 Algorithms

For a generalized Fermat prime p, our approach follows the concepts from Section 3.2, which treats any two elements x and y of $\mathbb{Z}/p\mathbb{Z}$ as polynomials f_x and f_y, then, uses polynomial multiplication algorithms to obtain the product xy. In practice, there are more details to be considered in order to reach high-performance. For instance, how do we efficiently convert a positive integer in the range $(0, r^3)$ into radix-r representation.

Consider $u = x\,y \mod p$ with $x, y, u \in \mathbb{Z}/p\mathbb{Z}$. We use the polynomial representation of the elements in the field, that is, $f_x(R) = x_{k-1}\,R^{k-1} + \cdots + x_1\,R + x_0$ and $f_y(R) = y_{k-1}\,R^{k-1} + \cdots + y_1\,R + y_0$. The first step is to multiply the two polynomials f_x and f_y. Computing $f_u(R) = f_x(R) \cdot f_y(R) \mod (R^k + 1)$ can be interpreted as a *negacyclic convolution*. A cyclic convolution computes $f(x) \cdot g(x) \mod (x^n - 1)$ for two polynomials f and g with degree less than n. Fast algorithms for computing cyclic convolutions via discrete Fourier transform (DFT) are presented, for instance, in [44]. Similar approaches can be used for computing negacyclic convolutions.

Let q be a prime, ω be an n-th primitive root of unity in $\mathbb{Z}/q\mathbb{Z}$, and θ be a $2n$-th primitive root of unity in $\mathbb{Z}/q\mathbb{Z}$. Also, we have two polynomials $f(x)$ and $g(x)$ with degree less than n, we use $\vec{a}$ and $\vec{b}$ to represent the coefficient vector of the f and g. The negacyclic convolution of f and g can be computed as follows:

$$\vec{A'} \cdot \mathsf{InverseDFT}(\mathsf{DFT}(\vec{A} \cdot \vec{a}) \cdot \mathsf{DFT}(\vec{A} \cdot \vec{b})) \tag{3.1}$$

where $\vec{A} = (1, \theta, \ldots, \theta^{n-1})$ and $\vec{A'} = (1, \theta^{-1}, \ldots, \theta^{1-n})$. All the dots between vectors are point-wise multiplications. The InverseDFT and DFTs are all computed at k points. In our implementation, we use unrolled DFTs (similar to the base-case DFTs given in Section 3.4.3 but relying on prime field arithmetic for a single machine word).

Notice that for f_x and f_y in $\mathbb{Z}/p\mathbb{Z}$, the size of each coefficient must be at most 63 bits wide. This implies that when we compute $f_u(R) = f_x(R) \cdot f_y(R) \mod (R^k+1)$, the size of the coefficients of f_u will be at most $\log_2 k + (2 \times 63) = \log_2 k + 126$, which is more than one machine word. We overcome this situation by means of a scheme based on the Chinese Remainder Theorem (CRT).

For k small enough, we use two machine word size primes p_1 and p_2 satisfying the relation of $R \leq \frac{p_1 p_2 - 1}{2}$ where $R = k\, r^2$ is greater or equal than each of $|u_0|, \ldots, |u_{k-1}|$. Let m_1 and m_2 be two integers such that $p_1 m_1 + p_2 m_2 = 1$. Then, each coefficient u_i of f_u can be computed using the Chinese Remaindering Theorem.

Now each coefficient u_i of f_u is the combination of k terms, so the absolute value of each u_i is bounded over by $k \cdot r^2$ which implies that it needs at most $\lfloor \log_2 k r^2 \rfloor + 1$bits to be encoded. Since k is usually between 4 to 256, a radix r representation of u_i of length 3 is sufficient to encode u_i. Hence, we denote by $[c_i, h_i, l_i]$ the 3 integers uniquely given by $u_i = c_i r^2 + h_i r + l_i$, where $0 \leq h_i, l_i < r$ and $c_i \in [-(k-1), k]$.

Then, we can rewrite:

$$f_u(R) = f_x(R) \cdot f_y(R) \mod (R^k + 1) = \sum_{i=0}^{k-1} (c_i\, R^{2+i} + h_i\, R^{1+i} + l_i\, R^i).$$

Now, we have all the coefficients of f_u in the form of $[l, h, c]$. Rearranging the k $[l, h, c]$ vectors gives us three vectors $\vec{l} = [l_0, \ldots, l_{k-1}], \vec{h} = [h_0, \ldots, h_{k-1}]$ and $\vec{c} = [c_0, \ldots, c_{k-1}]$.

Finally, we compute $\vec{l} + \vec{h}.r + \vec{c}.r^2$ to get the final result of $x\, y \in \mathbb{Z}/p\mathbb{Z}$. We refer to this approach of multiplying two arbitrary elements in $\mathbb{Z}/p\mathbb{Z}$ as the FFT-based multiplication in the generalized Fermat prime field. The complete solution is presented in Algorithm 4.

3.3.2 Implementation in C

In this section, we describe our implementation of the FFT-based multiplication for two arbitrary elements of $\mathbb{Z}/p\mathbb{Z}$. We follow the ideas of Algorithm 4 and take care of implementation details.

Note that Algorithm 4 heavily relies on single-precision modular multiplications, especially in the convolution step. To maximize practical performance, we use Montgomery's tricks from [45] for performing operations in $\mathbb{Z}/p\mathbb{Z}$, in particular multiplication. We use the improved Montgomery multiplication (similar to an algorithm from [46]) which we have implemented using *inline assembly* in C. The code can be found in the BPAS library.

Algorithm 4 FFT-based multiplication for two arbitrary elements in $\mathbb{Z}/p\mathbb{Z}$.

input:
- two vectors $\vec{x}$ and $\vec{y}$ representing the two elements x and y in $\mathbb{Z}/p\mathbb{Z}$,
- two number r and k such that $p = r^k + 1$ is a generalized Fermat number.

output:
- a vector $\vec{u}$ representing the result of $x \cdot y \in \mathbb{Z}/p\mathbb{Z}$.

constant values:
- two machine word size primes p_1 and p_2,
- two numbers m_1 and m_2 such that $p_1\, m_1 + p_2\, m_2 = 1$ holds.

procedure FFT-basedMultiplication($\vec{x}, \vec{y}, r, k$)
  $\vec{z_1}$:= NegacyclicConvolution($\vec{x}, \vec{y}, p_1, k$)
  $\vec{z_2}$:= NegacyclicConvolution($\vec{x}, \vec{y}, p_2, k$)
  for $0 \leq i < k$ **do**
    $[s_{0i}, s_{1i}]$:= CRT($p_1, p_2, m_1, m_2, z_{1i}, z_{2i}$)
    $[l_i, h_i, c_i]$:= LHC(s_{0i}, s_{1i}, r)
  end for
  $\vec{c}$:= MulPowR($\vec{c}, 2, k, r$)
  $\vec{h}$:= MulPowR($\vec{h}, 1, k, r$)
  $\vec{u}$:= BigPrimeFieldAddition($\vec{l}, \vec{h}, k, r$)
  $\vec{u}$:= BigPrimeFieldAddition($\vec{u}, \vec{c}, k, r$)
  return $\vec{u}$
end procedure

Note that in Algorithm 4, both the convolution and CRT steps require a large number of modular multiplication operations. With that in mind, before performing either of the convolutions, we convert the two vectors $\vec{x}$ and $\vec{y}$ into Montgomery representation, once for p_1 and once for p_2. After that, we compute the negacyclic convolutions. Once the convolution is carried out, we need to retrieve the result from the Montgomery representation. This step is performed as part of the CRT computation:

$$\begin{aligned} a'_2 &= (a_2\, m_1) \mod p_2, \\ a'_1 &= (a_1\, m_2) \mod p_1. \end{aligned}$$

In the next step, we compute the second part of the CRT algorithm:

$$a'_2\, p_1 + a'_1\, p_2$$

Note that here we need to perform two 64-bit multiplications (thus using two 128-bit numbers), then, add the results via 128-bit arithmetic. Once again for the sake of efficiency, we

turn to inline assembly in C (the implementation code can be found in the BPAS Library [43]). Finally, for u_i $(0 \leq i < k)$ as a coefficient of $f_u = f_x \cdot f_y \mod (R^k + 1) \in \mathbb{Z}$, the result is stored as a pair of 64-bit numbers $[s_0, s_1]$ so that we have $u_i = s_1\, 2^{64} + s_0$.

At this point, as we discussed in Section 3.3.1, we need to convert the coefficients of f_u into radix-based representation (l, h, c). Provided that the following relations are satisfied:

$$s_0 = q_0\, r + m_0 \quad \text{with} \quad q_0, m_0 < r, \tag{3.2}$$
$$s_1 = q_1\, r + m_1 \quad \text{with} \quad q_1, m_1 < r, \tag{3.3}$$
$$2^{64} = q_2\, r + m_2 \quad \text{with} \quad q_2, m_2 < r, \tag{3.4}$$

we proceed by computing the triple $[l', h', c']$ as follows:

$$\begin{aligned}[l', h', c'] &= (q_0\, r + m_0) + (q_1\, r + m_1)\,(q_2\, r + m_2) \\ &= q_1\, q_2\, r^2 + (m_1\, q_2 + m_2\, q_1 + q_0)\, r + (m_0 + m_1\, m_2) \\ &= c'\, r^2 + h'\, r + l'\end{aligned}$$

Notice that the triple $[l', h', c']$ is still not the final result since either of h' or l' can be greater than r. For that matter, we need to compute the quotient and the remainder of h' (resp. l') by r. As the value of r remains constant during the whole computation, we use an adaptation of Barret reduction [47] using 128-bit arithmetic for computing the division by r (for more details, see function div_by_const_R_ptr in [43]). Then, we have

$$l' = h1.r + l1 \quad \text{and} \quad h' = h2.r + l2$$

The final result is computed by the following additions:

$$l + h.r + c.r^2 = [l1, h1, 0] + [l2, h2, 0].r + [0, 0, c']$$

To this end, we have explained the full implementation of the FFT-based multiplication for multiplying two arbitrary elements in $\mathbb{Z}/p\mathbb{Z}$. In Section 3.5, we present experimental results for comparing our implementation against that of the GMP library [9].

3.4 A generic implementation of FFT over prime fields

In Section 3.4.1, we first review the tensor algebra formulation of FFT, following the presentation of [48]. In Section 3.4.2, we explain how one can use the recursive formulation of the six-step DFT to derive an iterative algorithm in which all DFT computations are performed via a fixed size *base-case*. In the context of Generalized Fermat prime fields, this reduction allows us to take advantage of the "cheap" multiplication by powers of the radix r introduced in Section 3.2. Finally, in Section 3.4.3, we discuss the implementation of efficient routines for computing the base-case DFT_K.

3.4.1 The tensor algebra formulation of FFT

In this section, we review the tensor formulation of FFT. Recall that over a commutative ring R, an n-point DFT_n is a linear map from R^n to R^n. For $N = JK$, we use the six-step FFT factorization presented in [48]:

$$\text{DFT}_N = L^N_K \, (I_J \otimes \text{DFT}_K) L^N_J \, D_{K,J} \, (I_K \otimes \text{DFT}_J) \, L^N_K \;\; with \;\; N = J\,K \tag{3.5}$$

Definition 2 *The stride permutation L^{KJ}_K permutes an input vector $\vec{x}$ of length KJ as follows, with $0 \leq i < K$ and $0 \leq j < J$:*

$$\vec{x}[iJ + j] \mapsto \vec{x}[jK + i] \tag{3.6}$$

For an input vector $\vec{x}$ of length KJ, if we look at the vector as a row-major $J \times K$ matrix M, then, the stride permutation L^{KJ}_K is equivalent to performing a transposition on M:

$$L^{KJ}_K(M_{J\times K}) = (M_{J\times K})^T \tag{3.7}$$

For example, let $\vec{x}_8 = [0, 1, 2, 3, 4, 5, 6, 7]$, we compute $L^{2\times 4}_2(\vec{x})$. We can rearrange $\vec{x}$ as a row-major 4×2 matrix M, then, perform a transpose:

$$M^T_{4\times 2} = \begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \\ 6 & 7 \end{bmatrix}^T = \begin{bmatrix} 0 & 2 & 4 & 6 \\ 1 & 3 & 5 & 7 \end{bmatrix}$$

We retrieve the result by reading the consequent rows of M. Therefore, we have $L^{2\times 4}_2(\vec{x}) = [0, 2, 4, 6, 1, 3, 5, 7]$.

Definition 3 *The twiddle factor $D_{K,J}$ is a matrix of the powers of ω:*

$$D_{K,J} = \bigoplus_{j=0}^{K-1} diag\,(1, \omega^j_i, \ldots, \omega^{j(J-1)}_i) \tag{3.8}$$

3.4.2 The BPAS implementation of the FFT

The dominant cost during computation of FFT over $\mathbb{Z}/p\mathbb{Z}$ is the time spent in the multiplication by twiddle factors (powers of root of unity). Even though we can compute all the twiddle factor multiplications with Algorithm 4, however, inspired by the ideas discussed in Fürer's paper [21], our goal is to efficiently compute FFT on a vector of size $N = K^e$ through base-case DFT_K's. We face three main challenges. First, we need an algorithm to reduce the

computation of DFT_N to base-case DFT_K's. Second, we need an efficient implementation of the base-case DFT_K which relies on cheap multiplications by K-th primitive roof of unity (as it is explained in Section 3.2). Finally, we need to have an FFT implementation which can be parallelized on a multi-core CPU, therefore the choice of the FFT algorithm is critical to achieve high performance.

In the BPAS library, and with respect to the above challenges, we decided to implement DFT over $\mathbb{Z}/p\mathbb{Z}$ based on the six-step FFT factorization of [48] (see Equation (3.5) in Section 3.4.1). The six-step FFT factorization provides an easy solution to the first challenge: we simply unroll Equation (3.5) until all DFT computations are performed through a sequence of DFT_K's. The process of reduction to the base-case is as follows. For computing the product $I_K \otimes \mathrm{DFT}_J$, we can further expand it until we reach the base-case DFT_K. The derived solution is presented in Algorithm 5.

Regarding the parallelization, Algorithm 5 is iterative and it has no recursive calls, it only includes a number of nested for-loops. This makes the whole implementation suitable for a parallel implementation on a multi-core CPU. In fact, the inner for-loop nests at Lines L5, L10, L16, L21, L25 can be executed in parallel. On that basis, we have parallelized our implementations of FFT over $\mathbb{Z}/p\mathbb{Z}$ using `Intel CilkPlus`. Experimental results for comparing parallel and serial implementations are reported in Section 3.5.

3.4.3 Efficient implementation of DFT_K

Once again, we benefit from reduction to a base-case. This time, for computing DFT_K, we reduce the whole computation to a sequence of base-case DFT_2's which are defined in the following way:

$$\mathrm{DFT}_2(x_0, x_1) = (x_0 + x_1, x_0 - x_1) \tag{3.9}$$

Then, for $K = 2^n$, we recursively apply the following factorization until all DFT computations are in DFT_2:

$$\mathrm{DFT}_{2^n} = L^{2^n}_2 \, (I_{2^{n-1}} \otimes \mathrm{DFT}_2) \, L^{2^n}_{2^{n-1}} \, D_{2,2^{n-1}} \, (I_2 \otimes \mathrm{DFT}_{2^{n-1}}) \, L^{2^n}_2 \tag{3.10}$$

Now, let us consider the example of base-case DFT_8 in $\mathbb{Z}/p\mathbb{Z}$ when $p = r^4 + 1$. Let us assume that ω_0 is an 8-th primitive root of unity (thus ${\omega_0}^8 = 1$). Also, let $\omega_1 = {\omega_0}^2$, thus a 4-th primitive root of unity (then, ${\omega_1}^4 = {\omega_0}^8 = 1$).

$$\mathrm{DFT}_8(\omega_0) = L^8_2 (I_4 \otimes \mathrm{DFT}_2) \, L^8_4 \, D_{2,4} \, (I_2 \otimes \mathrm{DFT}_4) \, L^8_2 \tag{3.11}$$

$$\mathrm{DFT}_4(\omega_1) = L^4_2 \, (I_2 \otimes \mathrm{DFT}_2) \, L^4_2 \, D_{2,2} \, (I_2 \otimes \mathrm{DFT}_2) \, L^4_2. \tag{3.12}$$

Algorithm 5 Computing DFT on K^e points in $\mathbb{Z}/p\mathbb{Z}$.

input:

- size of the base-case K (8, 16, 32, 64, 128, or 256),
- a positive integer e,
- a vector $\vec{x}$ of size K^e,
- ω which is a K^e-th primitive root of unity in $\mathbb{Z}/p\mathbb{Z}$.

output:

- the final result stored in $\vec{x}$

procedure DFT_GENERAL($\vec{x}, K, e, \omega$)
  for $0 \leq i < e-1$ **do**
    for $0 \leq j < K^i$ **do** ▷ Can be replaced with Parallel-For.
      stride_permutation($x_{jK^{e-i}}, K, K^{e-i-1}$) ▷ Step 1
    end for
  end for
  $\omega_a := \omega^{K^{e-1}}$
  for $0 \leq j < K^{e-1}$ **do** ▷ Can be replaced with Parallel-For.
    $\texttt{idx} := jK$
    DFT_K($x_{\texttt{idx}}, \omega_a$) ▷ Step 2
  end for
  for $e-2 \geq i \geq 0$ **do**
    $\omega_i := \omega^{K^i}$
    for $0 \leq j < K^i$ **do** ▷ Can be replaced with Parallel-For.
      $\texttt{idx} := j\,K^{e-i}$
      twiddle($x_{\texttt{idx}}, K^{e-i-1}, K, \omega_i$) ▷ Step 3
      stride_permutation($x_{\texttt{idx}}, K^{e-i-1}, K$) ▷ Step 4
    end for
    for $0 \leq j < K^{e-1}$ **do** ▷ Can be replaced with Parallel-For.
      $\texttt{idx} := jK$
      DFT_K($x_{\texttt{idx}}, \omega_a$) ▷ Step 5
    end for
    for $0 \leq j < K^i$ **do** ▷ Can be replaced with Parallel-For.
      $\texttt{idx} := jK^{e-i}$
      stride_permutation($x_{\texttt{idx}}, K, K^{e-i-1}$) ▷ Step 6
    end for
  end for
end procedure

Substituting Equation (3.12) in Equation (3.11), we have:

$$\begin{aligned}\mathrm{DFT}_8(\omega_0) =& L_2^8\,(I_4 \otimes \mathrm{DFT}_2)\,L_4^8\,D_{2,4}\,(I_2 \otimes L_2^4)(I_4 \otimes \mathrm{DFT}_2) \\ & (I_2 \otimes L_2^4)(I_2 \otimes D_{2,2})(I_4 \otimes \mathrm{DFT}_2)(I_2 \otimes L_2^4)(L_2^8).\end{aligned} \tag{3.13}$$

The unrolled Equation (3.13) follows from a sequence of basic operations, which helps us in the following ways. First, we avoid performing the permutation and actually moving data around. Instead, we precompute the position of elements after each permutation and hard-code those values in the algorithm for computing the base-case. Also, we reduce the number of multiplications in the base-case. Moreover, each multiplication in the base-case can be reduced to a cyclic shift (as explained in Section 3.2).

Avoiding stride permutations in DFT_K

In our example for DFT_8, there are 4 permutation steps in Equation (3.13). We begin by the two right-most ones, $(I_2 \otimes L^4_2)(L^8_2)$. Rather than moving the data, we precompute the position of permuted elements. Let $\vec{M} = (0,1,2,3,4,5,6,7)$ be the vector containing the initial position of the elements of $\vec{x}$. Then,

$$\vec{M}_1 = L^8_2\vec{M} = (0,2,4,6,1,3,5,7) \tag{3.14}$$

$$\vec{M}_2 = (I_2 \otimes L^4_2)\vec{M}_1 = (0,4,2,6)(1,5,3,7) \tag{3.15}$$

Moving from right to left in Equation (3.13), when we reach $I_4 \otimes \mathrm{DFT}_2$ (the third statement in Equation (3.13)), we apply four DFT_2's on elements of $\vec{x}$, while we retrieve the order of elements as recorded in M_2:

$$\mathrm{DFT}_2(0,4) \longrightarrow \mathrm{DFT}_2(2,6) \longrightarrow \mathrm{DFT}_2(1,5) \longrightarrow \mathrm{DFT}_2(3,7) \tag{3.16}$$

Following this trend, we reach L^8_4 and L^8_2 on the left-most side of Equation (3.13):

$$\vec{M}_3 = (L^8_4)\vec{M}_2 = (0,1,4,5,2,3,6,7) \tag{3.17}$$

$$\vec{M}_4 = (L^8_2)\vec{M}_3 = (0,4,2,6,1,5,3,7) \tag{3.18}$$

At the very end, we need to swap some elements of $\vec{x}$ in order to correct their position in the result vector. That means the position of elements in the result vector must be updated from what they are in $\vec{M}_4$ to the values in M_{out} in the following way:

$$\begin{array}{rcccccccc} \vec{M}_4 & = & (0, & 4, & 2, & 6, & 1, & 5, & 3, & 7) \\ & & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ \vec{M_{out}} & = & (0, & 1, & 2, & 3, & 4, & 5, & 6, & 7) \end{array}$$

Here, rather than permuting the whole vector, we only need to swap the elements that are shown in the same color. For case of DFT_8, we end up swapping only 4 out of 8 elements.

Twiddle multiplications in the DFT_K

Remember Equation (3.8):

$$D_{K,J} = \bigoplus_{j=0}^{K-1} diag\left(1, \omega_i^j, \ldots, \omega_i^{j(J-1)}\right)$$

Then, we have the following twiddle matrices as part of DFT_8:

$$D_{2,2} = (1, 1, \omega_1^0, \omega_1^1) \tag{3.19}$$
$$D_{2,4} = (1, 1, 1, 1, \omega_0^0, \omega_0^1, \omega_0^2, \omega_0^3) \tag{3.20}$$

As we are computing over $\mathbb{Z}/p\mathbb{Z}$ where the prime is $p = r^4 + 1$, then, the radix r is the 8-th root of unity, therefore, can be used for computation of DFT_8. Let $\omega_0 = r$ and $\omega_1 = r^2$, then, the twiddle matrices are updated as follows:

$$D_{2,4} = (1, 1, 1, 1, 1, r, r^2, r^3) \tag{3.21}$$
$$D_{2,2} = (1, 1, 1, r^2) \tag{3.22}$$

We see that more than half of the multiplications in the DFT_8 are by 1 and do not require any actual computation.

More importantly, the multiplications by the powers of the radix are done by cyclic shift from Section 3.2. In a similar way, this argument is valid for any DFT_K as long as we are computing modulo a generalized Fermat prime of the form $p = r^k + 1$.

At the end, putting all the optimizations together, and following Equation (3.13) from right to left, we get an unrolled algorithm presented in Algorithm 6 for computing DFT_8. The algorithm computes the DFT of a vector of size 8 over a generalized Fermat prime in the form of $p = r^4 + 1$, note that r is an 8-th primitive root of unity of p. Following the above process, we have implemented base-cases for K equal to 8, 16, 32, 64, 128, and 256 in the BPAS library. We believe that the currently implemented base-case sizes are large enough for real world applications. Thus, we have skipped prime sizes larger than 128 machine-words in our current implementation.

3.5 Experimentation

In this section, first, we briefly describe the setup used in our experimentation. Then, in Section 3.5.2, we present the comparison of the two implementations of the multiplication in $\mathbb{Z}/p\mathbb{Z}$ introduced in Section 3.3. Section 3.5.3 reports on the results for computing FFT over the big prime fields with the BPAS library. Finally, in Section 3.5.4, we analyze speedup that we gain for parallelizing each approach. All the experimental results have been verified using

Algorithm 6 Unrolled base-case DFT_8 over $\mathbb{Z}/p\mathbb{Z}$ for $p = r^4 + 1$.

1: **input:**
   - a vector $\vec{x}$ of 8 elements of $\mathbb{Z}/p\mathbb{Z}$,
   - the radix r from $p = r^4 + 1$.
2: **output:**
   - the final result stored in $\vec{x}$.
3: **procedure** DFT8($\vec{x}$,r)
4: DFT2(x_0, x_4); DFT2(x_2, x_6); ▷ DFT on permuted indexes.
5: DFT2(x_1, x_5); DFT2(x_3, x_7); ▷ DFT on permuted indexes.
6: $x_6 := x_6\, r^2$; ▷ Twiddle multiplication $x_6\, r^2$.
7: $x_7 := x_7\, r^2$; ▷ Twiddle multiplication $x_7\, r^2$.
8: DFT2(x_0, x_2); DFT2(x_4, x_6); ▷ DFT on permuted indexes.
9: DFT2(x_1, x_3); DFT2(x_5, x_7); ▷ DFT on permuted indexes.
10: $x_5 := x_5\, r^1$; ▷ Twiddle multiplication $x_5\, r^1$.
11: $x_3 := x_3\, r^2$; ▷ Twiddle multiplication $x_3\, r^2$.
12: $x_7 := x_7\, r^3$; ▷ Twiddle multiplication $x_7\, r^3$.
13: DFT2(x_0, x_1); DFT2(x_4, x_5); ▷ DFT on permuted indexes.
14: DFT2(x_2, x_3); DFT2(x_6, x_7); ▷ DFT on permuted indexes.
15: Swap(x_1, x_4); Swap(x_3, x_6); ▷ Final permutation.
16: **return** $\vec{x}$;
17: **end procedure**

equivalent code written in GMP [9].

3.5.1 Experimental setup

Table 3.1 provides the set of prime numbers we use for different base-cases. The k is between 4 and 128 (i.e. up to 128 machine-words).

We have used two node configurations for our benchmarking purposes. The first configuration which we refer to as `Intel-i7-7700K`, has an Intel-i7-7700K 4-core processor (with 8 threads when hyper-threading is enabled), clocking at 4.50 GHz, and equipped with 16 GB of memory (clocking at 2133 MHz). The second configuration which we refer to as `Xeon-X5650` has an Intel Xeon-X5650 processor with 6 physical cores (and 12 threads when hyper-threading is enabled) clocking at 2.66 GHz, and is equipped with 48 GB of memory (clocking at 1133 MHz).

Table 3.1: The set of big primes of different sizes which are used for experimentations.

prime	$K(=2k)$	k	r
P_4	8	4	$2^{59}+2^{58}+2^{11}$
P_8	16	8	$2^{59}+2^{57}+2^{39}$
P_{16}	32	16	$2^{58}+2^{55}+2^{45}$
P_{32}	64	32	$2^{58}+2^{55}+2^{17}$
P_{64}	128	64	$2^{57}+2^{56}+2^{11}$
P_{128}	256	128	$2^{57}+2^{52}+2^{20}$

3.5.2 Multiplication in generalized Fermat prime fields

As discussed in Section 3.3, we provide an algorithm for multiplying two arbitrary elements of the generalized Fermat prime field $\mathbb{Z}/p\mathbb{Z}$ (referred to as GFPF) which relies on negacyclic convolution using DFTs over small prime fields. Our goal is to compare the running-time of our approach with that of the integer arithmetic provided by GMP [9]. To this end, we provide the same input data to both multiplication functions (randomly generated data, but the same data passed to all experiments), the multiplication is carried out, and at the end, the results are verified.

Table 3.2 shows the time (in milliseconds) spent in computation of 10^6 multiplications using each of the two implementations (the number 10^6 is chosen as an input size which is large enough to reduce the errors in time measurement). Also, Table 3.2 shows the running-time ratio of GFPF versus the GMP multiplications. The experimentation has been conducted on `Intel-i7-7700K`. We observe that the GFPF implementation is slower than GMP multiplication, however, the GFPF multiplication becomes faster as the value of k increases.

Table 3.2: The running-time of computing 10^6 modular multiplications in $\mathbb{Z}/p\mathbb{Z}$ for P_8, P_{16}, P_{32}, and P_{64} (measured on `Intel-i7-7700K`).

prime	k	GFPF	GMP	Ratio ($\frac{t_{GFPF}}{t_{GMP}}$)
P_8	8	645 (ms)	171(ms)	3.77**x**
P_{16}	16	1318 (ms)	417 (ms)	3.16**x**
P_{32}	32	2852 (ms)	1179 (ms)	2.41**x**
P_{64}	64	6101 (ms)	3452 (ms)	1.76**x**

Recall that the GFPF multiplication has four steps (see Section 3.3.2):

I. negacyclic convolution (includes converting the vector into Montgomery representation),
II. Chinese remainder algorithm (includes converting the vector out from Montgomery representation),

III. LHC algorithm (fast division of a three machine-word number by radix r), and
IV. cyclic shift, addition, and normalization (carry-handling).

Table 3.3 shows the percentage of time spent in each step of the GFPF multiplication during multiplication of 10^6 arbitrary elements of $\mathbb{Z}/p\mathbb{Z}$, for primes P_8, P_{16}, P_{32}, and P_{64}, collected on `Intel-i7-7700K`. It also presents the actual running-time (shown in milliseconds); clearly, computing the convolution is the dominant cost.

Table 3.3: Time (in milliseconds) and percentage (%) of the total time spent in different steps of computing 10^6 GFPF multiplications of arbitrary elements in $\mathbb{Z}/p\mathbb{Z}$ for primes P_8, P_{16}, P_{32}, and P_{64} (measured on `Intel-i7-7700K`).

prime	k	Convolution		CRT		LHC		Normalization	
		Time	%	Time	%	Time	%	Time	%
P_8	8	323	45	150	21	208	29	35	5
P_{16}	16	851	52	288	18	425	26	64	4
P_{32}	32	2083	57	563	15	847	23	177	5
P_{64}	64	4751	61	1115	14	1497	19	434	6

3.5.3 FFT over big prime fields

In this section, we provide experimental data for computing FFTs over big prime fields. As we have explained in Section 3.4, our FFT implementations which compute DFT on a vector of size $N = K^e$ over $\mathbb{Z}/p\mathbb{Z}$ (with $p = r^k + 1$) are based on Algorithm 5. We compare the running-time of our GFPF implementation versus the GMP implementation, both executed in serial. Once more, we compare the running-time of the two implementations, this time both executed in parallel.

Table 3.4 provides the running-time and running-time ratio for our generalized Fermat prime fields (GFPF) based implementation versus the GMP implementation of computing FFT of size $N = K^e$ over $\mathbb{Z}/p\mathbb{Z}$ (for primes P_4, P_8, P_{16}, P_{32}, P_{64}, and P_{128}) in sequential and parallel mode. We skip the case of $N = K^3$ for P_{128} ($K = 256$) as it is too large to fit in the memory of either of our compute nodes. All measurements are completed on `Intel-i7-7700K`. Table 3.5 provides similar comparisons measured on `Xeon-X5650`. In the case of `Xeon-X5650`, we observe that with more cores and threads, our parallel GFPF implementation gains more speedup compared to the parallel GMP implementation. For both the serial and parallel cases, we find our implementation using GFPF multiplication is faster than GMP in most cases.

Table 3.4: The running-time (in milliseconds) and ratio ($t_{\text{GFPF}}/t_{\text{GMP}}$) of serial and parallel computation of FFT on vectors of size $N = K^e$ over $\mathbb{Z}/p\mathbb{Z}$ for P_4, P_8, P_{16}, P_{32}, P_{64}, and P_{128} (measured on `Intel-i7-7700K`).

prime	k	K	e	Serial			Parallel		
				GFPF	GMP	$\frac{t_{\text{GFPF}}}{t_{\text{GMP}}}$	GFPF	GMP	$\frac{t_{\text{GFPF}}}{t_{\text{GMP}}}$
P_4	4	8	2	0.019	0.030	0.63**x**	0.057	0.118	0.48**x**
P_4	4	8	3	0.314	0.363	0.86**x**	0.215	0.276	0.77**x**
P_8	8	16	2	0.181	0.202	0.89**x**	0.117	0.143	0.81**x**
P_8	8	16	3	5.771	5.486	1.05**x**	1.603	2.247	0.71**x**
P_{16}	16	32	2	1.644	1.730	0.95**x**	0.513	0.693	0.74**x**
P_{16}	16	32	3	103.423	104.620	0.98**x**	24.052	35.017	0.68**x**
P_{32}	32	64	2	14.815	20.341	0.72**x**	3.507	5.411	0.64**x**
P_{32}	32	64	3	1922.373	2431.867	0.79**x**	462.746	702.163	0.65**x**
P_{64}	64	128	2	140.995	278.188	0.50**x**	33.507	69.879	0.47**x**
P_{128}	128	256	2	580.961	3745.353	0.15**x**	154.064	905.799	0.17**x**

Table 3.5: The running-time (in miliseconds) and ratio ($t_{\text{GFPF}}/t_{\text{GMP}}$) of serial and parallel computation of FFT on vectors of size $N = K^e$ over $\mathbb{Z}/p\mathbb{Z}$ for P_4, P_8, P_{16}, P_{32}, P_{64}, and P_{128} (measured on `Xeon-X5650`).

prime	k	K	e	Serial			Parallel		
				GFPF	GMP	$\frac{t_{\text{GFPF}}}{t_{\text{GMP}}}$	GFPF	GMP	$\frac{t_{\text{GFPF}}}{t_{\text{GMP}}}$
P_4	4	8	2	0.051	0.071	0.71**x**	0.155	0.114	1.35**x**
P_4	4	8	3	0.843	0.917	0.91**x**	0.452	0.577	0.78**x**
P_8	8	16	2	0.472	0.546	0.86**x**	0.217	0.320	0.67**x**
P_8	8	16	3	16.661	15.231	1.09**x**	2.837	4.806	0.59**x**
P_{16}	16	32	2	4.444	5.085	0.87**x**	0.877	1.371	0.63**x**
P_{16}	16	32	3	284.080	297.904	0.95**x**	41.012	66.635	0.61**x**
P_{32}	32	64	2	39.809	64.307	0.61**x**	5.701	11.640	0.48**x**
P_{32}	32	64	3	4674.079	6501.669	0.71**x**	696.311	1289.061	0.54**x**
P_{64}	64	128	2	376.450	909.041	0.41**x**	53.578	140.610	0.38**x**
P_{128}	128	256	2	1395.310	13371.369	0.10**x**	240.362	1811.282	0.13**x**

3.5.4 Performance analysis of FFT implementations

In this section we compare the parallel speedup factors for each of GFPF and GMP approaches compared to their corresponding serial implementations. From the previous section and Table 3.2, we know that the GFPF multiplication of two arbitrary elements in $\mathbb{Z}/p\mathbb{Z}$ is slower than the GMP implementation. At the same time, Table 3.4 and 3.5 indicate that computing FFT on large vectors over $\mathbb{Z}/p\mathbb{Z}$ using GFPF multiplication turns out to be faster than GMP arithmetic in most cases, for both serial and parallel modes. This interesting result can be explained as follows.

When we compute DFT using generalized Fermat prime field arithmetic (including GFPF multiplication), the majority of the multiplications are performed in the base-cases, which are carried out in linear time through cyclic shift (a sequence of data movement, subtraction, and carry handling; see Section 3.2). Meanwhile, in the case of GMP arithmetic, all of the multiplications are done using the same function calls, with no consideration for the cheap multiplications in the base-case DFT_K.

Figure 3.1 presents the ratio of time spent in one modular multiplication operation in FFT over $\mathbb{Z}/p\mathbb{Z}$ on vectors of size $N = K^e$ between the GFPF implementation and the GMP arithmetic. We see that for the GFPF implementation the average time spent in one modular multiplication is much lower than the time spent in the same operation using GMP arithmetic.

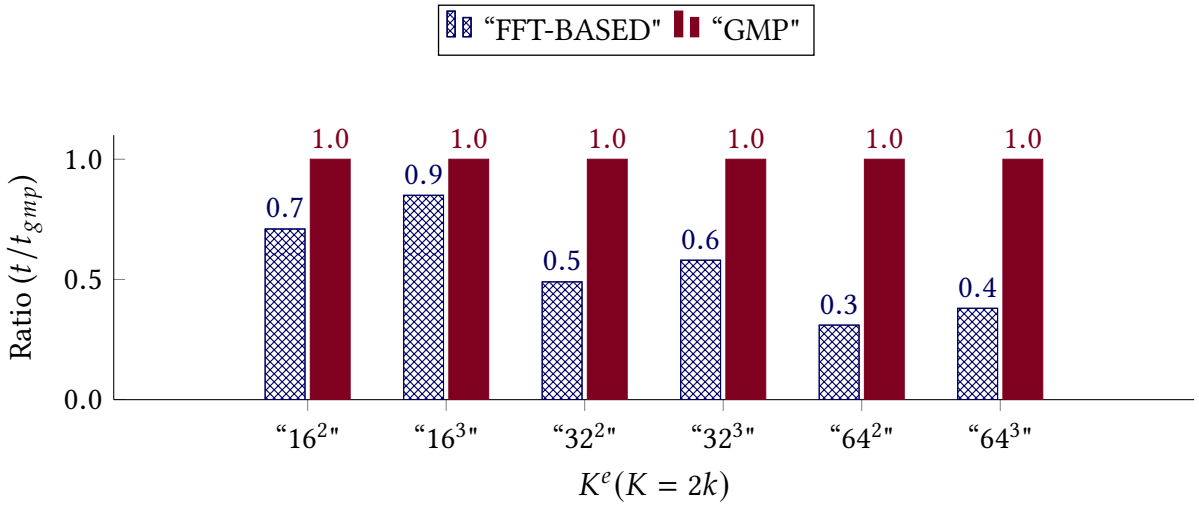

Figure 3.1: Ratio (t/t_{gmp}) of average time spent in one multiplication operation measured during computation of FFT over $\mathbb{Z}/p\mathbb{Z}$ on vectors of size $N = K^e$.

This result agrees with our estimation of increased performance due to Fürer's trick [21]. As it is demonstrated, by using cyclic shift for performing cheap multiplications in the base-case, we can lower the average time spent in multiplications, resulting in faster computation of the base-case DFT_K's, and consequently, speed up the computation of the whole FFT over $\mathbb{Z}/p\mathbb{Z}$.

Now, we take a closer look at the steps involved in the DFT computation. Table 3.6 provides the running-time data for every step of computing a DFT of size $N = K^3$ ($K = 64$)

over $\mathbb{Z}/p\mathbb{Z}$ for prime P_{32}. The timings are measured for both implementations on `Intel-i7-7700K`. As we observe, for both implementations in the serial mode, the time spent in precomputation and stride permutation is negligible compared to the time spent in twiddle multiplications and the base-case DFT_K's. Also, parallelization has little impact on precomputation and stride permutation. In contrast, parallelization significantly improves the time spent in twiddle multiplications and the base-case DFT_K's for both approaches.

Finally, Table 3.7 comapres the parallel speedup ratios for each implemenentation on both `Intel-i7-7700K` and `Xeon-X5650`. This table indicates that the parallelization of the GFPF implementation appears to be slightly more successful than the parallelization of the GMP implementation. This difference in the performance can be attributed to, by our measurements, the sharp management of computing resources (i.e. specialized arithmetic and minimal usage of memory). We have repeated the same benchmarks with hyper-threading disabled; this is also shown in Table 3.7. With hyper-threading disabled the speedup drops slightly, nevertheless, both implementations still gain nearly linear parellel speedup.

Table 3.6: Time spent (milliseconds) in different steps of serial and parallel computation of DFT of size $N = K^3$ over $\mathbb{Z}/p\mathbb{Z}$, for prime P_{32} ($K = 2k = 64$) measured on `Intel-i7-7700K`.

Mode	Variant	Precomputation	Permutation	DFT_K	Twiddle
Serial	GFPF	14 (ms)	72 (ms)	444 (ms)	1406 (ms)
	GMP	6 (ms)	177 (ms)	1229 (ms)	1026 (ms)
Parallel	GFPF	14 (ms)	51 (ms)	82 (ms)	330 (ms)
	GMP	6 (ms)	181 (ms)	284 (ms)	237 (ms)

Table 3.7: Ratio ($t_{serial}/t_{parallel}$) for serial vs. parallel execution of each implementation for $N = K^e$ ($K = 2k$, $e = 3$) measured on both `Intel-i7-7700K` and `Xeon-X5650` with and without hyper-threading enabled.

k	`Intel-i7-7700K`				`Xeon-X5650`			
	+ Hyper-threading		- Hyper-threading		+ Hyper-threading		- Hyper-threading	
	GFPF	GMP	GFPF	GMP	GFPF	GMP	GFPF	GMP
4	1.37**x**	1.25**x**	1.57**x**	1.31**x**	1.87**x**	1.59**x**	2.35**x**	1.95**x**
8	3.64**x**	2.44**x**	3.36**x**	2.34**x**	5.52**x**	3.17**x**	4.99**x**	3.40**x**
16	4.31**x**	2.96**x**	3.77**x**	2.69**x**	6.93**x**	4.47**x**	5.66**x**	4.16**x**
32	4.15**x**	3.48**x**	3.67**x**	3.07**x**	6.71**x**	5.04**x**	5.65**x**	4.47**x**

3.6 Conclusions and future work

We have presented an implementation of Fast Fourier Transforms over generalized Fermat prime fields on multi-threaded processors. Our parallel implementations using both specialized arithmetic and integer arithmetic from the GMP library achieve nearly linear parallel speedup. We noticed that the parallelization of our specialized implementation is slightly more successful than our GMP implementation. We attribute this higher performance to reduced number of arithmetic instructions due to using specialized arithmetic, minimal memory usage, and unrolling base-case DFT's and hard coding the constants.

Our results prove that developing specialized arithmetic (e.g. Montgomery multiplication, Barret reduction, cyclic shift introduced in Section 3.2 and using inline assembly) can be beneficial. Doing so leads to reduced overhead compared to a more generic implementation such as large integer arithmetic functions available in GMP, or other libraries on top of GMP. Unrolling the base-case DFT's improves the performance for two main reasons. First, by removing the majority of permutations (all except the last swap), it minimizes data movement. Second, compared to a naive implementation, a hard coded base-case reduces the number of multiplications by a power of radix to less than half (by simply avoiding the multiplications by 1 in the first place). Designing our implementation based on the iterative six step FFT algorithm was crucial; it allowed for more a finely scheduled parallelization on multi-core CPUs which obtains good speedup.

As part of our future work we should extend our implementation to arbitrary vector sizes, that is, the cases where the size N is not in the form K^e. Also, we must consider how to apply our approach to very large input sizes, for example, when the input vectors are too large to fit into main memory. Finally, we need to address another bottleneck of the current implementation, that is, the arbitrary multiplication in the generalized Fermat prime fields. We need a better solution for the multiplication between two polynomials with 64-bit integer coefficients; indeed, such a multiplication can result in coefficeints up to 192 bits, requiring multi-precision arithmetic.

4 KLARAPTOR: A Tool for Dynamically Finding Optimal Kernel Launch Parameters Targeting CUDA Programs

4.1 Introduction

Programming for high-performance parallel computing is a notoriously difficult task. Programmers must be conscious of many factors impacting performance including scheduling, synchronization, and data locality. Of course, program code itself impacts the program's performance, however, there are still further *parameters* which are independent from the code and greatly influence performance. For parallel programs three types of parameters influence performance: (i) *data parameters*, such as input data and its size; (ii) *hardware parameters*, such as cache capacity and number of available registers; and (iii) *program parameters*, such as granularity of tasks and the quantities that characterize how tasks are mapped to processors (e.g. dimension sizes of a thread block for a CUDA kernel).

Data and hardware parameters are independent from program parameters and are determined by the needs of the user and available hardware resources. Program parameters, however, are intimately related to data and hardware parameters. The choice of program parameters can largely influence the performance of a parallel program, resulting in orders of magnitude difference in timings (see Section 4.7). Therefore, it is crucial to determine values of program parameters that yield the best program performance for a given set of hardware and data parameter values.

In the CUDA programming model the kernel launch parameters, and thus the size and shape of thread blocks, greatly impact performance. This should be obvious considering that the memory accesses pattern of threads in a thread block can depend on the block's dimension sizes. The same could be said about multithreaded programs on CPU where parallel performance depends on task granularity and number of threads. Our general technique (see Section 4.4) is applied on top of some performance model to estimate program parameters which optimize performance. This could be applied to parallel programs in general, where performance models using program parameters exist. However, we dedicate this chapter to the discussion of GPU programs written in CUDA.

An important consequence of the impact of kernel launch parameters on performance is that an optimal thread block format (that is, dimension sizes) for one GPU architecture may not be optimal for another, as illustrated in [49]. This emphasizes not only the impact of hardware parameters on program parameters, but also the need for performance portability. That is to say, enabling users to efficiently execute the same parallel program on different architectures that belong to the same hardware platform.

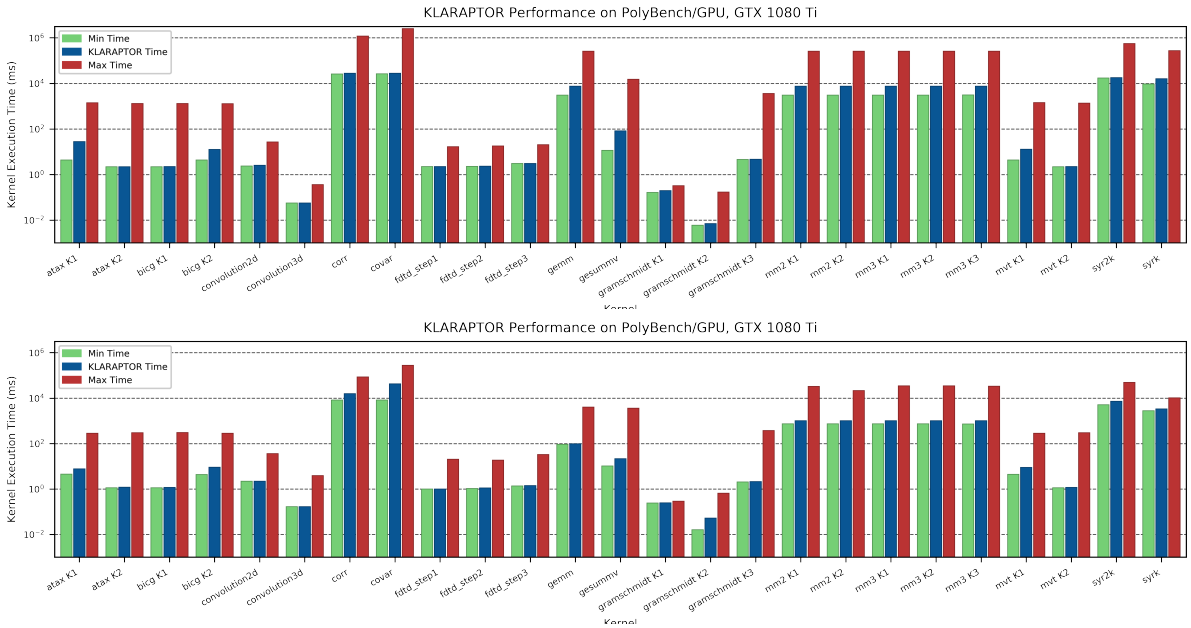

Figure 4.1: Comparing kernel execution time (log-scaled) for the thread block configuration chosen by KLARAPTOR versus the minimum and maximum times as determined by an exhaustive search over all possible configurations. Kernels are part of the PolyBench/GPU benchmark suite and executed on (1) a GTX 1080Ti with a data size of $N = 8192$ (except convolution3d with $N = 1024$), and (2) a GTX 760M with a data size of $N = 2048$ (except convolution3d with $N = 512$ and gemm with $N = 1024$).

In this chapter, we describe the development of KLARAPTOR (Kernel LAunch parameters RAtional Program estimaTOR), a tool for automatically and dynamically determining the values of CUDA kernel launch parameters which optimize the kernel's performance, for each kernel invocation independently. That is to say, based on the actual data and target device of a kernel invocation. The accuracy of KLARAPTOR's prediction is illustrated in Figure 4.1 where execution times are given for each kernel in the the PolyBench/GPU benchmark suite [50] on two different architectures. For each kernel, execution times are shown for three different thread block configurations: one chosen by KLARAPTOR, one resulting in the minimum time, and one resulting in the maximum time. The latter two are decided by an exhaustive search. In most cases, KLARAPTOR's prediction is very close to optimal; notice that the y-axis is log scaled. Further experimental results are reported in Section 4.7.

KLARAPTOR applies to CUDA a generic technique, also described herein in Section 4.4, to statically build a so-called *rational program* which is then used dynamically at runtime to

determine optimal program parameters for a given multithreaded program on specific data and hardware parameters. The key principle is based on an observation of most performance metrics. In most performance prediction models, *high-level performance metrics*, such as execution time, memory consumption, and hardware occupancy, can be seen as decision trees or flowcharts based on *low-level performance metrics*, such as memory bandwidth and cache miss rate. These low-level metrics are themselves piece-wise rational functions (PRFs) of program, data, and hardware parameters. This construction could be applied recursively to obtain a PRF for the high-level metric. We regard a computer program that computes such a PRF as a *rational program*, a technical notion defined in Section 4.2.

If one could determine these PRFs, then it would be possible to estimate, for example, the running time of a program based on its program, data, and hardware parameters. Unfortunately, exact formulas for low-level metrics are often not known, instead estimated through empirical measures or assumptions, or collected by profiling. This is a key challenge our technique addresses.

In most cases the values of the data parameters are only given at runtime, making it difficult to determine optimal values of the program parameters at an earlier stage. On another hand, a bad choice of program parameters can have drastic consequences. Hence, it is crucial to be able to determine the optimal program parameters at runtime without much overhead added to the program execution. This is precisely the intention of the approach proposed here.

4.1.1 Contributions

The goal of this work is to determine values of program parameters which optimize a multithreaded program's performance. Towards that goal, the method by which such values are found must be receptive to changing data and changing hardware parameters. Our contributions encapsulate this requirement through the dynamic use of a rational program. Our specific contributions include:

(i) a technique for devising a mathematical expression in the form of a *rational program* to evaluate a performance metric from a set of program and data parameters;
(ii) KLARAPTOR, a tool implementing the rational program technique to dynamically optimize CUDA kernels by choosing optimal launch parameters; and
(iii) an empirical and comprehensive evaluation of our tool on kernels from the `Polybench/GPU` benchmark suite.

4.1.2 Related works

The *Parallel Random Access Machine* (PRAM) model [51, 52], including PRAM models tailored to GPU code analysis such as TMM [53] and MCM [54] analyze the performance of parallel programs at an abstract level. More detailed GPU performance models are proposed such as MWP-CWP [55, 56], which estimates the execution time of GPU kernels based on the profiling

information of the kernels.

In the context of improving CUDA program performance, other research groups have used techniques such as loop transformation [57], auto-tuning [58, 59, 60, 61], dynamic instrumentation [62], or a combination of the latter two [63]. Auto-tuning techniques have achieved great results in projects such as ATLAS [8], FFTW [6], and SPIRAL [64] in which multiple kernel versions are generated *off-line* and then applied and refined *on-line* once the runtime parameters are known.

Although much research has been devoted to compiler optimizations for kernel source code or PTX code, previous works such as [65] and [49] suggest that kernel launch parameters (i.e. thread block configurations) have a large impact on performance and must be considered as a target for optimization. In [66], the authors present an input-adaptive GPU code optimization framework G-ADAPT, which uses statistical learning to find a relation between the input sizes and the thread block sizes. At linking time, the framework predicts the best block size for a given input size using the linear model obtained from compile time. This approach only considers the total size of the thread blocks and not their configuration. Meanwhile, the authors of [60] use a linear regression model to predict optimal thread block configurations (that is, dimension sizes and not just the total size). However, they assume kernel execution time scales linearly with data size. The authors in [67] have also developed a method determining the best thread block configuration, but similarly, they assume execution time scales linearly with data size. In [68], machine learning techniques are used in combination with auto-tuning to search for optimal configurations of OpenCL kernels, but their examples are limited to stencil computations.

4.1.3 Structure of the chapter

The remainder of this chapter is organized as follows. Section 4.2 formalizes and exemplifies the notion of rational programs and their relation to piece-wise rational functions and performance prediction. Section 4.3 gives on overview of the KLARAPTOR tool which applies our technique to CUDA kernels. The general algorithm underlying our tool, that is, building and using a rational program to predict program performance, is given in Section 4.4. Section 4.5 shows our specialization of that algorithm to CUDA programs. Our implementation is detailed in Section 4.6, while our implementation is evaluated in Section 4.7. Lastly, we draw conclusions and explore future work in Section 4.8.

4.2 Theoretical foundations

Let $\mathcal{P}$ be a multithreaded program to be executed on a targeted multiprocessor. Parameters influencing its performance include (i) *data parameters*, describing size and structure of the data; (ii) *hardware parameters*, describing hardware resources and their capabilities; and (iii) *program parameters*, characterizing parallel aspects of the program (e.g. how tasks are

mapped to hardware resources).

By fixing the target architecture, the hardware parameters, say, $\boldsymbol{H} = (H_1, \ldots, H_h)$ become fixed and we can assume that the performance of $\mathcal{P}$ depends only on data parameters $\boldsymbol{D} = (D_1, \ldots, D_d)$ and program parameters $\boldsymbol{P} = \big(P_1, \ldots, P_p\big)$. Moreover, an optimal choice of $\boldsymbol{P}$ naturally depends on a specific choice of $\boldsymbol{D}$. For example, in programs targeting GPUs the parameters $\boldsymbol{D}$ are typically dimension sizes of data structures, like arrays, while $\boldsymbol{P}$ typically specifies the formats of thread blocks.

Let $\mathcal{E}$ be a high-level performance metric for $\mathcal{P}$ that we want to optimize. More precisely, given the values of the data parameters $\boldsymbol{D}$, the goal is to find values of the program parameters $\boldsymbol{P}$ such that the execution of $\mathcal{P}$ optimizes $\mathcal{E}$. Performance prediction models attempt to estimate $\mathcal{E}$ from a combination of $\boldsymbol{P}$, $\boldsymbol{D}$, $\boldsymbol{H}$, and some model- or platform-specific low-level metrics $\boldsymbol{L} = (L_1, \ldots, L_\ell)$. It is natural to assume that these low-level performance metrics are themselves combinations of $\boldsymbol{P}$, $\boldsymbol{D}$, $\boldsymbol{H}$. This is an obvious observation from models based on PRAM such as TMM [53] and MCM [54].

Therefore, we look to obtain values for these low- and high-level metrics given values for program, and data parameters. To address our goal, we compute a mathematical expression for each metric, parameterized by data and program parameters, in the format of a *rational program* at the compile-time of $\mathcal{P}$. At the runtime of $\mathcal{P}$, given the specific values of $\boldsymbol{D}$ and a choice of $\boldsymbol{P}$, we can evaluate the rational programs to obtain a value for each metric and thus $\mathcal{E}$. These values can be used to determine which choice of $\boldsymbol{P}$ optimizes overall program performance. This method is detailed in Section 4.4. We take this section to define the rational program itself.

One could view a rational program as simply a computer program evaluating some performance-predicting model. However, as we will see in the following sections, it is more than that. Specifically, the encoding of some model as a flow chart whose nodes can then be approximated as a rational function is a powerful idea which can be used to simplify models and extrapolate results.

4.2.1 Rational programs

Let $X_1, \ldots, X_n, Y$ be pairwise different variables[1]. Let $\mathcal{S}$ be a sequence of three-address code (TAC [69]) instructions such that the set of the variables

1. that occur in $\mathcal{S}$, and
2. are never assigned a value by an instruction of $\mathcal{S}$

is exactly $\{X_1, \ldots, X_n\}$.

Consider $n = 2$. The sequence $\mathcal{S}_1$ below satisfies the above property while the sequence $\mathcal{S}_2$ does not.

[1] Variables refer to both its mathematical meaning and programming language concept.

	$\mathcal{S}_1$	$\mathcal{S}_1$
1:	$Y := X_1 + X_2$	$Y := X_1 + X_2$
2:	$Z := X_1 - X_2$	$X_1 := X_1 - X_2$
3:	$Y := Y \times Z$	$Y := Y \times X_1$

Indeed, in $\mathcal{S}_2$ the variable X_1 is assigned.

Definition We say that the sequence $\mathcal{S}$ is *rational* if every arithmetic operation used in $\mathcal{S}$ is either an addition, a subtraction, a multiplication, or a comparison of integer numbers in either fixed or arbitrary precision. We say that the sequence $\mathcal{S}$ is a *rational program* in $X_1, \ldots, X_n$ evaluating Y if the following two conditions hold:

1. $\mathcal{S}$ is rational, and
2. after specializing each of $X_1, \ldots, X_n$ to some integer value in $\mathcal{S}$, the execution of the specialized sequence always terminates and the last executed instruction assigns an integer value to Y.

Consider again $n = 2$. The sequence $\mathcal{S}_3$ and $\mathcal{S}_4$ are rational programs in X_1, X_2 evaluating Y while the sequences $\mathcal{S}_5$ is not. In all examples, we use language constructs **if**, **goto** and **exit** with their standard meaning, instead of more formal three-address code.

	$\mathcal{S}_3$	$\mathcal{S}_4$	$\mathcal{S}_5$
1:	$Y := X_1 + X_2$	$Y := X_1$	$Y := X_1 + X_2$
2:	$Z := X_1 - X_2$	**if** $Y > 0$ **goto** 4	$Z := X_1 / X_2$
3:	$Y := Y \times Z$	$Y := -Y$	$Y := Y \times Z$
4:	**exit**	**exit**	**exit**

Indeed, the sequence $\mathcal{S}_5$ cannot be specialized at $X_2 = 0$, due to the division X_1/X_2.

It is worth noting that the above definition can easily be extended to include Euclidean division, the integer part operations floor and ceiling, and arithmetic over rational numbers. For Euclidean division one can write a rational program evaluating the quotient q of integer a by b, leaving the remainder r to be simply calculated as $a - qb$. Then, floor and ceiling can be computed via Euclidean division. Rational numbers and their associated arithmetic are easily implemented using only integer arithmetic. Therefore, by adding these operations to Definition 4.2.1, the class of rational programs does not change. We regard rational programs as such henceforth.

4.2.2 Rational programs as flowcharts

For any sequence $\mathcal{S}$ of computer program instructions, one can associate $\mathcal{S}$ with a *control flow graph* (CFG). In the CFG of $\mathcal{S}$, the nodes are the *basic blocks* of $\mathcal{S}$. Recall that a *flowchart*

is another graphic representation of a sequence of computer program instructions. In fact, CFGs can be seen as particular flowcharts.

If, in a given flowchart C, every arithmetic operation occurring in every (process or decision) node is either an addition, subtraction, multiplication, or comparison of integers in either fixed or arbitrary precision, then C is the flowchart of a rational sequence of computer program instructions. Therefore, it is meaningful to depict rational programs using flowcharts, and vice versa, flowcharts as rational programs. For example, one could consider the metric of theoretical *hardware occupancy* as defined by NVIDIA. The following example details its definition, its depiction as a flowchart, as well as its dependency on program, data, and hardware parameters.

Example Hardware occupancy, *as defined in the* CUDA *programming model, is a measure of a program's effectiveness in using the Streaming Multiprocessors (SMs) of a GPU. Theoretical occupancy is calculated from a number of hardware parameters, namely:*

- *the maximum number* $R_{\max}$ *of registers per thread block,*
- *the maximum number* $Z_{\max}$ *of shared memory words per thread block,*
- *the maximum number* $T_{\max}$ *of threads per thread block,*
- *the maximum number* $B_{\max}$ *of thread blocks per SM and*
- *the maximum number* $W_{\max}$ *of warps per SM,*

as well as low-level kernel-dependent performance metrics, namely:

- *the number* R *of registers used per thread and*
- *the number* Z *of shared memory words used per thread block,*

and a program parameter, namely the number T *of threads per thread block. The occupancy of a* CUDA *kernel is defined as the ratio between the number of active warps per SM and the maximum number of warps per SM, namely* $W_{\text{active}}/W_{\max}$*, where*

$$W_{\text{active}} \leq \min\left(\lfloor B_{\text{active}} T/32 \rfloor, W_{\max}\right) \tag{4.1}$$

and B_{active} *is given by the flowchart in Figure 4.2. This flowchart shows how one can derive a rational program computing* B_{active} *from* $R_{\max}$, $Z_{\max}$, $T_{\max}$, $B_{\max}$, $W_{\max}$, R, Z, T. *It follows from formula (4.1) that* W_{active} *can also be computed by a rational program from* $R_{\max}$, $Z_{\max}$, $T_{\max}$, $B_{\max}$, $W_{\max}$, R, Z, T. *Finally, the same is true for theoretical occupancy of a* CUDA *kernel using* W_{active} *and* $W_{\max}$.

4.2.3 Piece-wise rational functions in rational programs

We begin with an observation describing the fact that a rational program can be viewed as a piece-wise rational function [2] .

[2]Here, rational function is in the sense of algebra, see Section 4.6.4.

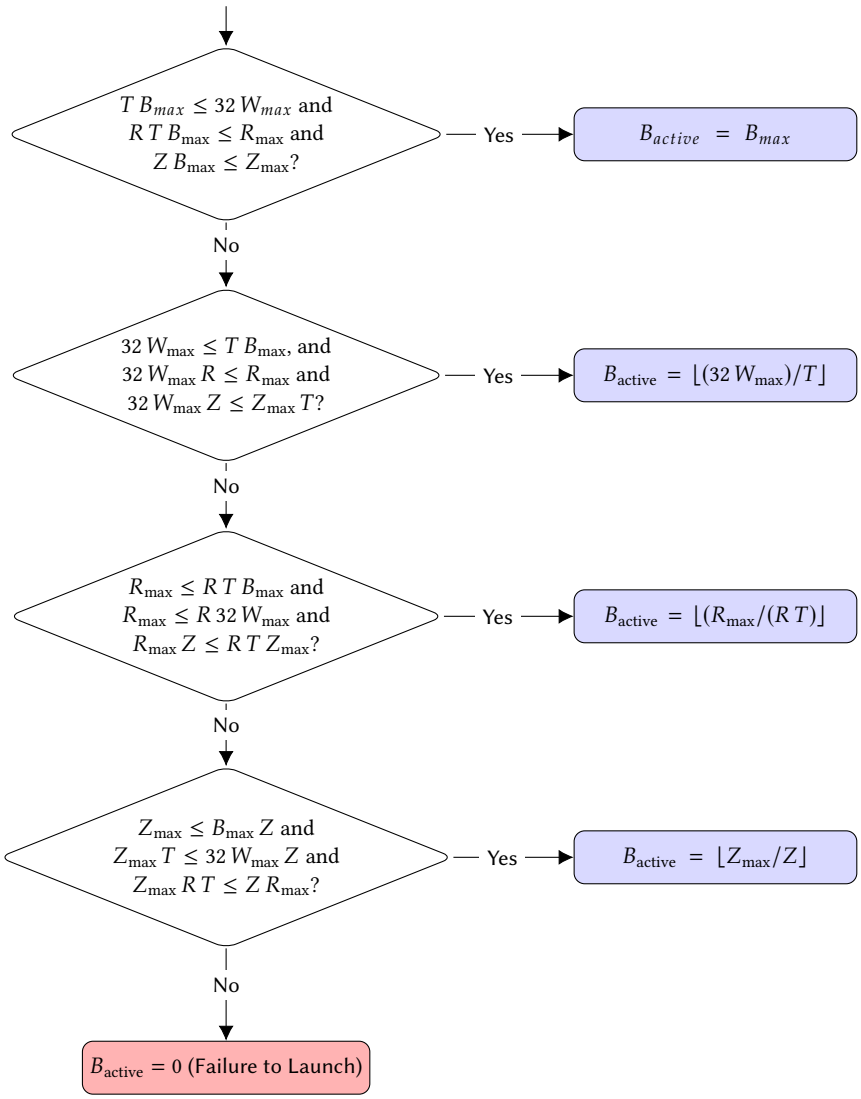

Figure 4.2: Rational program (presented as a flow chart) for the calculation of active blocks in CUDA.

Observation 1 Let $\mathcal{S}$ be a rational program in $X_1, \ldots, X_n$ evaluating Y. Let s be any instruction of $\mathcal{S}$ other than a branch or an integer part instruction. Hence, this instruction can be of the form $C = -A$, $C = A + B$, $C = A - B$, $C = A \times B$, where A and B can be any rational number. Let $V_1, \ldots, V_v$ be the variables that are defined at the entry point of the basic block of the instruction s. An elementary proof by induction yields the following fact. There exists a rational function in $V_1, \ldots, V_v$ that we denote by $f_s(V_1, \ldots, V_v)$ such that $C = f_s(V_1, \ldots, V_v)$ for all possible values of $V_1, \ldots, V_v$. From there, one derives the following observation. There exists a partition $\mathcal{T} = \{T_1, T_2, \ldots\}$ of $\mathbb{Q}^n$ (where $\mathbb{Q}$ denotes the field of rational numbers) and rational functions $f_1(X_1, \ldots, X_n)$, $f_2(X_1, \ldots, X_n)$, ... such that, if $X_1, \ldots, X_n$ receive respectively the values $x_1, \ldots, x_n$, then the value of Y returned by $\mathcal{S}$ is one of $f_i(x_1, \ldots, x_n)$ where i is such that $(x_1, \ldots, x_n) \in T_i$ holds. In other words, $\mathcal{S}$ computes Y as a *piece-wise rational function* (PRF). Notice that, trivially, if $\mathcal{S}$ contains only one basic block, then $\mathcal{S}$ can be given by a single rational function.

Example 4.2.2 shows that the hardware occupancy of a CUDA kernel is given as a piece-wise rational function in the variables $R_{\max}$, $Z_{\max}$, $T_{\max}$, $B_{\max}$, $W_{\max}$, R, Z, T. Hence, in this example, we have $n = 8$, and, as shown by Figure 4.2, its partition of $\mathbb{Q}^n$ contains 5 parts as there are 5 terminating nodes in the flowchart.

Suppose that a flowchart C representing the rational program $\mathcal{R}$ is partially known; to be precise, suppose that the decision nodes are known (that is, mathematical expressions defining them are known) while the process nodes are not. Then, from Observation 1, each process node can be given by a series of one or more rational functions. Trivially, a single formula can also be seen as a flowchart with a single process node. Determining each of those rational functions can be achieved by solving an *interpolation* or *curve fitting* problem. More generally, if the sequence of instructions in a process node involves non-rational functions (e.g. log) we can apply Stone-Weierstrass Theorem [70] to approximate each of those by a PRF.

It then follows that any performance metric, which can be depicted as a flow chart or a formula, can also be represented as a piece-wise rational function, and thus a rational program. For high-level performance metrics, which rely on low-level metrics, one could work recursively, first determining rational programs for the low-level metrics which depend on $\boldsymbol{P}$, $\boldsymbol{D}$, and $\boldsymbol{H}$, and then constructing a rational program for the high-level metric from those rational programs. Hence, by this recursive construction, we can fully determine a rational program for a high-level metric depending only on $\boldsymbol{P}$, $\boldsymbol{D}$, and $\boldsymbol{H}$. Of course, hardware parameters could be fixed given a target architecture to yield a rational program which depends only on $\boldsymbol{P}$ and $\boldsymbol{D}$. Again, notice that even where formulas for low-level metrics are not known, it is still possible to estimate them as PRFs, and thus rational programs, via a curve fitting.

As an example, consider occupancy (Example 4.2.2). One could first determine PRFs for the number of registers user per thread and the amount of shared memory used per thread block. Then, a PRF is determined for the number of active blocks (Figure 4.2) from these two low-level metrics, and a few more hardware and program parameters. Thus, by recursive construction, we have a PRF depending only on program and hardware parameters.

Lastly, we make one final remark. We assumed that the decision nodes in the flowchart of the rational program were known, however, we could relax this assumption. Indeed, each decision node is given by a series of rational functions. Hence, those could also be determined by solving curve fitting problems. However, we do not discuss this further since it is not needed in our proposed technique or implementation presented in the remainder of this chapter.

4.3 KLARAPTOR: a dynamic optimization tool for CUDA

The theory of rational programs is put into practice for the CUDA programming model by our tool KLARAPTOR. KLARAPTOR is a compile-time tool implemented using the LLVM Pass Framework and the MWP-CWP performance model to dynamically choose a CUDA kernel's launch parameters (thread block configuration) which optimize its performance. Most high-performance computing applications require computations be as fast as possible and so kernel performance is simply measured as its execution time.

As mentioned in Section 4.1, thread block configurations drastically affect the running time of a kernel. Determining optimal thread block configurations typically follows some heuristics, for example, constraining block size to be a multiple of 32 [71]. However, it is known that the dimension sizes of a thread block, not only its total size, affect performance [49, 65]. Moreover, since thread block configurations are intimately tied to the size of data being operated on, it is very unlikely that a static thread block configuration optimizes the performance of all data sizes. Our tool effectively uses rational programs to dynamically determine the thread block configuration which minimizes the execution time of a particular kernel invocation, considering the invocation's particular data size and target architecture. This is achieved in two main steps.

1. At the compile-time of a CUDA program, its kernels are analyzed in order to build rational programs estimating some performance metrics for each individual kernel. Each rational program, written as code in the C language, is inserted into the code of the CUDA program so that it is called before the execution of the corresponding kernel.
2. At runtime, immediately preceding the launch of a kernel, where data parameters have specific values, the rational program is evaluated to determine the thread block configuration which optimizes the performance of the kernel. The kernel is then launched using this thread block configuration.

Not only are we concerned with kernel performance, but also programmer performance. By that, we mean the efficiency of a programmer to produce optimal code. When a programmer is attempting to optimize a kernel, choosing optimal launch parameters can either be completely ignored, performed heuristically, determined by trial and error, or determined by an exhaustive search. The latter two options quickly become infeasible as data sizes grow large. Regardless, any choice of optimal thread block configuration is likely to optimize only a single data size.

For KLARAPTOR to be practical, not only does the choice of optimal kernel launch parameters need to be correct, but its two main steps must also be performed in an manner more efficient than trial and error or exhaustive search. Namely, the compile-time analysis cannot add too much overhead to the the compilation time and the runtime decision of the kernel launch parameters cannot overwhelm the program execution time. For the former, our analysis is performed quickly by analyzing kernel performance on only small data sizes, and then results are extrapolated. The time for this process typically ranges between 30 seconds and 2 minutes (see Section 4.7). For the later, the rational program evaluation is quick and simple, being only the evaluation of a few rational functions. Moreover, we maintain a runtime invocation history to instantly provide results for future kernel launches. Our implementation is detailed in Section 4.6.

We have made use of the `Polybench/GPU` benchmark suite as an empirical evaluation of the correctness of our tool on a range of CUDA programs. In Figure 4.1 we have already seen that KLARAPTOR accurately predicts the optimal or near-optimal thread block configuration. Before presenting more detailed results and experimentation in Section 4.7, we describe the steps followed by our tool to build and use rational programs for determining a thread block configuration which optimizes performance.

4.4 An algorithm to build and deploy rational programs

In this section the notations and hypotheses are the same as in Section 4.2. Namely, $\mathcal{E}$ is a high-level performance metric for the multithreaded program $\mathcal{P}$, $\boldsymbol{L}$ is a set of low-level metrics of size ℓ, and $\boldsymbol{P}$, $\boldsymbol{D}$, $\boldsymbol{H}$ are sets of program, data, and hardware parameters, respectively. Recall $\boldsymbol{P}$ has size p. Let us assume that the values of $\boldsymbol{H}$ are known at the compile-time of $\mathcal{P}$ while the values of $\boldsymbol{D}$ are known at runtime. Further, let us assume that $\boldsymbol{P}$ and $\boldsymbol{D}$ take integer values. Hence the values of $\boldsymbol{P}$ belong to a finite set $F \subset \mathbb{Z}^p$. That is to say, the possible values of $\boldsymbol{P}$ are tuples of the form $(\pi_1, \ldots, \pi_p) \in F$, with each π_i being an integer. Let us call such a tuple a *configuration* of the program parameters. Due to the nature of program parameters, those are not necessarily all independent variables For example, in CUDA, the product of the dimension sizes of a thread block is usually a multiple of the warp size (32).

Given a performance-prediction model for $\mathcal{E}$, one could work recursively to determine a single rational program $\mathcal{R}$, depending on only $\boldsymbol{D}$ and $\boldsymbol{P}$, evaluating $\mathcal{E}$, from a combination of rational programs constructed for each low-level metric in $\boldsymbol{L}$ and values of $\boldsymbol{D}$ and $\boldsymbol{P}$. Following Section 4.2.3, each of these rational programs are constructed by computing rational functions. Without loss of generality, let us assume each low-level metric is given by a single formula and thus a single rational function. Hence, we look to determine $g_1(\boldsymbol{D}, \boldsymbol{P})$, ..., $g_\ell(\boldsymbol{D}, \boldsymbol{P})$ for the ℓ low-level metrics. Finally, at runtime, given particular values of $\boldsymbol{D}$, the rational program for $\mathcal{E}$ can be evaluated for various values of $\boldsymbol{P}$ to determine the optimal. In the context of CUDA where we look to optimize execution time, the selection of program parameters leading to optimal performance is a complex task. We leave the discussion of that

to Section 4.5. In the remainder of this section we describe the general process to build and use rational programs to determine optimal configurations. The entire process is decomposed into six steps: the first three occur at compile-time and the next three at runtime.

1. **Data collection**: To perform a curve fitting of the rational functions $g_1(\boldsymbol{D}, \boldsymbol{P})$, ..., $g_\ell(\boldsymbol{D}, \boldsymbol{P})$ we require data points to fit. These are collected by (i) selecting a subset of K points from the space of possible values of $(\boldsymbol{D}, \boldsymbol{P})$; and (ii) executing the program $\mathcal{P}$, recording the values of the low-level performance metrics $\boldsymbol{L}$ as $\boldsymbol{V} = (V_1, \ldots, V_\ell)$, at each point in K. Data used for executing the programs is generated randomly, but could follow some scheme provided by the user.
2. **Rational function approximation**: For each low-level metric L_i we use the set of points K and the corresponding value V_i measured for each point to approximate the rational function $g_i(\boldsymbol{D}, \boldsymbol{P})$. We observe that if these values were known exactly the rational function $g_i(\boldsymbol{D}, \boldsymbol{P})$ could be determined exactly. In practice, however, these empirical values are likely to be noisy from profiling, and/or numerical approximations. Consequently, we actually determine a rational function $\hat{g}_i(\boldsymbol{D}, \boldsymbol{P})$ which approximates $g_i(\boldsymbol{D}, \boldsymbol{P})$.
3. **Code generation**: In order to generate the rational program $\mathcal{R}$, we proceed as follows:
   (i) we convert the rational program representing $\mathcal{E}$ into code, essentially encoding the CFG for computing $\mathcal{E}$;
   (ii) we convert each $\hat{g}_i(\boldsymbol{D}, \boldsymbol{P})$ into code, specifically sub-routines, estimating L_i; and
   (iii) we include those sub-routines into the code computing $\mathcal{E}$, which yields the desired rational program $\mathcal{R}$, fully constructed, and depending only on $\boldsymbol{D}$ and $\boldsymbol{P}$.
4. **Rational program evaluation**: At the runtime of $\mathcal{P}$, the data parameters $\boldsymbol{D}$ are given particular values. For those specified values of $\boldsymbol{D}$ and for all practically meaningful values of $\boldsymbol{P}$ from the set F,[3] we compute an estimate of $\mathcal{E}$ using $\mathcal{R}$. The evaluation of $\mathcal{E}$ over so many different possible program parameters is feasible for three reasons:
   (i) the number of program parameters is small, typically $p \leq 3$, see Section 4.6;
   (ii) the set of meaningful values for $\boldsymbol{P}$ is small (consider that in CUDA the product of thread block dimension sizes should be a multiple of 32 less than 1024); and
   (iii) the program $\mathcal{R}$ simply evaluates a few polynomial formulae and thus runs almost instantaneously.
5. **Selection of optimal values of program parameters**:
   When the search space of values of program parameters $\boldsymbol{P}$ is large, a numerical optimization technique is required for this step. But, as just explained, the total number of evaluations is quite small and thus an exhaustive search is feasible to determine an optimal configuration. However, it is possible that several configurations, up to some margin, optimize $\mathcal{E}$. In practice, one can refine results by comparing several near-optimal

[3] The values for $\boldsymbol{P}$ are likely to be constrained by the values $\boldsymbol{D}$. For example, if P_1, P_2 are the two dimension sizes of a two-dimensional thread block of a CUDA kernel operating on a square matrix of order D_1, then $P_1 P_2 \leq D_1^2$ is meaningful.

configurations by using secondary metrics, see Section 4.5 for the case of KLARAPTOR's implementation.

6. **Program execution**: Once an optimal configuration is selected, the program $\mathcal{P}$ can finally be executed using this configuration along with the values of $\boldsymbol{D}$.

4.5 Runtime selection of thread block configuration for a CUDA kernel

In the previous section we examined the general six-step process to build and use a rational program to select program parameters. We now look to specialize this process to CUDA and describe precisely the algorithm followed by KLARAPTOR to select program parameters (i.e. thread block configurations) in its attempt to minimize kernel execution time (steps 4 and 5 in the general six-step process).

Here, our high-level performance metric $\mathcal{E}$ is execution time, and we make use of the MWP-CWP model [55, 56] to estimate it. In this model the execution time is estimated as the estimated number of clock cycles E_CC. From the previous six-step process, it seems trivial to simply select the configuration which minimizes E_CC. However, in some practical examples, this can lead to a very poor choice of thread block configuration which does not minimize execution time. While the MWP-CWP model is a quite comprehensive performance prediction model, it mainly relies on occupancy for estimating the number of clock cycles. This has a serious drawback; the calculated occupancy is an upper bound which can be too optimistic and is not a strong predictor of performance for compute-intensive kernels (see [72] and "Help" sheet of NVIDIA occupancy calculator spreadsheet, under "Notes on occupancy" [73]). Therefore, we still make use of MWP-CWP but do not take E_CC directly as the performance predictor.

Using the values of some low-level metrics (also defined within the MWP-CWP model), together with E_CC, we perform a case discussion in order to select a thread block configuration. At runtime, the particular value of our data parameters is known, The values for the low-level metrics are then obtained using the data parameter values and the set of practically meaningful thread block configurations by evaluating the previously obtained rational functions $\hat{g}_i(\boldsymbol{D}, \boldsymbol{P})$ for them. This yields a dictionary whose keys are configurations and values are a list of estimated performance metrics. The low-level metrics of interest are: (i) the average number of computational instructions per thread (`Comp_Inst`), (ii) the average number of memory instructions per thread (`Mem_Inst`), (iii) the average number of clock cycles spent on global memory transactions (`Mem_CC`), and (iv) the amount of dynamic shared memory used.We then also compute occupancy and E_CC for each configuration.

At this point, we look to define a strategy which chooses a subset of configurations where any in the set would give near-optimal performance. Our proposed strategy takes into account not only occupancy and E_CC, but also the arithmetic intensity and efficiency of global memory read/write transactions of a kernel. We define arithmetic intensity as the ratio of

Comp_Inst/Mem_Inst. The main idea is first to determine if the kernel is memory intensive or compute intensive. If the kernel is memory intensive, we look to minimize memory instructions, otherwise it is assumed to be compute intensive and we look to both minimize time spent on memory transactions and maximize computational instructions. The latter might increase the chance of exploiting instruction-level parallelism (ILP) by the hardware (however, this is not guaranteed to happen at runtime). This strategy is detailed in Algorithm 7.

Algorithm 7 HueristicSelection

```
1:  procedure HueristicSelection(Dictionary D of [key,value] pairs with block configurations as
    keys and values of performance metrics as values, AME as average memory efficiency (percentage)
    of the kernel)
2:      [X, nRepeat] ← [10, number of entries in D];
3:      D← Stabilize (D, "ArithmeticIntensity", X, nRepeat);
4:                                          ▷ Stabilize D w.r.t. values of Arithmetic Intensity.
5:      AAR ← Average of arithmetic intensity for entries of D.
6:      MET ← 25%                           ▷ Setting the memory efficiency threshold to 25%.
7:      if AME <= MET and AAR ≤ 1.0 then                                    ▷ Memory-intensive
8:          [X, nRepeat] ← [10, 10];
9:          D← Stabilize (D, "E_CC", X, nRepeat);
10:         [X, nRepeat] ← [10, 1];
11:         D← Stabilize (D, "Mem_Inst", X, nRepeat);
12:         D← Optimize (D, "MIN", "Mem_Inst", X, nRepeat);
13:         D← Stabilize (D, "Comp_Inst", X, nRepeat);
14:     else                                                            ▷ Compute-intensive
15:         [X, nRepeat] ← [25, 1];
16:         D← Stabilize (D, "Occupancy", X, nRepeat);
17:         [X, nRepeat] ← [10, 1];
18:         D← Optimize (D, "MIN", "Mem_CC", X, nRepeat);
19:         D← Optimize (D, "MAX", "Comp_Inst", X, nRepeat);
20:         D← Stabilize (D, "Mem_Inst", X, nRepeat);
21:     end if
22:     Return D;
23: end procedure
```

Certain subroutines are used within Algorithm 7 in an attempt to filter outliers and select candidate configurations. The first function, Stabilize, removes outliers by iteratively filtering out configurations. The iteartion proceeds until the standard deviation of the values of the target metric no longer changes, or a maximum number of iterations is reached. To remove outliers, configurations whose value of the chosen metric falls in the top or bottom X% of values are removed. Note that standard deviation should indeed eventually reach some fixed value since the metrics take finitely many values. The next function, Optimize, simply gets the subset of configurations whose value for a particular metric falls in the top X% (if "MAX" is specified) or the bottom X% (if "MIN" is specified).

To decide whether a kernel is memory intensive or not, first, using a profiler (through nvprof

or CUPTI), we measure the average value of global memory (load/store) efficiency for the kernel for a set of block configurations. This value, which we will refer to as "Average Memory Efficiency" (AME) should be less than a certain threshold. This threshold, which we will refer to as "Memory Efficiency Threshold" (MET) can be determined through empirical study. Currently, we set MET to 25%. Using AME together with the mean arithmetic intensity of all configurations, indicates if a kernel is memory intensive or not. In particular, the mean arithmetic intensity should be at most equal to 1, that is, for every memory instruction there is at most one computational instruction performed. Finally, we note that the values for X and `nRepeat` in Algorithm 7 have also been determined through experimentation (Lines 2, 8, 10, 16, and 18).

4.6 The implementation of KLARAPTOR

In this section we give an overview of the implementation of our previously presented technique (Sections 4.4 and 4.5) specialized to CUDA in the KLARAPTOR tool. Our tool is built in the C language, making use of the LLVM Pass Framework (see Section 4.6.2) and the NVIDIA CUPTI API (see Section 4.6.3). KLARAPTOR is freely available in source at https://github.com/orcca-uwo/KLARAPTOR.

In the case of a CUDA kernel, the data parameters specify the input data size. In many examples this is a single parameter, say N, describing the size of an array (or the order of a multi-dimensional array), the values of which are usually powers of 2. Program parameters describe the kernel launch parameters, i.e. grid and thread block dimension sizes, and are also typically powers of 2. For example, a possible thread block configuration may be $1024 \times 1 \times 1$ (a one-dimensional thread block), or $16 \times 16 \times 2$ (a three-dimensional thread block). Lastly, the hardware parameters are values specific to the target GPU, for example, memory bandwidth, the number of SMs available, and their clock frequency.

We organize this section as follows. Sections 4.6.1 and 4.6.2 are specific to our implementation and do not correspond to any step of Section 4.4. The compile time steps 1 (data collection) and 2 (rational function estimation) are reflected in Sections 4.6.3 and 4.6.4, respectively, while step 3 requires no explanation. The runtime steps 4 (rational program evaluation) and 6 (program execution) are trivial to perform, while step 5 (selection of optimal configuration) is clear from the discussion in Section 4.5. Throughout this section we refer to the notion of a *driver program* as the code, for each individual kernel, using a rational program to select a configuration.

4.6.1 Annotating and preprocessing source code

Beginning with a CUDA program written in C/C++, we minimally annotate the host code to make it compatible with our *pre-processor*. We take into account the following points:

(i) the code targets at least CUDA Compute Capability (CC) 3.x;
(ii) there should be no CUDA runtime API calls as such calls will interfere with later CUDA driver API calls used by our tool, for example, `cudaSetDevice`;
(iii) the block dimensions and grid dimensions must be declared as the typical CUDA `dim3` structs.

For each kernel in the CUDA code, we add two pragmas, one specifying the dimension of the kernel (1, 2, or 3), and one defining the index of the kernel input arguments corresponding to the data size N. As an example, consider the code segment below of a CUDA kernel and the associated pragmas. This kernel operates of a two-dimensional array of order N, making it a two-dimensional kernel.

```
#pragma kernel_info_size_param_idx_Sample = 1;
#pragma kernel_info_dim_sample_kernel = 2;
__global__ void Sample (int *A, int N) {
    int tid_x = threadIdx.x + blockIdx.x*blockDim.x;
    int tid_y = threadIdx.y + blockIdx.y*blockDim.y;
    ...
}
```

Lastly, for each kernel, the user must fill two formatted configuration files which follow `Python` syntax. One specifies the constraints on the thread block configuration while the other specifies the grid dimensions. For example, for the 2D kernel `Sample` above, one could specify that its thread block configuration (bx, by, bz) must satisfy $bx < by^2$, $bx < N$ and $by < N$. Since the kernel dimension is given as 2, we assume $bz = 1$. Similarly, the grid dimensions (gx, gy, gz), could be specified as $gx = \lceil \frac{N}{bx} \rceil$, $gy = \lceil \frac{N}{by} \rceil$, $gz = 1$.

Now, a preprocessor processes the annotated source code, replacing CUDA runtime API calls with driver API kernel launches. This step includes source code analysis in order to extract a list of kernels, a list of kernel calls in the host code, and finally, the body of each kernel to be used for further analysis. A so-called "PTX lookup table" is built to store kernel information and static parameters. This table will be inserted into the "instrumented binary", the compiled CUDA program augmented by the driver programs.

4.6.2 Input/Output builder

The Input/Output builder Pass, or IO-builder, is a compiler pass written in the LLVM Pass Framework to build the previously mentioned "instrumented binary". This pass embeds an IO mechanism (i.e. a function call) to communicate with the driver program of a kernel for each of its invocations. Thus, at the runtime of the CUDA program being analyzed (step 6 of Section 4.4), an IO function is called before each kernel invocation to return six integers, (gx, gy, gz, bx, by, bz), the optimal kernel launch parameters.

The IO-builder pass goes through the following steps:

(i) obtain the LLVM intermediate representation of the instrumented source code and find

all CUDA driver API kernel calls;

(ii) relying on the annotated information for each kernel, determine which variables in the IR contain the value of N for a corresponding kernel call; and

(iii) insert a call to an IO function immediately before each kernel call in order to pass the runtime value of N to the corresponding driver program and retrieve the optimal kernel launch parameters.

4.6.3 Building a driver program: data collection

In order to perform the eventual rational function approximation, we must collect data and statistics regarding certain performance counters and runtime metrics (see Section 4.5, [55] and [74]). These metrics can be partitioned into three categories.

Firstly, *architecture-specific performance counters* of a kernel, characteristics influenced by the CC of the target device. These can be obtained at compile-time, since the target CC is specified at this time. These characteristics include the number of registers used per thread, the amount of static shared memory per thread block, and the number of (arithmetic and memory) instructions per thread.

Secondly, *runtime-specific performance counters* that depend on the behavior of the kernel at runtime. This includes values impacted by memory access patterns, namely, the number of memory accesses per warp, the number of memory instructions of each thread, and the total number of warps that are being executed. We have developed a customized profiler using NVIDIA's EVENT API within the CUPTI API to collect the required runtime performance counters. The profiler is customized to collect only the required information, allowing it to be very lightweight and avoid the huge overheads of a typical profiler (e.g. NVIDIA's `nvprof` [4]).

Thirdly, *device-specific parameters*, which describe an actual GPU card, allow us to compute a more precise performance estimate. A subset of such parameters can be determined by microbenchmarking the device (see [75] and [76]), this includes the memory bandwidth, and the departure delay for memory accesses. The remaining parameters can easily obtained by consulting the vendor's guide [31], or by querying the device itself via the CUDA driver API. Such parameters include the number of SMs on the card, the clock frequency of SM cores, and the instruction delay.

4.6.4 Building a driver program: rational function approximation

Using the runtime data collected in the previous step, we look to determine the rational functions $\hat{g}_i(\boldsymbol{D}, \boldsymbol{P})$ (see Section 4.4) which estimate the low-level metrics or other intermediate values in the rational program. For ease of discussion, we replace the parameters $\boldsymbol{D}$ and $\boldsymbol{P}$ with the generic variables $X_1, \ldots, X_n$.

A rational function is simply a fraction of two polynomials:

$$f(X_1, \ldots, X_n) = \frac{\alpha_1 \cdot (X_1^0 \cdots X_n^0) \ + \ \ldots \ + \ \alpha_i \cdot (X_1^{u_1} \cdots X_n^{u_n})}{\beta_1 \cdot (X_1^0 \cdots X_n^0) \ + \ \ldots \ + \ \beta_j \cdot (X_1^{v_1} \cdots X_n^{v_n})} \tag{4.2}$$

With a *degree bound* (an upper limit on the exponent) on each variable X_k in the numerator and the denominator, u_k and v_k, respectively, these polynomials can be defined up to some *parameters* (using the language of parameter estimation), namely the coefficients of the polynomials, $\alpha_1, \ldots, \alpha_i$ and $\beta_1, \ldots, \beta_j$. Through algebraic analysis of performance models like the MWP-CWP model, and empirical evidence, these degree bounds are relatively small.

We perform a parameter estimation (for each rational function) on the coefficients $\alpha_1, \ldots, \alpha_i$, $\beta_1, \ldots, \beta_j$ to determine the rational function precisely. This is a simple linear regression which can be solved by an over-determined system of linear equations, say by the method of linear least squares. However, the system suffers from *multicollinearity* (see [77, Chapter 23]) and can become rank-deficient. Solving using the typical QR-factorization is impossible; hence we use the computationally more intensive yet more numerically stable method of *singular value decomposition* (SVD; for details see [78, Chapter 4]). Our implementation uses LAPACK [79] for SVD and the Basic Polynomial Algebra Subprograms (BPAS) library [43] for efficient rational function and polynomial implementations.

4.7 Experimentation

In this section we examine the performance of KLARAPTOR by applying it to the CUDA programs of the `Polybench/GPU` benchmark suite [58]. We note here that many of the kernels in this suite perform relatively low amounts of work; they are best suited to being executed many times from a loop in the host code. Data in this section was collected using a GTX 1080Ti.

Table 4.1 provides experimental data for the main kernels in the benchmark suite `Polybench/GPU`. Namely, this table compares the execution times of the thread block configuration chosen by KLARAPTOR against the optimal thread block configuration found though exhaustive search. The table shows a couple of data sizes in order to highlight that the best configuration can change for different input sizes. While it may appear for some examples that there are large variations between timings of the KLARAPTOR-chosen configuration and the optimal, these should be considered within the full range of possible configurations. Recall from Figure 4.1 that compared to the worst possible timings, the KLARAPTOR-chosen configuration and the optimal result in very similar in timings.

In Figure 4.3 we compare the time it takes KLARAPTOR to perform its compile-time analysis and build the rational programs for each example in the `PolyBench/GPU` suite. This is compared against determining the optimal thread block configuration by an exhaustive search. Since KLARAPTOR's compile-time analysis is a one-time occurrence which optimizes for all data sizes, exhaustive search times are given as a sum for data sizes up to $N = 8192$.

Table 4.1: KLARAPTOR vs. exhaustive search for thread block configuration choice for kernels in Polybench/GPU.

Kernel	N	KLARAPTOR Time (ms)	Chosen Config.	Optimal Time (ms)	Optimal Config.
atax K1	4096	2.35	32, 4	0.85	32, 1
	8192	27.83	1, 64	4.33	16, 2
atax K2	4096	1.09	16, 2	1.04	32, 1
	8192	2.20	32, 1	2.19	64, 1
bicg K1	4096	1.05	256, 1	1.05	32, 1
	8192	2.23	256, 1	2.21	64, 1
bicg K2	4096	1.15	8, 4	0.85	32, 1
	8192	12.58	256, 4	4.35	512, 1
convolution2d	4096	0.79	256, 1	0.77	32, 4
	8192	2.54	256, 4	2.35	32, 4
corr	4096	5700.65	256, 1	5075.77	32, 1
	8192	27846.91	256, 1	26024.94	32, 1
covar	4096	5682.96	256, 1	5076.77	32, 1
	8192	27865.89	256, 1	26182.65	32, 1
fdtd_step1	4096	0.56	256, 1	0.56	32, 2
	8192	2.22	256, 4	2.22	32, 4
fdtd_step2	4096	0.58	256, 1	0.58	512, 1
	8192	2.33	32, 16	2.30	512, 1
fdtd_step3	4096	0.77	256, 1	0.77	512, 2
	8192	3.06	256, 4	3.05	1024, 1
gemm	4096	723.29	256, 1	386.76	32, 32
	8192	7481.13	256, 1	3069.66	32, 16
gesummv	4096	8.19	2, 16	1.62	32, 1
	8192	82.21	32, 16	11.58	64, 1
gramschmidt K1	4096	0.09	4, 32	0.09	256, 1
	8192	0.20	8, 32	0.17	64, 1
gramschmidt K2	4096	0.01	32, 2	0.01	256, 1
	8192	0.01	512, 2	0.01	256, 1
gramschmidt K3	4096	2.15	256, 1	2.11	32, 1
	8192	4.68	256, 1	4.61	32, 1
mm2 K1	4096	695.23	256, 1	384.93	32, 32
	8192	7531.13	256, 1	3062.26	32, 16
mm2 K2	4096	761.49	256, 1	386.61	32, 32
	8192	7533.08	256, 1	3077.75	32, 16
mm3 K1	4096	749.27	256, 1	388.40	32, 32
	8192	7531.56	256, 1	3065.34	32, 16
mm3 K2	4096	816.08	256, 1	389.13	32, 16
	8192	7532.66	256, 1	3067.87	32, 16
mm3 K3	4096	737.21	256, 1	392.81	32, 16
	8192	7530.24	256, 1	3085.43	32, 16
mvt K1	4096	1.15	8, 4	0.86	32, 1
	8192	12.90	256, 4	4.35	16, 2
mvt K2	4096	1.05	256, 1	1.05	32, 1
	8192	2.23	256, 1	2.21	128, 1
syr2k	4096	7050.62	1, 64	2097.15	4, 32
	8192	18013.51	16, 64	17398.88	4, 8
syrk	4096	2973.88	2, 16	1165.24	16, 16
	8192	15936.21	32, 16	9368.56	16, 16

The best and worst execution times for the main kernel in each example (for $N = 8192$) is also given to highlight the fact that our optimization step is sometimes faster than even a single execution of a kernel with a poor choice of thread block configuration. We note that for some kernels, with very quick running times, exhaustive search is not a bad option. However, some examples such as GRAMSCHMIDT, take an exorbitant amount of time for exhaustive search. This figure also shows that the one-time compile-time cost of optimization can often be amortized by only a few executions of the kernel.

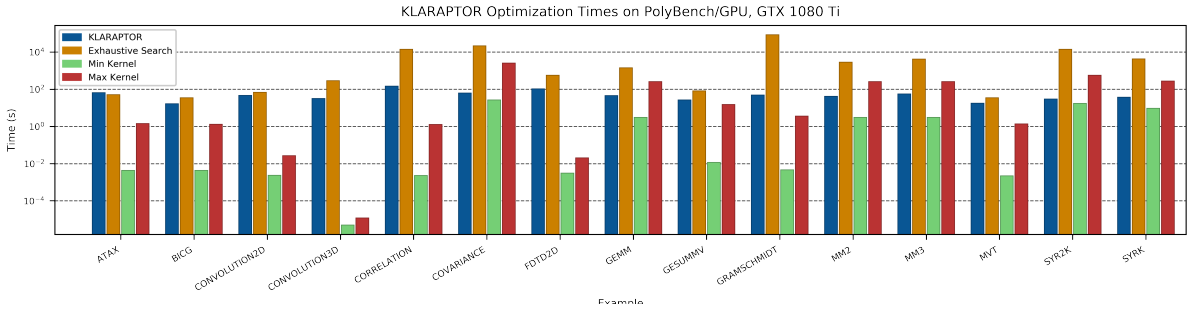

Figure 4.3: Comparing times (log-scaled) for (1) compile-time optimization steps of KLARAPTOR, (2) exhaustive search over all thread block configurations, the execution time for a kernel given (3) the best thread block configuration, and (4) the worst thread block configuration. Exhaustive search is given as a sum for values up to $N = 8192$ (except convolution3d with $N = 1024$).

4.8 Conclusions and future work

The performance of a single CUDA program can vary wildly depending on the target GPU device, the input data size, and the kernel launch parameters. Moreover, a thread block configuration yielding optimal performance for a particular data size or a particular target device will not necessarily be optimal for a different data size or different target device. In this chapter we have presented the KLARAPTOR tool for determining optimal CUDA thread block configurations for a target architecture, in a way which is adaptive to each kernel invocation and input data, allowing for dynamic data-dependent performance and portable performance. This tool is based upon our technique of encoding a performance prediction model as a rational program. The process of constructing such a rational program is a fast and automatic compile-time process which occurs simultaneously to compiling the CUDA program by use of the LLVM Pass framework. Our tool was tested using the kernels of the `Polybench/GPU` benchmark suite with great results.

However, one of our biggest challenges (see Section 4.5) is obtaining accurate values for occupancy. Recently, the author of [80] and [72] has suggested a GPU performance model relying on Little's law; it measures concurrency as a product of latency and throughput. This model considers both warp and instruction concurrency while previous models [31, 55, 56, 81] con-

sider only warp concurrency. The author's analysis of those models suggests their limitation is the significant underestimation of occupancy when arithmetic intensity (the number of arithmetic instructions per memory access) is intermediate. In future work we look to apply an improved performance prediction model in order to achieve even better results.

The algorithm presented in Section 4.4 is not specific to the problem of optimizing program parameters of CUDA kernels. This algorithm could also be used to help optimizing program parameters of multithreaded code targeting CPU or multi-process code targeting distributed systems. The main requirement is to have a mathematical model which expresses the metric to be optimized, in terms of quantities that can be measured, by means of a piece-wise rational function.

5 Arbitrary-precision Integer Multiplication on GPUs

5.1 Introduction

Arbitrary-precision integer arithmetic is driven by applications in solving systems of polynomial equations and cryptography. Such computations arise when high precision is required (that is, when values fit into multiple machine words), or, in order to avoid coefficient overflow due to intermediate expression swell.

Among the arithmetic operations, multiplication of large integers is especially important as it is at the core of many other algorithms. For the past few decades (beginning with Karatsuba-Offman in 1962 [82]), numerous algorithms for multiplying long integers have been proposed, each with various implementations and platform-specific optimizations. Multiplication has been the subject of active research, but recent results have mostly been focused improving the theoretical lower bound. Table 5.1 presents the complexity estimates for some of the algorithms used for multiplying two polynomials of degree less than k. Table 5.2 presents the complexity estimates for some of the algorithms used for multiplying two large integers with k digits. For more details see [83, 44, 23, 84, 19, 21, 41, 85, 86, 87].

Table 5.1: Comparison of algorithms for multiplying polynomials $f(x), g(x) \in R[x]$ of degree less than k.

Variant	Algorithm	$M(k)$
Non FFT-based	Classical (Toom-1)	$O(k^2)$
	Karatsuba-Ofman (Toom-2), 1962 [82]	$O(k^{\log_2^3}) = O(k^{1.58})$
	Toom-Cook (Toom-3), 1963 [83]	$O(k^{\log_3^5}) = O(k^{1.46})$
FFT-based	NTT/FFT multiplication [44, 23]	$O(\frac{9}{2}k\log(k))$

Meanwhile, the growing demand for faster computation alongside the recent advances in hardware technology have led to the development of a vast array of many-core and multi-core processors, programming models, and language extensions (e.g. CUDA and OpenCL for GPUs, and OpenMP and Cilk for multi-core CPUs). The massive computational power of

Table 5.2: Comparison of algorithms for multiplying k machine word integers.

Variant	Algorithm	$M(k)$
FFT-based	Fürer, 2009 [19, 21]	$O(k \log(k) 2^{O(\log^* k)}))$
	Harvey-van der Hoeven-Lecerf, 2015 [41]	$O(k \log(k) 2^{3\log^* k}))$
	Harvey-van der Hoeven, 2018 [85] (proved as an unconditional bound)	$O(k \log(k) 2^{2\log^* k}))$
	Covanov-Thomé, 2019 [86]	$O(k \log(k) 4^{\log^* k}))$
	Harvey-van der Hoeven, 2019 [87]	$O(k \log(k))$

parallel processors, especially GPUs, makes them viable targets for carrying out arbitrary-precision integer arithmetic.

Nevertheless, developing parallel algorithms, followed by implementing and optimizing them as multi-threaded parallel programs imposes a set of challenges. In this chapter we explain the current state of research on arbitrary-precision integer multiplication on CUDA-enabled GPUs and propose a parallel solution for this problem.

5.2 Essential definitions for designing parallel algorithms

We need the following definitions for design and analysis of parallel algorithms on many-core processors such as GPUs. See [88, 89] for more details.

- *Data parallelism*: executing the same operation or function across a large dataset.
- *Work*: The total time to execute a task on one processor, denoted as T_1.
- *Span*: Critical path in the DAG of a task, denoted as T_∞. In other words, it indicates the execution time on an infinite number of processors.
- *Parallelism*: The ratio of work to span, denoted as $\frac{T_1}{T_\infty}$.
- *Speedup*: For T_P, the execution time on P processors, speedup is computed as $\frac{T_1}{T_P}$.

5.3 Choice of algorithm for parallelization

At this point, we face three questions to answer.

1. Which algorithm to choose for a parallel implementation?
2. How to reduce the span and increase the degree of parallelism? In other words, at what level are the operations executed in a data-parallel fashion?
3. How to maximize the device utilization and avoid wasting clock cycles?

Improvements of the asymptotically fast algorithms such as the variants of FFT-based multiplication do not necessarily lead to practical outcomes for everyday computing due to the

fact that such algorithms need very large input sizes to be practically efficient. Besides that, even when computing with FFT-based solutions with large input sizes (such as [1] and [2]), we might still need to multiply large integers that are not as large as the input but still require multiple machine words. The multiplications of twiddle factors as part of a FFT is a significant example of this. This makes less efficient algorithms (in terms of algebraic complexity) favorable candidates for parallelization and efficient implementation.

In this work, we propose a new fine-grained parallel algorithm for multiplying arbitrary-precision integers of k digits.

5.4 Problem definition

Let X, Y be two vectors of N elements of ring R:

$$\vec{X} = (X_0, X_1, \ldots, X_{N-1}),$$
$$\vec{Y} = (Y_0, Y_1, \ldots, Y_{N-1}).$$

We are interested in a fast solution for pointwise multiplication of X by Y:

$$\vec{Z} = (X_0Y_0, X_1Y_1, \ldots, X_{N-1}Y_{N-1})$$

Let $\beta = 2^b$, where b is the size of a machine word, R is either $\mathbb{Z}/m\mathbb{Z}$ for $m = \beta^k$, or $\mathbb{Z}/p\mathbb{Z}$ for $p = r^k + 1$ where the radix r fits into a machine word. In other words, each element of R, which we will refer to as a "large integer", fits into k machine words and is represented as a vector $A = (a_0, a_1, \ldots, a_{k-1})$ with each digit $a_i < \beta$. The value of β depends on the architecture and is typically either $\beta = 2^{32}$ or $\beta = 2^{64}$. For the rest of this work, we assume $\beta = 2^{32}$, k is a multiple of 32, and N is a positive integer.

Note that we can encode $A, B \in R$ as polynomials $A(x) = a_0 + a_1x + \ldots + a_{k-1}x^{k-1}$ and $B(x) = b_0 + b_1x + \ldots + b_{k-1}x^{k-1}$ in $R[x]$. Then, we can compute $A \times B$ over R by evaluating $A(x).B(x)$ at $x = \beta$. In practice, evaluation at β does not help with the final result as it leads to a $2k$-digit integer that should eventually be represented in the base β. A more efficient solution is to to perform a sequence of carry-handling which we will refer to as an *ℓhc-normalization step*.

Now, we consider two possibilities for assigning work to threads that compute the pointwise multiplication of X by Y. Precisely, this decision will determine the level of data parallelism in our implementation in the following ways.

(i) Parallelization at the level of coefficients: Assume each $X_i.Y_i$ product is computed by exactly one thread. This approach could be efficient if k is not large enough (e.g. between 2 to 16 digits), or when an efficient parallel algorithm is not available.

(ii) Parallelization at the level of digits: This solution is based on parallel arithmetic, that is, the nontrivial process of parallelizing the operations inside a function. In this case, each

$X_i.Y_i$ product is computed by multiple threads. This approach requires some synchronization mechanism which can cause some overhead if handled carelessly. However, for large enough values of k (e.g., more than or equal to 32 digits) this can be a more efficient solution provided that we have a low span algorithm with a high degree of parallelism.

We keep in mind the trade-off between the two approaches with respect to the cost of scheduling threads, global memory communications, and synchronization overhead. In [1], we have implemented the first approach for twiddle factor multiplications. In this work, we study the second approach both theoretically and experimentally.

5.5 A fine-grained parallel multiplication algorithm

Algorithm 8 presents a naive algorithm for multiplication of two k-digit large integers.

Algorithm 8 Naive algorithm for computing the product of large k-digit integers A and B.

```
1:  input: Two k-digit integers A and B stored as vectors Ā and B̄.
2:  output: Large 2k-digit integer N as the product of A by B.
3:  procedure NaiveMult(Ā, B̄, k)
4:      M(M_0, M_1, ··· , M_{2k−1}) ← ([0,0,0],[0,0,0],··· ,[0,0,0])  ▷ Vector of 2k triple digits.
5:      for (i = 0; i < k; i = i + 1) do
6:          for (j = 0; j < k; j = j + 1) do
7:              M_(i+j) ← M_(i+j) + A_i B_j        ▷ Single-digit multiplication and triple-digit addition.
8:          end for
9:      end for
10:     N ← ConvertLHC(M, k)                        ▷ ℓhc normalization step.
11:     return N
12: end procedure
```

Algorithm 11 presents an algorithm for a carry handling procedure which we will refer to as ℓhc normalization.

Next, Algorithm 9 presents an improved version of Algorithm 8 that takes advantage of *Karatsuba intermediate product*s.

Karatsuba intermediate product: The Karatsuba intermediate product for machine word size digits a_i, a_j, b_i, b_j is defined as follows:

$$a_i b_j + a_j b_i = (a_i - a_j)(b_j - b_i) + z_i + z_j$$

where $z_i = a_i b_i$ and $z_j = a_j b_j$. There are two main ideas at the core of Algorithm 9:

(i) we precompute the values of $z_i = a_i b_i$ for $0 \leq i < k$, and
(ii) the sequence of for-loops I, II, and III are well-structured for parallel execution.

Note that in Algorithm 9, there are no dependencies between any two iterations in Loops I and III, this indicates that they are well-suited for parallel execution, and in particular, a GPU implementation. However, Loop II needs a chunking strategy for efficient usage of the global memory. As the chunking strategy, we divide the vector of coefficients for each of the operands A and B into chunks of size s. For the sake of simplicity, we assume k is a multiple of s and s is a multiple of 32.

Rewriting Loop II of Algorithm 9 with a chunking strategy leads to Algorithm 10. We have two nested loops over `SingleCM` and `DoubleCM` routines, which are shown in Algorithm 12 and 13, respectively. Procedure `M5Mult` in Algorithm 10 is well-suited for parallel implementation, meanwhile, as we will show in the next section Algorithm 10 has the same work complexity as Algorithm 9.

Algorithm 9 Sequential algorithm for computing the product of large k-digit integers A and B while taking advantage of Karatsuba intermediate products.

1: **input:** Two k-digit integers A and B stored as vectors $\vec{A}$ and $\vec{B}$.
2: **output:** Large $2k$-digit integer N as the product of A by B.
3: **procedure** `SequentialMult`$(\vec{A}, \vec{B}, k)$
4: $\quad M(M_0, M_1, \cdots, M_{2k-1}) \leftarrow ([0,0,0],[0,0,0],\cdots,[0,0,0])$ ▷ Vector of $2k$ triple digits.
5: $\quad Z(Z_0, Z_1, \cdots, Z_{k-1}) \leftarrow ([0,0],[0,0],\cdots,[0,0])$ ▷ Vector of k double digits.
6: $\quad$ **for** $(i = 0;\ i < k;\ i = i + 1)$ **do** ▷ Loop I
7: $\quad\quad Z_i \leftarrow A_i B_i$ ▷ Single-digit multiplication.
8: $\quad$ **end for**
9: $\quad$ **for** $(i = 0;\ i < k;\ i = i + 1)$ **do** ▷ Loop II
10: $\quad\quad$ **for** $(j = i + 1;\ j < k;\ j = j + 1)$ **do**
11: $\quad\quad\quad T \leftarrow (A_i - A_j)(B_j - B_i) + Z_i + Z_j$ ▷ Computing Karatsuba intermediate product.
12: $\quad\quad\quad M_{(i+j)} \leftarrow M_{(i+j)} + T$ ▷ One triple-digit addition.
13: $\quad\quad$ **end for**
14: $\quad$ **end for**
15: $\quad$ **for** $(i = 0;\ i < k;\ i = i + 1)$ **do** ▷ Loop III
16: $\quad\quad M_{2i} \leftarrow M_{2i} + Z_i$ ▷ Triple-digit addition.
17: $\quad$ **end for**
18: $\quad N \leftarrow$ `ConvertLHC`(M, k) ▷ ℓhc normalization step.
19: $\quad$ **return** N
20: **end procedure**

5.6 Complexity analysis

For simplifying the complexity analysis, we use Table 5.3 which presents the relative cost of arithmetic operations used in the analysis. In this table, $T_{\mathbf{A1}}$ is considered as the base-case. Note that $\eta = \frac{T_{\mathbf{M1}}}{T_{\mathbf{A1}}}$ is defined as a measure for comparing the cost of a single-digit multiplication to a single-digit addition.

Algorithm 10 Chunk-based algorithm for computing the product of large k-digit integers A and B while taking advantage of Karatsuba intermediate products.

input: Two k-digit integers A and B stored as vectors $\vec{A}$ and $\vec{B}$.
output: Large $2k$-digit integer N as the product of A by B.
procedure `M5Mult`$(\vec{A}, \vec{B}, k)$
    $\mathtt{M}(\mathtt{M}_0, \mathtt{M}_1, \cdots, \mathtt{M}_{2k-1}) \leftarrow ([0,0,0],[0,0,0],\cdots,[0,0,0])$ ▷ Vector of $2k$ triple digits.
    $\mathtt{Z}(\mathtt{Z}_0, \mathtt{Z}_1, \cdots, \mathtt{Z}_{k-1}) \leftarrow ([0,0],[0,0],\cdots,[0,0])$ ▷ Vector of k double digits.
    for $(\mathtt{i} = 0;\ \mathtt{i} < \mathtt{k};\ \mathtt{i} = \mathtt{i} + 1)$ **do** ▷ Loop I can be executed in parallel.
        $\mathtt{Z}_\mathtt{i} \leftarrow \mathtt{A}_\mathtt{i}\mathtt{B}_\mathtt{i}$
    end for
    $\lambda \leftarrow \frac{\mathtt{k}}{\mathtt{s}}$ ▷ Number of chunks.
    for $(\mathtt{c}_\mathtt{i} = 0;\ \mathtt{c}_\mathtt{i} < \lambda;\ \mathtt{c}_\mathtt{i} = \mathtt{c}_\mathtt{i} + 1)$ **do** ▷ Can be executed in parallel.
        $\mathtt{M} \leftarrow$ `SingleCM`$(\mathtt{M}, \mathtt{A}, \mathtt{B}, \mathtt{Z}, \mathtt{c}_\mathtt{i}, \mathtt{s})$
    end for
    for $(\mathtt{c}_\mathtt{i} = 0;\ \mathtt{c}_\mathtt{i} < \lambda;\ \mathtt{c}_\mathtt{i} = \mathtt{c}_\mathtt{i} + 1)$ **do** ▷ This loop CANNOT be executed in parallel.
        for $(\mathtt{c}_\mathtt{j} = \mathtt{c}_\mathtt{i} + 1;\ \mathtt{c}_\mathtt{j} < \lambda;\ \mathtt{c}_\mathtt{j} = \mathtt{c}_\mathtt{j} + 1)$ **do** ▷ Can be executed in parallel.
            $\mathtt{M} \leftarrow$ `DoubleCM`$(\mathtt{M}, \mathtt{A}, \mathtt{B}, \mathtt{Z}, \mathtt{c}_\mathtt{i}, \mathtt{c}_\mathtt{j}, \mathtt{s})$
        end for
    end for
    for $(\mathtt{i} = 0;\ \mathtt{i} < \mathtt{k};\ \mathtt{i} = \mathtt{i} + 1)$ **do** ▷ Loop III can be executed in parallel.
        $\mathtt{M}_{2\mathtt{i}} \leftarrow \mathtt{M}_{2\mathtt{i}} + \mathtt{Z}_\mathtt{i}$
    end for
    $\mathtt{N} \leftarrow$ `ConvertLHC`$(\mathtt{M}, \mathtt{k})$ ▷ ℓhc normalization step.
    return N
end procedure

Table 5.3: Relative cost of arithmetic operations.

Operation	Cost	Definition
A1	$T_{\mathbf{A1}}$ (Platform-dependent)	Adding two single-digit unsigned integers.
M1	$\eta T_{\mathbf{A1}}$ for $\eta \geq 1$	Multiplying two single-digit unsigned integers.
S1	$T_{\mathbf{A1}}$	Subtracting two single-digit unsigned integers.
A3	$3T_{\mathbf{A1}}$	Adding two triple-digit unsigned integers with carries.
M2	$2T_{S1} + T_{\mathbf{M1}} + 2T_{\mathbf{A3}}$	Computing a Karatsuba intermediate product.

Let $\lambda = \dfrac{k}{s}$ be the number of chunks. We begin with the work-complexity analysis of `NaiveMult`; it has k^2 iterations, each iteration computing a single-digit multiplication and a triple-digit addition (except the first iteration):

$$T_1^{\texttt{NaiveMult}}(k) = k^2 T_{\mathbf{M1}} + (k-1)^2 T_{\mathbf{A3}} \tag{5.1}$$

Then, we compute the work for Loops I, II, and III of `SequentialMult` as follows. Loop I

has k iterations, each iteration computing a single-digit multiplication. Loop II has $\frac{k(k-1)}{2}$ iterations, each computing a Karatsuba intermediate product and a triple-digit addition. Finally, Loop III has k iterations, each iteration computing a triple-digit addition. To summarize:

$$T_1^{\mathrm{I}}(k) = k.T_{\mathbf{M1}} \tag{5.2}$$

$$T_1^{\mathrm{II}}(k) = \frac{k(k-1)}{2}(T_{\mathbf{M2}} + T_{\mathbf{A3}}) \tag{5.3}$$

$$T_1^{\mathrm{III}}(k) = k.T_{\mathbf{A3}} \tag{5.4}$$

In total, the work for `SequentialMult` is:

$$\begin{aligned} T_1^{\texttt{SequentialMult}}(k) &= T_1^{\mathrm{I}}(k) + T_1^{\mathrm{II}}(k) + T_1^{\mathrm{III}}(k) \\ &= k.T_{\mathbf{M1}} + \frac{k(k+1)}{2}T_{\mathbf{A3}} + \frac{k(k-1)}{2}T_{\mathbf{M2}} \end{aligned} \tag{5.5}$$

Rewriting (5.1) and (5.5) in terms of single-digit additions and multiplications, then simplifying them solely with respect to single-digit addition we have:

$$T_1^{\texttt{NaiveMult}}(k) = k^2 T_{\mathbf{M1}} + 3(k-1)^2 T_{\mathbf{A1}} = (\eta k^2 + 3(k-1)^2)T_{\mathbf{A1}}$$

$$T_1^{\texttt{SequentialMult}}(k) = \frac{k(k+1)}{2}T_{\mathbf{M1}} + \frac{11k^2 - 5k}{2}T_{\mathbf{A1}} = \frac{\eta k^2 + 11k^2 + \eta k - 5k}{2}T_{\mathbf{A1}}$$

At this step, we compare `SequentialMult` and `M5Mult`. First, we should mention that `SingleCM` is a special case of `DoubleCM` that has the advantage of cutting the number of multiplications in half.

For chunk size s, the execution of `SingleCM` leads to computation of $\frac{s.(s-1)}{2}$ Karatsuba intermediate products whereas the execution of `DoubleCM` results in computing s^2 Karatsuba intermediate products. With that in mind, we prove that together `SingleCM` and `DoubleCM` are computationally as expensive as Loop II:

$$T_1^{\texttt{SingleCM}}(k) = \frac{s.(s-1)}{2}(T_{\mathbf{M2}} + T_{\mathbf{A3}}) \tag{5.6}$$

$$T_1^{\texttt{DoubleCM}}(k) = s^2(T_{\mathbf{M2}} + T_{\mathbf{A3}}) \tag{5.7}$$

$$\begin{aligned} T_1^{\texttt{SingleCM}}(k) + T_1^{\texttt{DoubleCM}}(k) &= \lambda T_1^{\texttt{SingleCM}}(k) + \frac{\lambda(\lambda-1)}{2}T_1^{\texttt{DoubleCM}}(k) \\ &= \frac{\lambda s(\lambda s - 1)}{2}(T_{\mathbf{M2}} + T_{\mathbf{A3}}) \end{aligned} \tag{5.8}$$

Note that $\lambda s = k$, then:

$$T_1^{\texttt{SingleCM}}(k) + T_1^{\texttt{DoubleCM}}(k) = \frac{k(k-1)}{2}(T_{\mathbf{M2}} + T_{\mathbf{A3}}) = T_1^{\mathrm{II}}(k) \tag{5.9}$$

Algorithm 11 Algorithm for applying ℓhc normalization on a vector $\vec{M}$ of $2k$ triple digits.

input: Vector $\vec{M}$ of $2k$ triple digits.
output: Large $2k$-digit integer N as the result of ℓhc normalization.
procedure ConvertLHC($\vec{M}, k$)
    $\mathrm{N}(\mathrm{N}_0, \mathrm{N}_1, \cdots, \mathrm{N}_{2k-1}) \leftarrow (0, 0, \cdots, 0)$ ▷ Vector of $2k$ digits.
    $\mathrm{L}(\mathrm{L}_0, \mathrm{L}_1, \cdots, \mathrm{L}_{2k-1}) \leftarrow (\mathrm{M}_0[0], \mathrm{M}_1[0], \cdots, \mathrm{M}_{2k-1}[0])$ ▷ Vector of $2k$ digits.
    $\mathrm{H}(\mathrm{H}_0, \mathrm{H}_1, \cdots, \mathrm{H}_{2k-1}) \leftarrow (\mathrm{M}_0[1], \mathrm{M}_1[1], \cdots, \mathrm{M}_{2k-1}[1])$ ▷ Vector of $2k$ digits.
    $\mathrm{C}(\mathrm{C}_0, \mathrm{C}_1, \cdots, \mathrm{C}_{2k-1}) \leftarrow (\mathrm{M}_0[2], \mathrm{M}_1[2], \cdots, \mathrm{M}_{2k-1}[2])$ ▷ Vector of $2k$ digits.
    for (i = 2k − 1; i ≥ 1; i = i − 1) **do** ▷ Shift elements of H with stride of 1.
        $\mathrm{H}_i \leftarrow \mathrm{H}_{i-1}$
    end for
    $\mathrm{H}_0 \leftarrow 0$ ▷ Set the first element of H equal to zero.
    for (i = 2k − 1; i ≥ 2; i = i − 1) **do** ▷ Shift elements of C with stride of 2.
        $\mathrm{C}_i \leftarrow \mathrm{C}_{i-2}$
    end for
    $\mathrm{C}_0 \leftarrow 0, \mathrm{C}_1 \leftarrow 0$ ▷ Set the first two elements of C equal to zero.
    carry = 0
    for (i = 0; i < 2k; i = i + 1) **do**
        $(\mathrm{N}_i, \mathrm{t}) \leftarrow \mathrm{L}_i + \mathrm{H}_i + \mathrm{C}_i +$ carry
        carry ← t
    end for
    return N
end procedure

Therefore, the work estimates for both M5Mult and SequentialMult are equal:

$$\begin{aligned} T_1^{\texttt{M5Mult}}(k) &= T_1^{\mathrm{I}}(k) + T_1^{\texttt{SingleCM}}(k) + T_1^{\texttt{DoubleCM}}(k) + T_1^{\mathrm{III}}(k) \\ &= T_1^{\mathrm{I}}(k) + T_1^{\mathrm{II}}(k) + T_1^{\mathrm{III}}(k) \\ &= T_1^{\texttt{SequentialMult}}(k) \end{aligned} \tag{5.10}$$

Recall that in the previous section we claimed M5Mult is well-structured for parallel execution. Indeed, the algorithm leads to a low estimated span, and therefore, high degree of parallelism:

$$\begin{aligned} T_\infty^{\mathrm{I}}(k) &= T_{\mathbf{M1}} \\ T_\infty^{\texttt{SingleCM}}(k) &= \frac{s(s-1)}{2}(T_{\mathbf{M2}} + T_{\mathbf{A3}}) \\ T_\infty^{\texttt{DoubleCM}}(k) &= \lambda s^2 (T_{\mathbf{M2}} + T_{\mathbf{A3}}) \\ T_\infty^{\mathrm{III}}(k) &= T_{\mathbf{A3}} \end{aligned}$$

Algorithm 12 The algorithm for computing product of chunk c from both A and B.

input:

- Vector $\vec{M}$ of $2k$ triple digits for storing results.
- Two k-digit integers A and B stored as vectors $\vec{A}$ and $\vec{B}$.
- Vector $\vec{Z}$ storing precomputed products.
- Chunk index c and chunk size s.

output:

- Vector $\vec{M}$ of $2k$ triple digits with updated results.

procedure `SingleCM`($\vec{M}, \vec{A}, \vec{B}, \vec{Z}, c, s$)
  for (d = 1; d ≤ $\frac{s}{2}$; d = d + 1) **do**
    for (i = 0; i < $\frac{s}{2}$; i = i + 1) **do**
      for (t = 0; t < 2; t = t + 1) **do**
        if d == $\frac{s}{2}$ and t == 1 **then**
          continue
        end if
        j ← i + d
        i′ ← i + t * $\frac{s}{2}$
        j′ ← j + t * $\frac{s}{2}$
        if s ≤ i′ or s ≤ j′ **then**
          [i′, j′] ← [j′, i′] ▷ Swap values of i′ and j′.
        end if
        i′ ← i′ + c * s
        j′ ← j′ + c * s
        T ← $(A_{i'} - A_{j'})(B_{j'} - B_{i'}) + Z_{i'} + Z_{j'}$
        ▷ Computing Karatsuba intermediate product.
        $M_{(i'+j')} \leftarrow M_{(i'+j')} + T$ ▷ One triple-digit addition.
      end for
    end for
  end for
  return M
end procedure

Collectively, the parallelism of `M5Mult` amounts to:

$$
\begin{aligned}
T_{\infty}^{\texttt{M5Mult}}(k) &= T_{\infty}^{\texttt{I}}(k) + T_{\infty}^{\texttt{SingleCM}}(k) + T_{\infty}^{\texttt{DoubleCM}}(k) + T_{\infty}^{\texttt{III}}(k) \\
&= T_{\mathbf{M1}} + \frac{s(s-1)}{2}(T_{\mathbf{M2}} + T_{\mathbf{A3}}) + (\lambda s^2)(T_{\mathbf{M2}} + T_{\mathbf{A3}}) + T_{\mathbf{A3}} \\
&= T_{\mathbf{M1}} + \frac{((2\lambda+1)s^2 - s)}{2}(T_{\mathbf{M2}} + T_{\mathbf{A3}}) + T_{\mathbf{A3}} \\
&= (\frac{((2\lambda+1)s^2 - s)}{2} + 1)T_{\mathbf{M1}} + (11\frac{((2\lambda+1)s^2 - s)}{2} + 3)T_{\mathbf{A1}} \\
&= \Big(\frac{(\eta+11)(2\lambda s^2 + s^2 - s)}{2} + (\eta+3)\Big)T_{\mathbf{A1}} = \Big(\frac{(\eta+11)(2ks + s^2 - s)}{2} + (\eta+3)\Big)T_{\mathbf{A1}}
\end{aligned}
$$

Algorithm 13 The algorithm for computing product of chunks c_i and c' from both A and B.

input:

- Vector $\vec{M}$ of $2k$ triple digits for storing results.
- Two k-digit integers A and B stored as vectors $\vec{A}$ and $\vec{B}$.
- Vector $\vec{Z}$ storing precomputed products.
- Chunk indices c and c', and chunk size s.

output:

- Vector $\vec{M}$ of $2k$ triple digits with updated results.

procedure DoubleCM($\vec{M}, \vec{A}, \vec{B}, \vec{Z}, c, c', s$)
  for $0 \leq \mathtt{i} < \mathtt{s}$ **do**
    for $0 \leq \mathtt{j} < \mathtt{s}$ **do**
      $\mathtt{i'} \leftarrow \mathtt{i} + \mathtt{c} * \mathtt{s}$
      $\mathtt{j'} \leftarrow \mathtt{j} + \mathtt{c'} * \mathtt{s}$
      $\mathtt{T} \leftarrow (\mathtt{A'_i} - \mathtt{A'_j})(\mathtt{B'_j} - \mathtt{B'_i}) + \mathtt{Z_{i'}} + \mathtt{Z_{j'}}$ ▷ Computing Karatsuba intermediate product.
      $\mathtt{M_{(i'+j')}} \leftarrow \mathtt{M_{(i'+j')}} + \mathtt{T}$ ▷ One triple-digit addition.
    end for
  end for
  return M
end procedure

To conclude, even though both M5Mult and SequentialMult have the same cost in terms of arithmetic operations, however, M5Mult is well-structured to be parallelized and has a high degree of parallelism. As it is shown in Figure 5.1, for all the values of $2^5 \leq k \leq 2^{14}$ and $1 \leq \eta \leq 5$, $T_1^{\texttt{SequentialMult}}(k)$ is usually less than $T_1^{\texttt{NaiveMult}}(k)$. Obviously, this makes M5Mult unsuitable for a sequential implementation. On the other hand, we expect that an efficient parallel implementation of M5Mult on GPUs will lead to high degree of parallelism. In fact, we can estimate the degree of parallelism by using the complexity estimates of the previous section for different values of s, k, and η. For example, Figure 5.2 presents the estimates for degree of parallelism for various values of k with $s = 32$ for $\eta = 5$.

5.7 Experimentation

In this section, we present our experimental results for studying the performance of M5Mult.

First, we compare the performance of our GPU implementation of M5Mult against another GPU implementation of schoolbook multiplication algorithm. The work in [90], up to our knowledge, is the only other parallelization of schoolbook multiplication; the CUDA implementation of this algorithm, which we will refer to as OptMult, is part of CUMODP library[90].

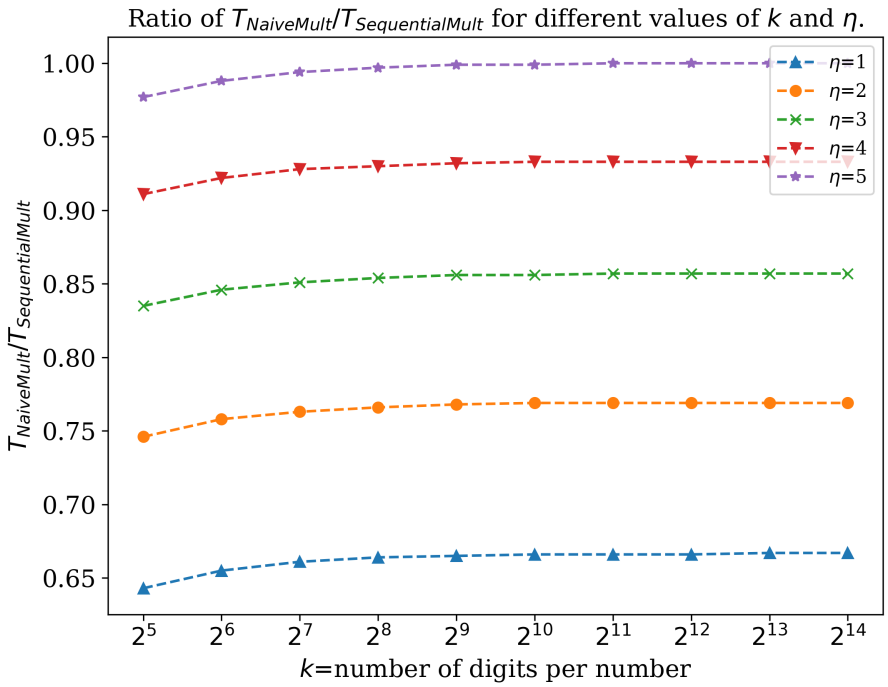

Figure 5.1: Estimated running-time ratio for various values of k and η.

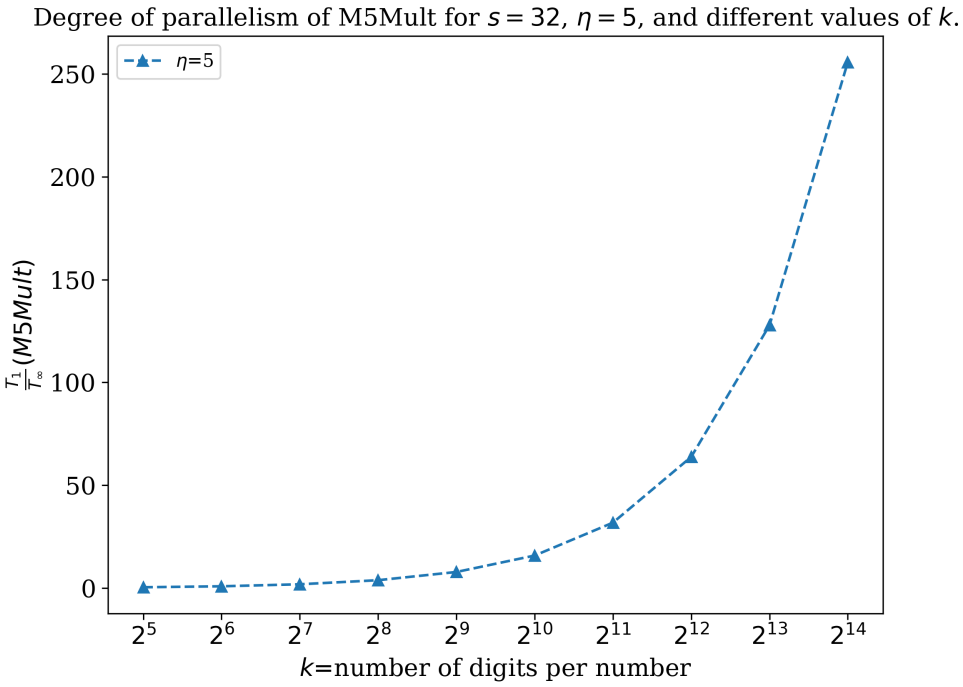

Figure 5.2: Estimated degree of parallelism for various values of k with $s = 32$ and $\eta = 5$.

Figure 5.3 presents the plot of average running time (in milliseconds) for computing batches of $k = 256$ digit integers using `OptMult` and `M5Mult`, also, Figure 5.4 presents the ratio diagram for the same comparison.

To have a better understanding of the performance of `M5Mult` and `OptMult`, we have profiled the two implementations on a NVIDIA GTX1080Ti for multiplying batches of $N = 1024$ integers of size $k = 256$ digits. Tables 5.4 and 5.5 present the performance counters for `M5Mult` and `OptMult`, respectively.

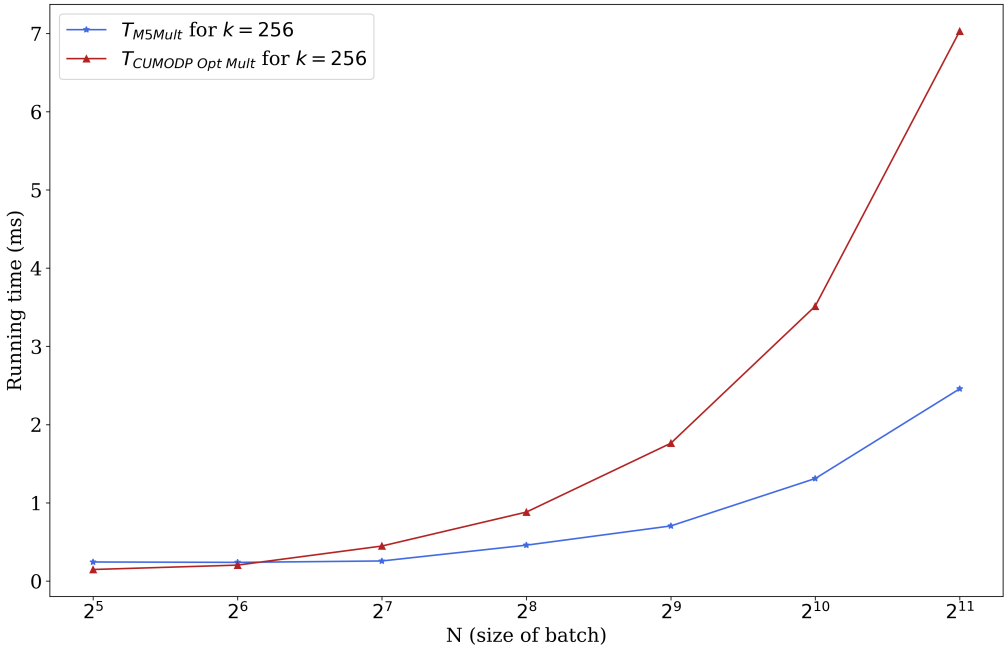

Figure 5.3: The average running time (in milliseconds) for computing batches of $k = 256$ digit integer multiplications using `OptMult` and `M5Mult`.

Table 5.4: Profiling results for computing a batch of $N = 2^{10}$ integers of size $k = 256$ digits using `M5Mult` with $s = 256$ ($\lambda = \frac{k}{s} = 1$) collected on NVIDIA GTX1080Ti.

Kernel for computing Z_i's (Loop I)	
Metric Name	Maximum
Achieved Occupancy	77 %
DRAM Read Throughput	163 GB/s
DRAM Write Throughput	149 GB/s
Shared Efficiency	0.00 %
DRAM Utilization	High
Branch Efficiency	100.00 %
IPC	0.75
Instruction Replay Overhead	0.016
Shared Load Throughput	0.00 B/s
Shared Store Throughput	0.00 B/s

Kernel for computing `SingleCM`	
Metric Name	Maximum
Achieved Occupancy	49 %
DRAM Read Throughput	4.51 GB/s
DRAM Write Throughput	6.41 GB/s
Shared Efficiency	64.86 %
DRAM Utilization	Low
Branch Efficiency	95.94 %
IPC	2.68
Instruction Replay Overhead	0.000073
Shared Load Throughput	1697 GB/s
Shared Store Throughput	472 GB/s

Kernel for adding Z_i's (Loop III)	
Metric Name	Maximum
Achieved Occupancy	80 %
DRAM Read Throughput	175 GB/s
DRAM Write Throughput	141 GB/s
Shared Efficiency	0.00 %
DRAM Utilization	High
Branch Efficiency	100.00 %
IPC	0.189
Instruction Replay Overhead	0.007
Shared Load Throughput	0.00 B/s
Shared Store Throughput	0.00 B/s

Alternatively, we have a pragmatic but unfair comparison between our GPU implementation and the highly optimized CPU implementation of the GMP library [9]. Our current imple-

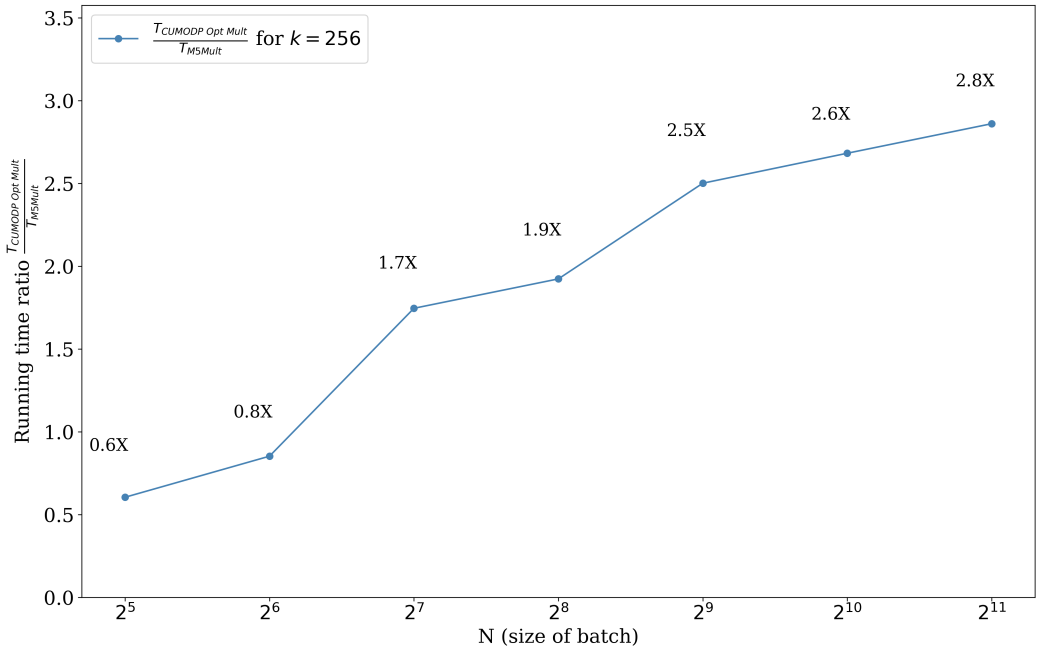

Figure 5.4: Comparing ratio of the average running time of `OptMult` to the average running time of `M5Mult` for computing batches of $k = 256$ digit integer multiplications.

Table 5.5: Profiling results for computing a batch of $N = 2^{10}$ integers of size $k = 256$ digits using `OptMult` from CUMODP library, collected on NVIDIA GTX1080Ti.

Kernel for computing `OptMult` (CUMODP)	
Metric Name	Max
Achieved Occupancy	68 %
DRAM Read Throughput	504.23 MB/s
DRAM Write Throughput	2.61 MB/s
Shared Efficiency	37.20 %
DRAM Utilization	Low
Branch Efficiency	86.71 %
IPC	0.62
Instruction Replay Overhead	0.000586
Shared Load Throughput	173 GB/s
Shared Store Throughput	73 GB/s

mentation has significant slowdown in comparison with GMP for integers larger than 1024 machine words. However, for values of k between 32 and 1024 machine words it demonstrates promising speedup ratios. For example, Figures 5.5 and 5.6 present the ratio of running time of GMP to `M5Mult` for computing multiplication of batches of N integers of $32 \leq k \leq 1024$. The results for GMP and `M5Mult` have been collected on an Intel-i7-7700K and a NVIDIA GTX1080Ti, respectively.

As it is demonstrated in Figures 5.5 and 5.6, for $32 \leq k \leq 1024$ `M5Mult` gains speedup as the size of a batch increases. On the other hand, the performance of `M5Mult` drops as k grows and gets closer to 1024.

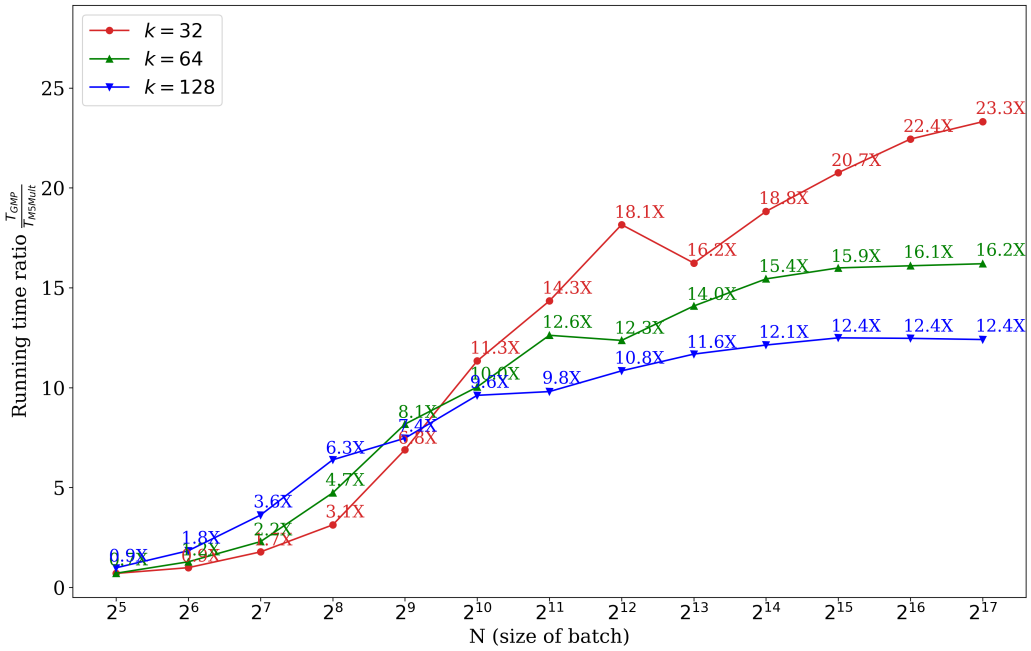

Figure 5.5: Comparing the ratio of the running time of GMP to the running time of `M5Mult` for computing batches of N integer multiplications of $32 \leq k \leq 128$ digits.

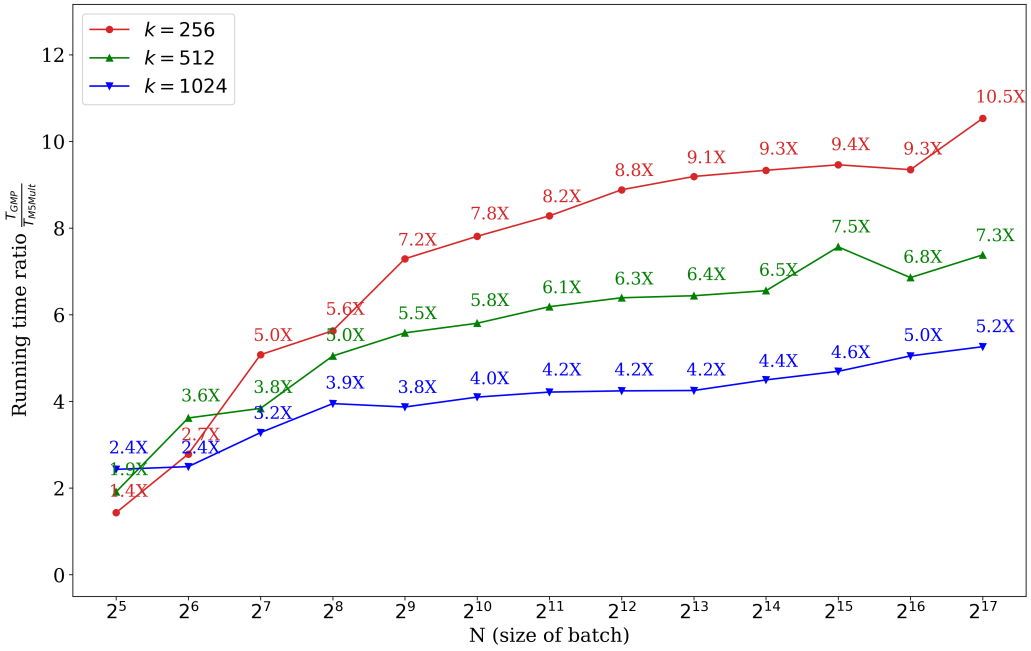

Figure 5.6: Comparing the ratio of the running time of GMP to the running time of `M5Mult` for computing batches of N integer multiplications of $256 \leq k \leq 1024$ digits.

5.8 Discussion

As a future work, we would like to apply this algorithm for carrying out component-wise multiplication of twiddle factors in computing FFT over large prime fields. This can further improve the performance of parallel implementation of FFT over large prime fields.

6 Conclusion

Our results in Chapter 2 show the advantage of the big prime field approach. To be precise, for a range of vector sizes, one can find a suitable large prime modulo which FFTs outperform the CRT-based approach.

In Chapter 3 we have presented an implementation of Fast Fourier Transforms over generalized Fermat prime fields on multi-threaded processors. Our parallel implementations using both specialized arithmetic and integer arithmetic from the GMP library achieve nearly linear parallel speedup. We noticed that the parallelization of our specialized implementation is slightly more successful than our GMP implementation. We attribute this higher performance to reduced number of arithmetic instructions due to using specialized arithmetic, minimal memory usage, and unrolling base-case DFT's and hard coding the constants. More precisely, our results prove that developing specialized arithmetic (e.g. Montgomery multiplication, Barret reduction, cyclic shift introduced in Section 3.2 and using inline assembly) can be beneficial. Doing so leads to reduced overhead compared to a more generic implementation such as large integer arithmetic functions available in GMP, or other libraries on top of GMP.

In Chapter 4, we have presented the KLARAPTOR tool for determining optimal CUDA thread block configurations for a target architecture, in a way which is adaptive to each kernel invocation and input data, allowing for dynamic data-dependent performance and portable performance. This tool is based upon our technique of encoding a performance prediction model as a rational program. The process of constructing such a rational program is a fast and automatic compile-time process which occurs simultaneously to compiling the CUDA program by use of the LLVM Pass framework. Our tool was tested using the kernels of the `Polybench/GPU` benchmark suite.

Finally, in Chapter 5, we have presented `M5Mult` which is well-structured to be parallelized and has a reasonably good degree of parallelism. Obviously, `M5Mult` is unsuitable for a sequential implementation. We have reported our experimental results for comparing the performance of our GPU implementation of `M5Mult` against another GPU-based implementation of schoolbook multiplication, as well as integer multiplication of the GMP library which targets CPUs.

Bibliography

[1] L. Chen, S. Covanov, D. Mohajerani, and M. Moreno Maza, "Big prime field FFT on the GPU," in *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation, ISSAC 2017, Kaiserslautern, Germany, July 25-28, 2017*, pp. 85–92, 2017.

[2] S. Covanov, D. Mohajerani, M. Moreno Maza, and L. Wang, "Big prime field FFT on multi-core processors," in *Proceedings of the 2019 on International Symposium on Symbolic and Algebraic Computation, ISSAC 2019, Beijing, China, July 15-18, 2019* (J. H. Davenport, D. Wang, M. Kauers, and R. J. Bradford, eds.), pp. 106–113, ACM, 2019.

[3] A. Brandt, D. Mohajerani, M. M. Maza, J. Paudel, and L. Wang, "KLARAPTOR: A tool for dynamically finding optimal kernel launch parameters targeting CUDA programs," *CoRR*, vol. abs/1911.02373, 2019.

[4] NVIDIA Corporation, "Profiler User's Guide." https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[5] 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"*, vol. 93, no. 2, pp. 232– 275, 2005.

[6] M. Frigo and S. G. Johnson, "FFTW: an adaptive software architecture for the FFT," in *IEEE,ICASSP '98, Seattle, Washington, USA, May 12-15, 1998*, pp. 1381–1384, 1998.

[7] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," in *Proceedings of the IEEE*, vol. 93, pp. 216–231, 2005.

[8] R. C. Whaley and J. Dongarra, "Automatically tuned linear algebra software," in *Proceedings of the 1998 IEEE/ACM Conference on Supercomputing*, 1998.

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

[10] W. B. Hart, "Fast Library for Number Theory: An Introduction, booktitle = Proceedings of the Third International Congress on Mathematical Software, series = ICMS'10, year =

2010, location = Kobe, Japan, pages = 88–91, numpages = 4, publisher = Springer-Verlag, address = Berlin, Heidelberg, note = http://flintlib.org,"

[11] V. Shoup, "Number theory library (NTL) for C++," *Available at Shoup's homepage http://www.shoup.net/ntl/*, 2016.

[12] E. A. Arnold, "Modular algorithms for computing Gröbner bases," *J. Symb. Comput.*, vol. 35, no. 4, pp. 403–419, 2003.

[13] X. Dahan, M. Moreno Maza, É. Schost, W. Wu, and Y. Xie, "Lifting techniques for triangular decompositions," in *ISSAC 2005, Proceedings* (M. Kauers, ed.), pp. 108–115, ACM, 2005.

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

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

[16] R. Agarwal and C. Burrus, "Fast convolution using fermat number transforms with applications to digital filtering," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 22, no. 2, pp. 87–97, 1974.

[17] V. S. Dimitrov, T. V. Cooklev, and B. D. Donevsky, "Generalized fermat-mersenne number theoretic transform," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 41, pp. 133–139, Feb 1994.

[18] S. Covanov and E. Thomé, "Fast arithmetic for faster integer multiplication," *CoRR*, vol. abs/1502.02800, 2015.

[19] M. Fürer, "Faster integer multiplication," in *Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego, California, USA, June 11-13, 2007* (D. S. Johnson and U. Feige, eds.), pp. 57–66, ACM, 2007.

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

[21] M. Fürer, "Faster integer multiplication," *SIAM J. Comput.*, vol. 39, no. 3, pp. 979–1005, 2009.

[22] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," *ACM Trans. Algorithms*, vol. 8, no. 1, pp. 4:1–4:22, 2012.

[23] A. Schönhage and V. Strassen, "Schnelle multiplikation großer zahlen," *Computing*, vol. 7, no. 3-4, pp. 281–292, 1971.

[24] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography.* Springer-Verlag New York, Inc., 2004.

[25] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[26] X. Li, M. Moreno Maza, R. Rasheed, and É. Schost, "The modpn library: Bringing fast polynomial arithmetic into maple," *J. Symb. Comput.*, vol. 46, no. 7, pp. 841–858, 2011.

[27] C. Chen, S. Covanov, F. Mansouri, M. Moreno Maza, N. Xie, and Y. Xie, "The basic polynomial algebra subprograms," in Hong and Yap [91], pp. 669–676.

[28] S. A. Haque, X. Li, F. Mansouri, M. Moreno Maza, W. Pan, and N. Xie, "Dense arithmetic over finite fields with the CUMODP library," in Hong and Yap [91], pp. 725–732.

[29] C. Chen, S. Covanov, F. Mansouri, M. Moreno Maza, N. Xie, and Y. Xie, "Parallel integer polynomial multiplication," in *SYNASC 2016*, pp. 72–80, 2016.

[30] J. Cooley and J. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965.

[31] NVIDIA Corporation, "CUDA C Programming Guide." https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[32] D. Mohajerani, "Fast fourier transforms over prime fields of large characteristic and their implementation on graphics processing units," Master's thesis, The University of Western Ontario, London, ON, Canada, 2016. http://ir.lib.uwo.ca/cgi/viewcontent.cgi?article=6094&context=etd.

[33] F. Franchetti, Y. Voronenko, and M. Püschel, "Tools and techniques for performance - FFT program generation for shared memory: SMP and multicore," in *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing, November 11-17, 2006, Tampa, FL, USA*, p. 115, 2006.

[34] A. Ali, L. Johnsson, and J. Subhlok, "Scheduling fft computation on smp and multicore systems," in *Proceedings of the 21st Annual International Conference on Supercomputing*, ICS '07, (New York, NY, USA), pp. 293–301, ACM, 2007.

[35] L. Meng, Y. Voronenko, J. R. Johnson, M. Moreno Maza, F. Franchetti, and Y. Xie, "Spiral-generated modular FFT algorithms," in *PASCO* (M. Moreno Maza and J. Roch, eds.), pp. 169–170, ACM, 2010.

[36] M. Moreno Maza and Y. Xie, "FFT-based dense polynomial arithmetic on multi-cores," in *HPCS*, vol. 5976 of *Lecture Notes in Computer Science*, pp. 378–399, Springer, 2009.

[37] A. De, P. P. Kurur, C. Saha, and R. Saptharishi, "Fast integer multiplication using modular arithmetic," in *STOC*, pp. 499–506, 2008.

[38] A. De, P. P. Kurur, C. Saha, and R. Saptharishi, "Fast integer multiplication using modular arithmetic," *SIAM J. Comput.*, vol. 42, no. 2, pp. 685–699, 2013.

[39] D. Harvey, J. van der Hoeven, and G. Lecerf, "Even faster integer multiplication," *J. Complexity*, vol. 36, pp. 1–30, 2016.

[40] D. Harvey, J. v. d. Hoeven, and G. Lecerf, "Faster polynomial multiplication over finite fields," *J. ACM*, vol. 63, pp. 52:1–52:23, Jan. 2017.

[41] D. Harvey, J. van der Hoeven, and G. Lecerf, "Fast polynomial multiplication over f260," in *Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation*, ISSAC '16, (New York, NY, USA), pp. 255–262, ACM, 2016.

[42] S. Covanov and E. Thomé, "Fast integer multiplication using generalized Fermat primes," *Mathematics of Computation*, 2018.

[43] M. Asadi, A. Brandt, C. Chen, S. Covanov, F. Mansouri, D. Mohajerani, R. Moir, M. Moreno Maza, L. Wang, N. Xie, and Y. Xie, "Basic Polynomial Algebra Subprograms (BPAS)," 2019. http://www.bpaslib.org.

[44] J. von zur Gathen and J. Gerhard, *Modern Computer Algebra (3. ed.).* Cambridge University Press, 2013.

[45] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[46] S. Covanov, "Putting Fürer Algorithm into practice," tech. rep., ORCCA Lab, London, 2014.

[47] P. Barrett, "Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor," in *Conference on the Theory and Application of Cryptographic Techniques*, pp. 311–323, Springer, 1986.

[48] F. Franchetti and M. Püschel, "FFT (fast fourier transform)," in *Encyclopedia of Parallel Computing*, pp. 658–671, 2011.

[49] Y. Torres, A. González-Escribano, and D. R. Llanos, "ubench: exposing the impact of CUDA block geometry in terms of performance," *The Journal of Supercomputing*, vol. 65, no. 3, pp. 1150–1163, 2013.

[50] S. Grauer-Gray and L.-N. Pouchet, "Implementation of polybench codes GPU processing," 2012. http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/GPU/.

[51] L. J. Stockmeyer and U. Vishkin, "Simulation of parallel random access machines by circuits," *SIAM J. Comput.*, vol. 13, no. 2, pp. 409–422, 1984.

[52] P. B. Gibbons, "A more practical PRAM model," in *Proc. of SPAA*, pp. 158–168, 1989.

[53] L. Ma, K. Agrawal, and R. D. Chamberlain, "A memory access model for highly-threaded many-core architectures," *Future Generation Comp. Syst.*, vol. 30, pp. 202–215, 2014.

[54] S. A. Haque, M. Moreno Maza, and N. Xie, "A many-core machine model for designing algorithms with minimum parallelism overheads," in *Parallel Computing: On the Road to Exascale, Proceedings of the International Conference on Parallel Computing, ParCo 2015*, vol. 27 of *Advances in Parallel Computing*, pp. 35–44, IOS Press, 2015.

[55] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *36th ISCA 2009*, pp. 152–163, 2009.

[56] J. Sim, A. Dasgupta, H. Kim, and R. W. Vuduc, "A performance analysis framework for identifying potential benefits in GPGPU applications," in *PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*.

[57] M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "A compiler framework for optimization of affine loop nests for GPGPUs," in *ICS 2008, Island of Kos, Greece, June 7-12, 2008*, pp. 225–234, 2008.

[58] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to GPU codes," in *Proc. InPar*, pp. 1–10, IEEE, 2012.

[59] M. Khan, P. Basu, G. Rudy, M. Hall, C. Chen, and J. Chame, "A script-based autotuning compiler system to generate high-performance CUDA code," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, 2013.

[60] K. Sato, H. Takizawa, K. Komatsu, and H. Kobayashi, "Automatic tuning of CUDA execution parameters for stencil processing," in *Software Automatic Tuning, From Concepts to State-of-the-Art Results*, 2010.

[61] J. Kurzak, Y. Tsai, M. Gates, A. Abdelfattah, and J. J. Dongarra, "Massively parallel automated software tuning," in *ICPP 2019, Kyoto, Japan, 2019*, pp. 92:1–92:10, 2019.

[62] T. Kistler and M. Franz, "Continuous program optimization: A case study," *(TOPLAS)*, vol. 25, no. 4, pp. 500–548, 2003.

[63] C. Song, L.-P. Wang, and T. J. Martínez, "Automated code engine for graphical processing units: Application to the effective core potential integrals and gradients," *Journal of chemical theory and computation*, vol. 12, no. 1, pp. 92–106, 2015.

[64] M. Püschel, J. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, and R. Johnson, "Spiral: A generator for platform-adapted libraries of signal processing alogorithms," *IJHPCA*, vol. 18, no. 1, pp. 21–45, 2004.

[65] C. Chen, X. Chen, A. Keita, M. Moreno Maza, and N. Xie, "MetaFork: A compilation framework for concurrency models targeting hardware accelerators and its application to the generation of parametric CUDA kernels," in *Proceedings of CASCON 2015*, pp. 70–79, 2015.

[66] Y. Liu, E. Z. Zhang, and X. Shen, "A cross-input adaptive framework for GPU program optimizations," in *2009 IEEE International Symposium on Parallel & Distributed Processing*, pp. 1–10, IEEE, 2009.

[67] R. V. L., B. N., and A. D. M., "Autotuning GPU kernels via static and predictive analysis," in *ICPP 2017*, pp. 523–532, 2017.

[68] J. D. Garvey and T. S. Abdelrahman, "Automatic performance tuning of stencil computations on gpus," in *ICPP 2015*, pp. 300–309, 2015.

[69] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[70] M. H. Stone, "The generalized weierstrass approximation theorem," *Mathematics Magazine*, vol. 21, no. 5, pp. 237–254, 1948.

[71] NVIDIA Corporation, "CUDA C Best Practices Guide." https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html.

[72] V. Volkov, *Understanding Latency Hiding on GPUs*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2016.

[73] NVIDIA Corporation, "CUDA Occupancy Calculator." https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html.

[74] NVIDIA Corporation, "CUDA runtime API." http://docs.nvidia.com/cuda/pdf/CUDA_Runtime_API.pdf.

[75] X. Mei and X. Chu, "Dissecting GPU memory hierarchy through microbenchmarking," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 1, pp. 72–86, 2017.

[76] H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *IEEE ISPASS 2010, 28-30 March 2010, White Plains, NY, USA*, pp. 235–246, IEEE Computer Society, 2010.

[77] J. E. Gentle, W. K. Härdle, and Y. Mori, *Handbook of computational statistics: concepts and methods*. Springer Science & Business Media, 2012.

[78] R. Corless and N. Fillion, *A graduate introduction to numerical methods*. Springer, 2013.

[79] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*. Philadelphia, PA: SIAM, 3rd ed., 1999.

[80] V. Volkov, "A microbenchmark to study GPU performance models," in *Proc. PPoPP*, pp. 421–422, 2018.

[81] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An adaptive performance modeling tool for GPU architectures," PPoPP '10, pp. 105–114, ACM, 2010.

[82] A. A. Karatsuba and Y. P. Ofman, "Multiplication of many-digital numbers by automatic computers," in *Doklady Akademii Nauk*, vol. 145, pp. 293–294, Russian Academy of Sciences, 1962.

[83] D. E. Knuth, *The art of computer programming, Volume II: Seminumerical Algorithms, 3rd Edition.* Addison-Wesley, 1998.

[84] D. G. Cantor and E. Kaltofen, "On fast multiplication of polynomials over arbitrary algebras," *Acta Informatica*, vol. 28, no. 7, pp. 693–701, 1991.

[85] D. Harvey and J. van der Hoeven, "Faster integer multiplication using short lattice vectors," *CoRR*, vol. abs/1802.07932, 2018.

[86] S. Covanov and E. Thomé, "Fast integer multiplication using \goodbreak generalized fermat primes," *Math. Comput.*, vol. 88, no. 317, pp. 1449–1477, 2019.

[87] D. Harvey and J. Van Der Hoeven, "Integer multiplication in time o (n log n)," 2019.

[88] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009.

[89] W. D. Hillis and G. L. S. Jr., "Data parallel algorithms," *Commun. ACM*, vol. 29, no. 12, pp. 1170–1183, 1986.

[90] S. A. Haque and M. Moreno Maza, "Plain polynomial arithmetic on gpu," in *Journal of Physics: Conference Series*, vol. 385, p. 012014, IOP Publishing, 2012.

[91] H. Hong and C. Yap, eds., *Mathematical Software - ICMS 2014 - 4th International Congress, Seoul, South Korea, August 5-9, 2014. Proceedings*, vol. 8592 of *Lecture Notes in Computer Science*, Springer, 2014.

Curriculum Vitae

Name: Davood Mohajerani

Education:

2017-2021 University of Western Ontario
London, Ontario, Canada
Ph.D. in Computer Science

2015-2016 University of Western Ontario
London, Ontario, Canada
M.Sc. in Computer Science

2010-2015 Isfahan University of Technology
Isfahan, Iran
B.Sc. in Computer Engineering