Parallel Additive Fast Fourier Transform Algorithms

Matan Hamilis

Technion - Computer Science Department - M.Sc. Thesis MSC-2016-15 - 2016

Parallel Additive Fast Fourier Transform Algorithms

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

Matan Hamilis

Submitted to the Senate
of the Technion — Israel Institute of Technology
Tamuz 5776 Haifa July 2016

The Research Thesis Was Done Under The Supervision of Prof. Eli Ben-Sasson and Prof. Mark Silberstein, in the Faculty of Computer Science.

Some results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author's doctoral research period, the most up-to-date versions of which being:

ICS 16' E. Ben-Sasson, M. Hamilis, M. Silberstein, and E. Tromer, Fast Multiplication in Binary Fields on GPUs via Register Cache, Proceedings of the 30^{th} International Conference on Supercomputing, ACM 2016.

E. Ben-Sasson and I. Ben-Tov and A. Chiesa and A. Gabizon and D. Genkin and M. Hamilis and E. Pergament and M. Riabzev and M. Silberstein and E. Tromer and M. Virza, Computational integrity with a public random string from quasi-linear PCPs, cryptology ePrint Archive, Report 2016/646.

Acknowledgements

First, I would like to thank my advisors, Eli and Mark, for their infinite patience and for wholeheartedly answering every question I had in any time. For cultivating my curiosity by posing new challenges, for giving me the tools to face these challenges and for pointing out possible research directions along the way.

I would like to thank my family as well, for constantly pushing me to saturate my uncontainable thirst for knowledge, for their inexpressible support and wisdom and for helping me to make one step at a time towards my dreams.

I am thankful for having my friends inside and outside the Technion for being there beside me along the way, for sharing their paths of lives and experiences with mine and for helping me maintaining a balanced, exciting and gladdening life. The importance of your interest in my work is immeasurable and without it, achieving the same results would have been impossible. For those who were listening to me without even understanding what I was talking about just so I would feel comfortable about my achievements and for those who did understand me for their constructive commentary and critique.

Last but not least I would like to thank my students along the years, which undoubtedly I have learned from more than they did from me.

The Generous Financial Help Of The Hilda and Manasche Ben-Shlomo Fellowship And The Technion Is Gratefully Acknowledged.

Contents

List of Figures

List of Algorithms

Abstract

Fast Fourier Transforms(FFTs), particularly over finite fields, have a main role in a large set of applications in the fields of signal and image processing, coding and cryptography. The computation of additive FFTs over finite fields is considered as a simpler and more scalable method than multiplicative FFTs due to the additive and recursive structure of finite fields. In this work we present an implementation of an algorithm to compute additive FFTs over finite fields of characteristic two – "binary fields" – to evaluate and interpolate polynomials of high degree over large affine subspaces. While previous works were applied only to linear subspaces, we apply a small modification to an existing algorithm to compute additive FFTs over affine subspaces as well. We present a parallel implementation of this algorithm for the GPU architecture and discuss its performance.

The FFT algorithm relies on an implementation of finite field arithmetics. Binary fields are used in a variety of applications in cryptography and data storage. Multiplication of two finite field elements is a fundamental operation and a well-known computational bottleneck in many of these applications, as they often require multiplication of a large number of elements. In this work we focus on accelerating multiplication in "large" binary fields of sizes greater than 2^{32}. We devise a new parallel algorithm optimized for execution on GPUs. This algorithm makes it possible to multiply large number of finite field elements, and achieves high performance via *bit-slicing* and fine-grained parallelization.

The key to the efficient implementation of the algorithm is a novel performance optimization methodology we call the *register cache*. This methodology speeds up an algorithm that caches its input in shared memory by transforming the code to use per-thread registers instead. We show how to replace shared memory accesses with the `shuffle`() intra-warp communication instruction, thereby significantly reducing or even eliminating shared memory accesses. We thoroughly analyze the register cache approach and characterize its benefits and limitations.

We apply the register cache methodology to the implementation of the binary finite field multiplication algorithm on GPUs. We achieve up to 138× speedup for fields of size 2^{32} over the popular, highly optimized Number Theory Library (NTL) [V. 03], which uses the specialized `CLMUL` CPU instruction, and over 30× for larger fields of size below 2^{256}. Our register cache implementation enables up to 50% higher performance compared to the traditional shared-memory based design.

Chapter 1

Introduction

Motivation

Interactive proofs (IP) were introduced to the world by Babai and Moran [L. 88] and by Goldwasser et al. [S. 89]. In an interactive proof, a protocol takes place between two main entities, a computationally-unbounded *Prover* and a computationally-bounded *Verifier*. Along the protocol the prover tries to prove a certain claim to the verifier, while the verifier has to verify the prover's proof using a probabilistic procedure. He can also ask the prover some questions regarding his proof, get the prover's answers and so on. After getting all the information he needs, the verifier can either *accept* or *reject* the proof.

A special kind of IP protocols are called PCP-protocols [L. 90, L. 91, S. 98, AS98] in which the verifier does not read the whole proof given to him by the prover, but only a small and negligible part of it and decides whether to accept or reject the proof according to the part he has read.

A major practical implication of the theorem is the ability to succinctly prove the computational integrity of a program running in time $T(n)$ using a PCP protocol with proof of length $\mathbf{poly}(T(n))$ is presented in [L. 90], [L. 91], [Kil92] and [Mic94].

The application motivating this work is to efficiently implement a family of probabilistically checkable proofs (PCP) of quasi-linear length, based on the work of Ben-Sasson and Sudan [E. 08]. This application is envisioned to enable verifiable execution, whereby a client that offloads a computation to untrusted computing resources, e.g., to a cloud system, receives a proof which attests that the results have indeed been produced by the execution of the offloaded computation. This property is also known as *computational integrity* and can be proved using PCPs in which the prover (e.g. the cloud system) proves the computational integrity of a given computation to a verifier (e.g. a client).

Since the prover is computationally unbounded, his role in the PCP protocol can be, theoretically, expensive in terms of computation. And in practice the prover's running time (and space consumption) turns out to be the main bottleneck preventing the system from running in feasible time. The prover executes a program and wishes to prove the

computational integrity of its execution to the verifier. To do that it has to encode the execution trace using error correcting codes that possess some interesting properties. It is known that error correcting codes that are based on low-degree polynomials have these properties. Particularly, Ben-Sasson and Sudan in their PCP [E. 08] have used Reed-Solomon codes [I. 60] that are based on univariate polynomials. With other additional restrictions it was required that the Reed-Solomon codes will be evaluated over affine-spaces in finite fields of characteristic-2 or characteristic-q where $q-1$ has small prime factors. The execution encoding algorithm is based on the evaluation and interpolation of polynomials over affine subspaces. These can be done efficiently using additive FFTs and inverse FFTs (IFFTs) and the implementation of them in finite fields of characteristic two is the scope of this work.

Fast Fourier Transforms

Fast Fourier Transforms(FFTs), particularly over finite fields, have a main role in a large set of applications in the fields of signal and image processing, coding and cryptography [D. 82, R. 02, F. 95, M. , Wel67, LRRY78, J. 98b].

The *discrete Fast-Fourier-Transform* (DFFT) algorithm for finite fields takes as input a polynomial $P(x)$ over a finite field GF $\left(p^k\right)$ and an a set of finite field elements and calculates $P(\alpha)$ for all α in that set. The inverse discrete fast Fourier transform (IDFFT) algorithm takes as input a function $f : S \to \text{GF}\left(p^k\right)$ where S is a set of elements from $\text{GF}\left(p^k\right)$ of size n and calculates the interpolation polynomial $P(x)$ over GF $\left(p^k\right)$ of degree $n-1$ such that for each $\alpha \in S : P(\alpha) = f(\alpha)$.

In 1965 the study of the implementation of DFFT algorithms has began by James Cooley and John Tukey who published in their historical paper [J. 65] a full description of an implementation for a DFFT algorithm known to Gauss [Gau66]. This algorithm was a multiplicative-FFT, as evaluating a polynomial over a set that is a multiplicative group and by that utilizing some of the multiplicative properties of that group. In finite fields of low characteristic (e.g. $GF(2^n)$ there are also additive subgroups over which DFFT algorithm can work. These algorithms, known as additive-DFFTs, evaluate a polynomial over a linear subspace. The computation of additive DFFTs in finite fields over affine subspaces is considered as a simpler and more scalable method than multiplicative DFFTs due to the additive and recursive structure of subspaces in finite fields. In this work we focus on the implementation of additive DFFTs and additive IDFFTs to which we will simply refer as FFTs and IFFTs, as non-discrete FFTs are out of the scope of this work. The first additive FFT / IFFT algorithm for a subspace of size n was the algorithm of Von-Zur-Gather and Gerhard that was published in [J. 03]. This algorithm performs $O(n \cdot \log^2 n)$ finite field multiplications and additions. In practice, finite field multiplication, is much slower and consumes and might consume a large portion of the running time. Therefore, we wish to find an FFT algorithm that achieves two goals,

1. Minimizes the number of finite-field multiplications

2. Minimize the time that each multiplication takes.

In this work we present an adaptation of Gao and Matteer's additive FFT and IFFT algorithm to affine subspaces over finite fields of characteristic two [S. 10] with a smaller number of finite field multiplications of $O(n \cdot \log n)$, compare to Von-Zur-Gather and Gerhard's algorithm.

We present an implementation of this algorithm to the GPU architectures and evaluate its performance. We implemented a CPU version of the algorithm as well. However, the full details of the implementation of the CPU algorithm is out of the scope of this work and we focus on the GPU implementation.

For completeness, we clarify that the CPU implementation of FFT achieves good running times and can evaluate polynomials of degree 2^{30} over 2^{30} elements in less than 20 minutes in our benchmark using a single thread. However, it scales badly on a high number of CPUs. The reason of this lack of scalability on CPU is left out of the scope of this work.

The GPU implementation gives more than 16x throughput compared to the serial CPU implementation. These implementations' performance heavily relies on the existence of efficient finite field multiplication an the implementation of such on the GPU architecture is the main focus of this work.

Finite Fields

Except for additive FFTs, binary fields have numerous applications in cryptography and data storage. For instance, the Advanced Encryption Standard (AES) [J. 98a] uses GF $\left(2^8\right)$, as does the error correction scheme used on Compact Discs (CDs) and Digital Versatile Discs (DVDs). Large fields are the basis for distributed storage systems like those used by Google and Amazon, which employ fields of size $2^{32}, 2^{64}$ and 2^{128} to ensure secure and reliable storage of data on multiple disks [J. 12]. They are also the basis for the application motivating this work: an efficient implementation of a family of probabilistically checkable proofs (PCP) of quasi-linear length [E. 08]. PCPs require very large binary fields: most of our work focuses on GF $\left(2^{32}\right)$ and GF $\left(2^{64}\right)$ but we also support fields of up to GF$\left(2^{2048}\right)$. Because all the applications mentioned above need to perform multiplication of a large number of finite field elements, their performance is dominated by the cost of finite field multiplication, motivating the never-ending quest for more efficient implementations of this fundamental arithmetic operation.

In this work we focus on accelerating finite field multiplication for large binary extension fields of size larger than GF$\left(2^{32}\right)$ on GPUs, where field elements are represented using a standard basis (cf. Chapter 2 for definitions). The main computational bottleneck in this case is the multiplication of polynomials over GF(2), that is, polynomials with $\{0, 1\}$-coefficients. The challenge posed by polynomial multiplication operations over

GF(2) has led Intel and AMD to add an instruction set extension `CLMUL` to support it in hardware.

We devise a novel parallel algorithm for multiplication in large binary extension fields on GPUs, which significantly outperforms the dedicated CPU hardware implementation. The algorithm is based on two main ideas: First, we apply *bit-slicing*, enabling a single thread to perform *32 multiplications in parallel*. As a result, all the arithmetic operations involved in multiplication are performed on 32 bits together instead of a single bit at a time for single multiplication, therefore matching the width of hardware registers and enabling full ALU utilization. Second, the computation of a single multiplication is further parallelized in a fine-grained manner to eliminate execution divergence among the participating threads. This critical step allows these computations to be mapped to the threads of a single GPU warp, whose threads are executed in lock-step.

We then focus on an implementation of the algorithm on modern NVIDIA GPUs. The key to implementation efficiency is a novel optimization technique that we call the *register cache*. The register cache enables us to use per-thread registers in conjunction with the `shuffle`() intrinsics, that enables intra-warp sharing of register values among threads, to construct a *register-based cache for threads in a single warp*. This cache serves the same purpose as the on-die shared memory, but is much faster thanks to higher bandwidth and reduced synchronization overhead. We propose a general methodology for transforming a traditional algorithm that stores its inputs in shared memory into a potentially more efficient one that uses private per-thread registers to cache the input for the warp's threads. We thoroughly study the benefits and limitations of the register cache approach on the example of a well-known k-Stencil kernel.

Finally, we apply the register cache methodology to optimize the implementation of the finite field multiplication algorithm for $\mathrm{GF}(2^N)$, where N=32,...,2048. The primary challenge is to scale the efficient single-warp implementation to larger fields while retaining the performance benefits of the register cache methodology. We analyze several design options, and apply an algorithm that uses a low-degree multiplication as a building block for multiplication in larger fields.

We evaluate our implementation across a variety of field and input sizes using NVIDIA Titan-X GPU with 12GB of memory, and compare it to a highly optimized CPU version of a popular Number Theory Library (NTL) [V. 03] running on a single core of Intel® Xeon® CPU E5-2620 v2 @ 2.10GHz that uses the Intel's CLMUL CPU instruction set extension. Our optimized implementation that uses register cache is up to 138× faster than NTL for $\mathrm{GF}(2^{32})$ when multiplying more than 2^{25} finite field elements. The register cache approach enables us to speed up the original shared memory version by about 50% over all field sizes.

Our contributions in this thesis are as follows:

1. A novel algorithm for polynomial multiplication over GF(2) on GPUs,
2. A general optimization methodology for using GPU registers as an intra-warp user-managed cache, along with an in depth analysis of this approach and its

application to polynomial multiplication.

3. Efficient GPU finite field multiplication that is up to two orders of magnitude faster in fields ($\mathrm{GF}(2^{32})$) than the CPU implementation that uses the specialized hardware instruction.
4. Efficient parallel implementation on CPU and cuda-GPU architectures of the additive FFT and inverse FFT algorithms.

This work is organized as follows. In chapter 2 we give some introductory background information on the theory of finite fields. In chapter 3 we present the problem of finite field multiplication in binary fields and discuss some previous results in that field. We also present the FFT algorithm which we implement in this work. In chapter 4 we briefly present the outlines of our CPU implementation of the FFT algorithm. Chapter 5 introduces the reader to the architecture and computational model of the GPU. In chapter 5.2 we introduce the *Register Cache* methodology to accelerate computation on GPU via caching values in registers. A small use-case example is given in which the benefits of this methodology are present. In chapter 6 we apply the register cache methodology on the multiplication of elements in binary fields. Chapter 7 discusses the implementation of the FFT algorithm in cuda-GPUs. Chapter 8 presents the performance evaluation of our finite field multiplication and FFT algorithms. Main conclusions and open questions for further research are given at chapter 9.

Related work

2-gapped polynomials The CPU implementation of NTL [V. 03] for the multiplication in binary fields uses the `CLMUL` [G. 14] instruction and employs 2-gapped polynomials to replace reduction with multiplications. We apply a similar algorithm in our work.

SIMD and bit-slicing The CPU SIMD instructions have been used to perform bit-slicing to parallelize $\mathrm{GF}(2^n)$ multiplication [J. 13]. Their implementation, however, is limited to small fields (up to $\mathrm{GF}(2^{32})$). The GPU architecture suits SIMD computation and can provide the same functionality as the CPU SIMD instruction set [S. 11]. The proposed implementation is, however, also limited to small fields (e.g $\mathrm{GF}(2^{16})$). Our implementation applies to larger fields.

Finite field multiplication on GPUs The previous works [J. 13, S. 11] are limited to fields of size smaller than 2^{32}. Particularly, Plank [J. 13] shows a CPU implementation that deals with computing a product of multiple elements by a single scalar, using scalar-dependent pre-computed lookup tables. Our work focuses on multiplying many *pairs of arbitrary elements*, therefore the lookup table approach is inapplicable.

Cohen et al. [A. 10] describes an implementation of finite field multiplication in specific binary fields. The performance reported in their paper is 3-orders of magnitude

slower than the performance reported in our work, and their implementation would benefit from bit slicing, register cache and reduced synchronization techniques presented here.

An implementation of finite field multiplication on GPUs over $GF(q)$ for some specific large NIST primes q is discussed in [K. 12]. Our implementation, however, is optimized for binary fields in a scalable fashion to achieve a generic implementation for a large variety of field sizes.

Register-based optimizations The benefits of reusing data in registers on GPUs to boost performance are well known. Volkov and Demmel [V.] present GPU implementations of LU decomposition and SGEMM.

Enfedaque et al. [P. 15] show how to implement the DWT (discrete wavelet transform) of an image of varying sizes where each warp calculates a different part of the output. They also show that shuffle-based communication achieves better results when the data each warp fetches from global memory is reused more times, as also confirmed by our results (cf. Section 5.2).

Davidson and Owens [A. 11] suggest a method called *register packing* to reduce shared memory traffic in GPU when dealing with a downsweep patterned computation, by performing some parts of the computation in registers.

Catanzaro et al. [B. 14] show a shuffle-based implementation for SIMD architectures, including the GPU. They discuss the benefits of the instruction for reducing shared-memory bandwidth and show the relation to the Array of Structs – Struct of Arrays transforms.

nVIDIA's Kepler Tuning Guide [nVi15] stresses the benefits of registers over shared memory in terms of latency and capacity. The `shuffle` instruction is suggested as an alternative for the use of shared memory in some cases.

We leverage the lessons learned in the previous work, and take one additional step by suggesting a register cache design methodology for reducing shared memory accesses to the input data. We demonstrate the application of this methodology on a challenging case of finite field multiplication in binary fields, and show that it achieves significant performance benefits.

Chapter 2

Preliminaries

This chapter briefly reminds the basic elements of polynomial rings and Galois fields that are necessary to our implementation of additive FFTs. For a thorough introduction to Galois fields see, e.g., [R. 97a].

The structure of this chapter is a follows; First a general definition to finite fields is given, then we discuss two of the most common representations for finite field elements, the *Polynomial Bases* and *Normal Bases*.

In the following chapters all references to finite field elements assume these are represented using a polynomial basis. The definition of a normal basis, being yet another popular representation for finite fields elements, is given here for completeness. We do not discuss the implementation of finite field multiplication represented using normal bases in this work.

2.1 Finite Extension Fields' Elements and Bases[1]

2.1.1 Definitions

A *finite field* or *Galois field* is a field with a finite number of elements. It is known that the number of elements in a finite field can only be a power of a prime number. Let p be a prime and q be a power of p, we denote by $\mathrm{GF}(q^n)$ or F_{q^n} the Galois-Field with q^n elements, which can be viewed as an extension field over F_q of order n. Therefore F_{q^n} can interpreted as a vector space of dimension n over F_q. Let $\alpha_0, \alpha_1, \ldots, \alpha_{n-1}$ be n linearly independent elements in F_{q^n} over F_q. Any element $e \in F_{q^n}$ can be represented as $e = \sum_{i=0}^{n-1} a_i \cdot \alpha_i$ where $a_i \in F_q$. We use the notation $e = (a_0, a_1, \ldots, a_{n-1})$ to state the $e = \sum_{i=0}^{n-1} a_i \cdot \alpha_i$.

Let $a = (a_0, a_1, \ldots, a_{n-1})$, $b = (b_0, b_1, \ldots, b_{n-1})$ be two elements in F_{q^n}. The addition of a and b is defined as $a + b = (a_0 + b_0, a_1 + b_1, \ldots, a_{n-1} + b_{n-1})$ which is a simple component-wise addition of the entries of a and b over F_q. However, multiplication tends to be not only more complicated but also more time consuming. We now give

[1] Definitions are based on [Gao93]

a general definition for the multiplication operation over finite extension fields using *multiplication tables*.

Denote by $T^0, T^1, \ldots, T^{n-1}$ be n matrices of size $n \times n$ over F_q s.t.

$$\alpha_i \alpha_j = \sum_{k=0}^{n-1} T_{ij}^k \alpha_k$$

So, T_{ij}^k is the coefficient of α_k in the product of α_i with α_j. Given three elements $a, b, c \in F_{q^n}$ such that $c = a \cdot b$ and $c = (c_0, c_1, \ldots, c_{n-1})$ the component c_k in the multiplication $a \cdot b$ is defined as $c_k = a \cdot T^k \cdot b$.

2.2 Polynomial Bases

The ring of polynomials Given a prime p $\mathrm{GF}(p)$ is the field with p elements $(0, 1, \ldots, p-1)$, with addition ($\oplus$) and multiplication ($\odot$) performed modulo p.

$\mathrm{GF}(2)$ is a field with two elements $(0, 1)$, with addition ($\oplus$) and multiplication ($\odot$) performed modulo 2. A *polynomial over GF*(2) is an expression of the form $A(x) := \sum_{i=0}^{d} a_i x^i$, where $a_i \in \mathrm{GF}(2)$ and x is a formal variable; henceforth we simply call $A(x)$ a *polynomial* because all finite field elements in this work are represented as polynomials over $\mathrm{GF}(2)$. The *degree* of A, denoted $\deg(A)$, is the largest index i such that $a_i \neq 0$. Addition and multiplication of polynomials (also called *ring addition and multiplication*) are defined in the natural way, i.e., for $B(x) = \sum_{i=0}^{m} b_i x^i$ with $m \geq d$ we have $A(x) \oplus B(x) = \sum_{i=0}^{m} (a_i \oplus b_i) x^i$ and $A(x) \odot B(x) = \sum_{j=0}^{d+m} x^j \cdot \bigoplus_{i=0}^{j} a_i \odot b_{j-i}$. The set of polynomials, augmented with the operations of addition and multiplication defined above, forms the *ring of polynomials over GF*(2), denoted $\mathrm{GF}(2)[x]$. Later, we reduce the problem of efficient multiplication in the *field* $\mathrm{GF}(2^n)$ to the problem of multiplying polynomials in the *ring* $\mathrm{GF}(2)[x]$.

The standard representation of a binary field The most common way to represent $\mathrm{GF}(2^n)$, also used here, is via a *standard basis*, as described next. A polynomial $r(x) \in \mathrm{GF}(2)[x]$ of degree n is called *irreducible* if there is no pair of polynomials $g(x), f(x) \in \mathrm{GF}(2)[x]$ such that $r(x) = g(x) \odot f(x)$ and $\deg(g), \deg(f) < n$. Many irreducible polynomials exist for every degree n. (Later, a special class of irreducible polynomials will be used to speed up multiplication.) Having fixed an irreducible $r(x)$, for every pair A, B of polynomials of degree $< n$, there exists a unique polynomial C of degree $< n$ such that $r(x)$ divides $A(x) \odot B(x) \oplus C(x)$ in the ring $\mathrm{GF}(2)[x]$; i.e., there exists $C'(x)$ such that $A(x) \odot B(x) \oplus C(x) = r(x) \odot C'(x)$. Denote the transformation that maps the pair of polynomials $(A(x), B(x))$ to the polynomial $C(x)$ by $\otimes_r$, where r is used to emphasize that this transformation depends on the irreducible polynomial $r(x)$. The set of polynomials of degree $< n$, along with ring addition $\oplus$ and

multiplication $\otimes_r$ defined above, is a *standard basis* representation[2] of GF (2^n). When the irreducible polynomial h is clear from context, we shall often drop it and denote GF (2^n) multiplication simply by $\otimes$.

Example of multiplication in standard representation In this example we show the field multiplication of two elements in GF $\left(2^4\right)$, using the standard representation induced by the irreducible degree-4 polynomial $r(x) := x^4 + x + 1$. Consider the two elements $A(x) = x + x^3$ and $B(x) = 1 + x^2$, represented in the standard basis by $a := (1010), b := (0101)$. To compute the 4-bit string $c = a \otimes_r b$ we work as follows:

- Compute the product $C'(x)$ of the two polynomials $A(x), B(x)$ in the ring GF $(2)[x]$, namely, $C'(x) := A(x) \odot B(x) = (x + x^3) \odot (1 + x^2) = x + \cancel{2x^3} + x^5$ (middle term canceled because we work modulo 2).
- Compute the remainder $C(x)$ of the division of $C'(x)$ by $r(x)$; in our example $C(x) = x^2$ and one can verify that $\deg(C) < 4$ and $r(x) \odot x = C'(x) \oplus C(x)$, as defined above.

Thus, $a \otimes_r b = c$ where $c := (0100)$.

Field multiplication reduces to ring multiplication The previous definitions and example show two main points that we exploit next. First, when multiplying two elements in the standard representation induced by $r(x)$, it suffices to (i) multiply polynomials in the ring GF $(2)[x]$ and then (ii) compute the remainder modulo $r(x)$. Second, the structure of $r(x)$ may influence the complexity of computing field multiplication.

2.3 Normal Bases

Given an element $\alpha \in F_{q^n}$ a normal basis for over F_q has the special form of $\alpha, \alpha^q, \ldots, \alpha^{q^{n-1}}$, let us denote by α_i the element α^{q^i}. Notice the fact that ${\alpha_i}^{q^j} = \alpha_{i+j}$. Therefore, given an element $a = a_0, a_1, \ldots, a_{n-1}$ in F_{q^n} note that $a^q = (a_{n-1}, a_0, a_1, ..., a_{n-2})$ so taking an element to the power of q is computationally simple as a right cyclic shift of the vector once and taking an element to the power of q^k is doing a cyclic shift by k places.

In our case, for $q = 2$, it is important not only that squaring can be executed in a fast manner for itself, because fast squaring affects the time needed for exponentiation using the repeated squaring and multiplication method, which by itself can make the inversion over the field much faster.

Additional important advantage is derived from the following observation, $\alpha_i \cdot \alpha_j = (\alpha \cdot \alpha_{j-i})^i$ (assuming $j \geq i$). So the k^{th} coefficient of $\alpha_i \cdot \alpha_j$ is the k^{th} coefficient of $(\alpha \cdot \alpha_{j-i})^i$ which is the $k - i$ coefficient of $\alpha \cdot \alpha_{j-i}$. Therefore for all i, j, k where $k, j > i$ it holds that $T_{ij}^k = T_{0,j-i}^{k-i}$, by taking $k = i$. So in fact we only need a multiplication

[2]The term "basis" refers to the algebraic fact that the n elements $1, x, x^2, \ldots, x^{n-1}$ are linearly independent over GF (2), i.e., they form a basis for GF (2^n) over GF (2); cf. [R. 97a] for more information.

table for $\alpha \cdot \alpha_i$ for all $0 \leq i < n$ and by this reducing by one dimension the size of multiplication tables, saving more space. Luckily, it was proved that there is a normal basis for any finite Galois extension of fields (The normal basis theorem), conjectured by Eisenstein in 1850 [Eis50] and first proved by Hensel in 1888 [Hen88].

In conclusion, only one multiplication table would suffice, notice that the time needed to multiply two elements depends on the number of non-zero entries in the table which will be called the *complexity of the base* and will be denoted by $c(N)$ where N is the normal basis of the field. So, we will be interested in bases with low complexities. An important theorem proved by Mullin et al. [R. 89] states that for each normal basis N of F_{q^n} over F_q, $c(N) \geq 2n - 1$, bases with this complexity will be called *optimal normal bases*. Optimal normal bases don't exist for all n for $q = 2$, but according to [Gao93] for 27 values of n where $2 \leq n \leq 64$ for which there exist a normal basis in F_{2^n} over F_2.

Chapter 3

Theoretical Discussion

3.1 Fast Multiplication in GF(2^n)

Finite field multiplication is generally far more time consuming than addition, both in terms of bit operations and in terms of machine cycles, when turning into finite field software implementation. This particularly holds for the fields we are interested in, finite fields of characteristic 2.

Multiplication speed in GF(2^n) is tightly connected to the field representation (addition is XOR under any basis for GF(2^n) over GF(2)). Two of the most common representations are, as stated in chapter 2,

1. Standard Basis: Elements are polynomials in GF (2) $[X]$ and multiplication is carried out modulo an irreducible polynomial of degree n over GF (2) $[X]$.

2. Normal Basis: Elements are the Frobenius automorphisms of a basic element α and multiplication is defined by a matrix. See [Gao93] for additional details.

We work under the standard basis. To speed up multiplication we choose a special kind of irreducible polynomial, called a *2-gapped* polynomial. We show that multiplication in GF(2^n) can be reduced to one multiplication of polynomials of degree n-1 and two multiplications of polynomials of degree $\frac{n}{2}$ over the ring GF (2) $[X]$ and $2n$ additions in GF(2). Let us first introduce the notion and importance of $k-$*Gapped* polynomials in GF(2^n) field for $k = 2$, as described in algorithm 3.1.

Definition 3.1.1. An irreducible polynomial $r(x) = \sum a_i x^i$ of degree d is **k-Gapped** if it can be written as $r(x) = x^d - r_1(x)$ where deg $(r_1(x)) \leq {}^d/_k$

Denote by $h_i(x)$ the value of $h(x)$ calculated on step i in algorithm 3.1

Lemma 3.1.2. $h_1(x) \equiv h_2(x) \mod r(x)$

Algorithm 3.1 2-Gapped Multiplication in GF(2^n)

Input:

- $a(x), b(x)$ of degree at most $n-1$ in $\mathbb{F}_2[X]$.
- $r(x) = x^n - r_1(x)$, 2-Gapped polynomial in $\mathbb{F}_2[X]$ of degree n.

Output: $h(x) = (a(x) * b(x)) \mod r(x)$

1: $h(x) \leftarrow a(x) * b(x)$
2: $h(x) \leftarrow h_0^{3n/2-1}(x) \oplus h_{3n/2}^{2n-1}(x) \odot r_1(x) * x^{n/2}$
3: $h(x) \leftarrow h_0^{n-1}(x) \oplus h_n^{3n/2}(x) \odot r_1(x)$
4: **return** $h(x)$

Proof.

$$\begin{aligned} a(x) \cdot b(x) &= h_1(x) \\ &= h_1^0(x) + x^{3n/2} \cdot h_1^1(x) && \deg\left(h_1^i(x)\right) \leq n/2 - 1 \\ &\equiv h_1^0(x) + r_1(x) \cdot h_1^1(x) \cdot x^{n/2} \mod r(x) && x^n \equiv r_1(x) \mod r(x) \\ &= h_2(x) \mod r(x) \end{aligned}$$

Lemma 3.1.3. $\deg(h_2(x)) \leq 3n/2 - 1$

Proof.

$$\begin{aligned} \deg(h_2(x)) &= \max\left(\deg\left(h_1^0(x)\right), \deg\left(x^{n/2}\right) + \deg\left(h_1^1(x)\right) + \deg(r_1(x))\right) \\ &\leq \max(n-1, n/2 + n/2 - 1 + n/2) \\ &= 3n/2 - 1 \end{aligned}$$

Lemma 3.1.4. $h_2(x) \equiv h_3(x) \mod r(x)$

Proof.

$$\begin{aligned} h_2(x) &= h_2^0(x) + x^n \cdot h_2^1(x) && \deg\left(h_2^0(x)\right) \leq n-1 \quad \deg\left(h_2^1(x)\right) \leq n/2 - 1 \\ &\equiv h_2^0(x) + r_1(x) \cdot h_2^1(x) \mod r(x) && x^n \equiv r_1(x) \mod r(x) \\ &= h_3(x) \mod r(x) \end{aligned}$$

Lemma 3.1.5. $\deg(h_3(x)) \leq n - 1$

Proof.

$$\begin{aligned} \deg(h_3(x)) &= \max\left(\deg\left(h_2^0(x)\right), \deg\left(h_2^1(x)\right) + \deg(r_1(x))\right) \\ &\leq \max(n-1, n/2 - 1 + n/2) \\ &= n/2 - 1 \end{aligned}$$

Algorithm 3.2 k-Gapped Multiplication in GF(p^m)

Input:

- $a(x), b(x)$ of degree at most $m-1$ in $\mathbb{F}_p[x]$.
- $r(x) = x^m - r_1(x)$, k-Gapped polynomial in $\mathbb{F}_p[x]$ of degree m.

Output: $h(x) = (a(x) \cdot b(x)) \mod r(x)$

1: $\ell \leftarrow {}^{m}/_{k}$
2: $h(x) \leftarrow a(x) \odot b(x)$
3: **for** $i = k-1$ down to 0 **do**
4: $\quad t \leftarrow m + \ell \cdot i$
5: $\quad h(x) \leftarrow h_0^{t-1}(x) \oplus h_t^{t+\ell-1}(x) \odot r_1(x) \odot x^{t-m}$
6: **return** $h(x)$

Lemma 3.1.6. $h_3(x) = (a(x) \cdot b(x) \mod r(x))$

Proof. From lemma 3.1.2 and lemma 3.1.4 we get $h_3(x) \equiv a(x) \cdot b(x) \mod r(x)$. From lemma 3.1.5 we get that $\deg(h_3(x)) \leq n-1$ so the equality holds. □

Algorithm 3.1 minimizes number of polynomial multiplications, can be adapted to multiplication in our field of interest, GF$\left(2^{64}\right)$.

3.1.1 Generalization for Optimized Multiplication in k-Gapped Finite Fields

In algorithm 3.2 we also present an extension to finite field multiplication in general $k-Gapped$ fields GF(p^m).

Let us denote by $A(n)$ and $M(n)$ as the numbers of additions and multiplications in GF(p) that performed when multiplying to polynomials of degree at most n over the ring GF$(p)[X]$.

Theorem 3.1. *Algorithm 3.2 performs,*

- $2m + A(m-1) + k \cdot A\left(\frac{m}{k}\right)$ *additions in* $GF\left(p^k\right)$.
- $M(m-1) + k \cdot M\left(\frac{m}{k}\right)$ *multiplications in* $GF\left(p^k\right)$

Proof. We will count separately the number of operations within polynomials multiplications and out of them.

- The number of additions in GF$\left(p^k\right)$ which are not part of polynomial multiplications is at most $2m$. In each iteration we add the polynomial $h_t^{t+\ell-1}(x) \odot r_1(x) \odot x^{t-m}$ which has at most $\frac{2m}{k}$ non-zero coefficients, which are the topmost coefficients, while others will be zero. This addition requires $\frac{2m}{k}$ additions. Over k iterations there will be $2m$ additions in total.
- There are no multiplications in GF(p) except for those which are part of polynomial multiplications.

Algorithm 3.3 Naïve polynomial multiplication

Input:
$a(x), b(x)$ of degree at most $n-1$.
Output: $c(x) = a(x) \odot b(x)$
1: **for** $i = 0, \ldots, n-1$ **do**
2: $c_i \leftarrow 0$
3: **for** $j = 0, \ldots, i$ **do**
4: $c_i \leftarrow c_i \oplus a_j \odot b_{i-j}$
5: **for** $i = n, \ldots, 2n-2$ **do**
6: $c_i \leftarrow 0$
7: **for** $j = i, \ldots, 2n-2$ **do**
8: $c_i \leftarrow a_{n-1+i-j} \odot b_{j-n+1}$
9: **return** $c(x) = \sum_{i=0}^{2n-2} c_i \cdot x^i$

- Let us denote by $A(n)$ and $M(n)$ the number of additions and multiplications in GF(p) needed to multiply two polynomials of degree n over the ring GF(p) [X] respectively. Our algorithm first multiplies two polynomials of degree at most $m-1$ in and then multiplies k times polynomials of degree $\frac{m}{k}$, all over the ring GF (p) $[X]$. This takes $M(m-1)+k \cdot M\left(\frac{m}{k}\right)$ multiplications and $A(m-1)+k \cdot A\left(\frac{m}{k}\right)$ additions in GF$\left(p^k\right)$ □

Notice that the number of multiplications and additions depends on the algorithm that is used to multiply polynomials. The complexity of polynomial multiplication has been extensively studied. The number of bit operations of the naïve algorithm (see Algorithm 3.3) is $O\left(n^2\right)$. More sophisticated algorithms by Karatsuba [KO63] and by Schonhage and Strassen [SS71, D. 91] are asymptotically faster, requiring $O\left(n^{\log_2 3}\right)$ and $O(n \log n \log \log n)$ bit operations, respectively.

In this work we use the naïve Algorithm 3.3 because it is fastest for polynomials of degrees below 1000 [M. 05] and its simplicity makes it a prime starting point for study.

Lemma 3.1.7 (Correctness)**.** *Algorithm 3.2 outputs* $(a(x) \cdot b(x)) \mod r(x)$.

Proof. Denote by $h_j(x)$ the value of $h(x)$ as computed after the iteration in which $i = j$ and $h_k(x)$ will be $h(x)$ before the loop. So $h_k(x) = a(x) \cdot b(x)$. We will prove by induction on j that $h_j(x) \equiv (a(x) \cdot b(x)) \mod r(x)$ and that the degree of $h_j(x)$ is at most $m + j \cdot \ell - 1$.

In the base case, $j = k$, and $h_k(x) = a(x) \cdot b(x)$ so its' degree is at most $2m$ and the claim obviously holds.

Assume the for some n we know that the claim holds, now we shall prove it for $n-1$. Denote by t the value of variable t in this iteration where $t = m + \ell \cdot (n-1)$. According

to step 5 of the algorithm $h_{n-1}(x) = (h_n)_0^{t-1}(x) + (h_n)_t^{t+\ell-1}(x) \cdot r_1(x) \cdot x^{t-m}$ so,

$$\begin{aligned}\deg(h_{n-1}(x)) &= \deg\left((h_n)_0^{t-1}(x) + (h_n)_t^{t+\ell-1}(x) \cdot r_1(x) \cdot x^{t-m}\right) \\ &= \max\left(\deg\left((h_n)_0^{t-1}(x)\right), \deg\left((h_n)_t^{t+\ell-1}(x) \cdot r_1(x) \cdot x^{t-m}\right)\right) \\ &= \max(t-1, \ell-1+\ell+t-m)\end{aligned}$$

Since $k \geq 2$ then $\ell = {}^{m}/_{k} \leq {}^{m}/_{2}$ so.

$$\begin{aligned}\deg(h_{n-1}(x)) &\leq \max(t-1, 2 \cdot {}^{m}/_{2} + t - m) \\ &= \max(t-1, t-1) \\ &= t-1 \\ &= m + \ell \cdot (n-1) - 1.\end{aligned}$$

Now we shall prove by induction that $h_i(x) \equiv h_k(x) \mod r(x)$. The base case is obvious for k. Assume that $h_n(x) \equiv h_k(x) \mod r(x)$, since $\deg(h_n(x)) \leq m+\ell \cdot (n)-1$ then, it can be written as $(h_n)_t^{t+\ell-1}(x) \cdot r_1(x) \cdot x^{t-m}$

$$\begin{aligned}h_{n-1}(x) &= (h_n)_0^{t-1}(x) + (h_n)_t^{t+\ell-1}(x) \cdot r_1(x) \cdot x^{t-m} \\ &\equiv (h_n)_0^{t-1}(x) + (h_n)_t^{t+\ell-1}(x) \cdot x^m \cdot x^{t-m} \mod r(x) \\ &= (h_n)_0^{t-1}(x) + (h_n)_t^{t+\ell-1}(x) \cdot x^t \\ &= h_n(x)\end{aligned}$$

So we return $h_0(x)$ of degree at most $m-1$ the equivalent to $h_k(x) = a(x) \cdot b(x)$ modulo $r(x)$, which proves the correctness of the algorithm. □

3.1.2 Finding a k-Gapped polynomial

According to Lidl and Niederreiter [R. 97b] the probability that a randomly chosen polynomial is irreducible is roughly ${}^{1}/_{n}$. In a paper published by HP [G. 98] it was mentioned that in binary extension fields, it should be quite probable (i.e. probability is bigger than a constant) to find $k - \textit{Gapped}$ pentanomials for k s.t. $\binom{k}{3} \approx n$, they also present a list of $k - \textit{Gapped}$ pentanomials that satisfy this equality for any practical n. They also raise an open question whether do irreducible binary pentanomials exist of degree n that are $\Omega\left(\sqrt[3]{n^2}\right)$ gapped.

Notice that multiplying any polynomial of degree m by a quadrinomial or pentanomial can be done in $O(m)$, so under the assumptions presented in [G. 98], this is a modular reduction in the polynomial ring $\mathrm{GF}(2)[X]$ with linear time complexity.

3.2 Generalizing Gao & Mateer's Additive FFT for affine subspaces

The Additive FFT algorithm introduced by Gao and Matteer [S. 10], when applied to binary fields, evaluates a $2^n - 1$ degree polynomial over a subspace of dimension n for general n in a finite field of characteristic two. It was the first algorithm that broke the $\Omega(n \cdot \log^2(n))$ multiplications barrier, with only $O(n \cdot \log(n))$ base field multiplications. See [Mat08] for previous FFT algorithms with the same runtime that were suited only for subspaces with dimensions which is a power of two.

We present a variation of that algorithm that fits affine subspaces as well. For the sake of completeness, we will describe the whole algorithm, relying on formulations and notations used by Gao and Mateer in their paper mentioned above.

3.2.1 Taylor Expansion

The additive FFT algorithm computes at some points the generalized Taylor expansion of polynomials at $\left(x^2 - x\right)$. A more general definition can be found in [J. 03] and [S. 10].

Given a polynomial $f(x) \in \mathbb{F}[x]$ of degree strictly smaller than $n = 2^{k+2}$ where $\mathbb{F}$ is a finite field of characteristic 2, the taylor expansion algorithm of f at $\left(x^2 - x\right)$ finds $m = \frac{n}{2}$ linear functions $h_0(x), h_1(x), ..., h_{m-1}(x) \in \mathbb{F}[x]$, such that,

$$f(x) = h_0(x) + h_1(x) \cdot (x^2 - x) + ... + h_{m-1} \cdot (x^2 - x)^{m-1}$$

We will denote this expansion as

$$T(f, n) = (h_0, ..., h_{m-1})$$

To compute the Taylor expansion, we first write $f(x)$ as $f(x) = f_0(x) + x^{2^{k+1}} \left(f_1(x) + x^{2^k} f_2(x)\right)$ where

- $\deg f_0 < 2^{k+1}$
- $\deg f_1 < 2^k$
- $\deg f_2 < 2^k$

$\mathbb{F}$ is a finite field of characteristic two, therefore,

$$x^{2^{k+1}} = (x^2 - x)^{2^k} + x^{2^k}$$

thus

$$f(x) = f_0(x) + x^{2^k} (f_1(x) + f_2(x)) + (x^2 - x)^{2^k} \left(f_1(x) + f_2(x) + x^{2^k} f_2(x)\right)$$

Algorithm 3.4 Taylor Expansion at $x^2 - x$

Input: (f, n) where $n \geq 1$ and $f(x) \in \mathbb{F}[x]$ of degree $< n$.
Output: $T(f, n)$, the taylor expansion of $f(x)$ at $x^2 - x$.

1: **if** $n \leq 2$ **then**
2: **return** $f(x)$
3: Find k such that $2^{k+1} < n \leq 2^{k+2}$.
4: Divide $f(x)$ into three parts as $f(x) = f_0(x) + x^{2^k+1}\left(f_1(x) + x^{2^k} f_2(x)\right)$
5: Set $h \leftarrow f_1 + f_2, \quad g_0 \leftarrow f_0 + x^{2^k} h, \quad g_1 \leftarrow h + x^{2^k} f_2$.
6: $V_1 \leftarrow T(g_0, n/2)$
7: $V_2 \leftarrow T(g_1, n/2)$
8: **return** (V_1, V_2)

Let $h(x) = f_1(x) + f_2(x), \quad g_0(x) = f_0(x) + x^{2^k} h_(x), \quad g_1(x) = h(x) + x^{2^k} f_2(x)$.
Then,

$$f(x) = g_0(x) + g_1(x)(x^2 - x)^{2^k}$$

Due to the degrees of f_0, f_1, f_2 we know that

$$\deg g_0, g_1 < 2^{k+1}$$

Therefore,

$$T(f, n) = \left(T(\left(g_0, 2^{k+1}\right)), T\left(g_1, 2^{k+1}\right)\right)$$

The time complexity of the algorithm, as described in [S. 10] is,

$$\begin{cases} \leq n \cdot \lceil \log_2(n/t) \rceil, & \text{for any } n \\ = \frac{1}{2} n \lceil \log_2(n/t) \rceil, & \text{when } n/t \text{ is a power of two} \end{cases} \tag{3.1}$$

A full description of the algorithm can be found in algorithm 3.4.

3.2.2 Additive FFT in Binary Fields Over Affine Subspaces

In this section we will conform with the notations of Gao and Mateer and extend their algorithm presented in [S. 10] to calculate additive FFTs over any affine subspace and not only over subspaces.

Our additive FFT algorithm works over a finite field $\mathbb{F}$ of characteristic 2. It gets as input a polynomial $f(x) \in \mathbb{F}[x]$, a basis of a subspace $\langle \beta_1, \ldots, \beta_m \rangle$ of dimension m, where $\beta_1, ..., \beta_m$ are linearly independent over GF(2), it also gets as input an affine shift s_B.

Let us define an ordering of the elements of B. Given a number $0 \leq i < 2^m$ with binary representation

$$i = a_1 + a_2 \cdot 2 + \cdots + a_m \cdot 2^{m-1} = (a_1, a_2, \cdots, a_m)_2,$$

Where each a_j is either 0 or 1. The i^{th} element of affine subspace B is

$$B[i] = s + a_1\beta_1 + a_2\beta_2 + \cdots + a_m\beta_m.$$

The algorithm's output is the evaluation of $f(x)$ over all elements in the affine subspace $B = s_B + \langle \beta_1, ..., \beta_m \rangle$, and will be denoted as,

$$FFT(f, m, B) = \left(f\left(B[0]\right), f\left(B[1]\right), \cdots, f\left(B[2^m - 1]\right)\right)$$

The algorithm is recursive, we show how to reduce a problem of size $n > 2$ to two problems of size $k = n/2 = 2^{m-1}$. Let

$$\begin{aligned} \gamma_i &= \beta_i \cdot \beta_m^{-1}, \quad 1 \le i \le m-1 \\ s_G &= s_B \cdot \beta_m^{-1} \end{aligned}$$

and

$$G = s_G + \langle \gamma_1, ..., \gamma_m \rangle \tag{3.2}$$

Let $g(x) = f(\beta_m x)$. Evaluating $f(x)$ over B is equivalent to the evaluation of $g(x)$ over $G \cup (G+1)$. So we wish to calculate $\mathrm{FFT}(g, G)$ and $\mathrm{FFT}(g, (G+1))$. Let $D = s_D + \langle \delta_1, \ldots, \delta_{m-1} \rangle$ where,

$$\delta_i = \gamma_i^2 - \gamma_i, \quad 1 \le i \le m-1$$

We know that each γ_i is not 1 or 0, so δ_i is not 0. Since $\gamma_1, ..., \gamma_m$ and 1 are linearly independent over GF(2) the elements $\delta_1, \ldots, \delta_{m-1}$ are linearly independent over GF(2) as well and span the affine subspace,

$$D = s_D + \langle \delta_1, \ldots, \delta_{m-1} \rangle$$

of size $k = 2^{m-1} = n/2$.

Notation 1. Given $\alpha = a_1\gamma_1 + \cdots + a_{m-1}\gamma_{m-1} \in G$, the element α^* is

$$\alpha^* = \alpha^2 - \alpha = a_1\delta_1 + \cdots + a_{m-1}\delta_{m-1}$$

Therefore,

$$G[i]^* = D[i], \quad 0 \le i < k$$

Suppose we are given the Taylor expansion of $g(x)$ at $x^2 - x$.

$$g(x) = \sum_{i=0}^{k-1} (g_{i0} + g_{i1}x) \cdot \left(x^2 - x\right)^i \tag{3.3}$$

and $g_{ij} \in \mathbb{F}$. Let

$$g_0(x) = \sum_{i=0}^{k-1} g_{i0} \cdot x^i, \quad \text{and} \quad g_1(x) = \sum_{i=0}^{k-1} g_{i1} \cdot x^i. \tag{3.4}$$

Notice that for any $\alpha \in G$ and $b \in \mathrm{GF}(2)$, since $(\alpha + b)^2 - (\alpha + b) = \alpha^*$, we have

$$g(\alpha + b) = (g_0(\alpha^*) + \alpha \cdot g_1(\alpha^*)) + b g_1(\alpha^*) \tag{3.5}$$

Therefore, the FFT of $g(x)$ can be calculated from the FFTs of $g_0(x)$ and $g_1(x)$ over D. Let the FFT of $g_0(x)$ and $g_1(x)$ over D be,

$$\begin{aligned} FFT(g_0, m-1, D) &= (u_0, u_1, \ldots, u_{k-1}), \quad u_i = g_0(D[i]) \\ FFT(g_1, m-1, D) &= (v_0, v_1, \ldots, v_{k-1}), \quad v_i = g_1(D[i]) \end{aligned} \tag{3.6}$$

Equation 3.5 implies that

$$FFT(g, m-1, G) = (w_0, w_1, \ldots, w_{k-1})$$

Where $w_i = u_i + G[i] \cdot u_i$ for $0 \leq i < k$. It also implies that,

$$FFT(g, m-1, G+1) = FFT(g, m-1, G) + FFT(g_1, m-1, D).$$

This reduction step is applied recursively until the input polynomials are linear functions that can be evaluated easily. In algorithm 3.5 a summary of the written above can be found.

The only two additions we made to Gao and Mateer's algorithm is calculating recursively a series of affine shifts. The only place which these shifts take place is the bottom of the recursion, where we evaluate the linear function as described in step 1.

We will now compute the runtime of the algorithm. To compute the basis elements of G and D and the affine shifts in step 5, we perform $2m+2(m-1)+\cdots+2\cdot 2 = m(m+1) = O(\log_2^2(n))$ multiplications, and the number of additions is $m+(m-1)+\cdots+2 = m(m-1)/2 = O(\log_2^2(n))$. In step 2, we compute the powers of β_m^i for $2 \leq i \leq n-1$, with a total number of multiplications that is at most $(2^m-2)+(2^{m-1}-2)+\cdots+(2^2-2) < 2\cdot 2^m = 2n$. Up until now, the whole computation can be preprocessed, and costs negligible time.

In step 1 the recursion ends and it costs 2 multiplications and 2 additions. Step 2 costs an additional $n-1$ multiplications (besides computing the powers of β_m). The Taylor expansions cost additional $\frac{1}{2} \cdot n \cdot \log_2(n) - \frac{1}{2} \cdot n$ additions. Step 7 has two invocations of the FFT algorithm of size $n/2$. Step 10 costs n multiplications and n additions. Let $M(n)$ and $A(n)$ denote the number of multiplications and additions performed by the algorithm, respectively, on an input of size n. Then $M(2) = A(2) = 2$,

Algorithm 3.5 Additive FFT of length $n = 2^m$

Input:

- $f(x) \in \mathrm{GF}\left(2^t\right)[X]$ of degree $< n = 2^m$.
- $B = \langle \beta_1, ..., \beta_m \rangle$, a basis with linearly independent elements over GF(2).
- S, an affine shift to the subspace spanned by β_i's.

Output: $FFT(f, m, B, S) = (f(B[0] + S), ..., f(B[n-1] + S))$

1: If $m = 1$ then return $f(S), f(S + \beta_1)$. ▷ Linear Evaluation Phase.
2: Compute $g(x) = f(\beta_m x)$. ▷ Shift Phase.
3: Compute the Taylor expansion of $g(x)$ as in algorithm 3.4 ▷ Taylor Expansion Phase.
4: Let $g_0(x)$ and $g_1(x)$ from $g(x)$ as in (3.4). ▷ Shuffle Phase.
5: Compute $\gamma_i \leftarrow \beta_i \cdot \beta_m^{-1}$, $\delta_i \leftarrow \gamma_i^2 - \gamma_i$ for $1 \le i < m$, $s_G \leftarrow S \cdot \beta_m^{-1}$ and $s_D \leftarrow s_G^2 - s_G$.
6: Let $G \leftarrow s_G + \langle \gamma_1, \ldots, \gamma_{m-1} \rangle$, and $D \leftarrow \langle \delta_1, \ldots, \delta_{m-1} \rangle$.
7: Let $k = 2^{m-1}$ compute
$\mathrm{FFT}(g_0, m-1, D, s_D) = (u_0, u_1, \ldots, u_{k-1})$, and
$\mathrm{FFT}(g_1, m-1, D, s_D) = (v_0, v_1, \ldots, v_{k-1})$.
8: **for** $0 \le i < k$ **do**
9: $w_i \leftarrow u_i + G[i] \cdot v_i$
10: $w_{k+1} \leftarrow w_i + v_i$. ▷ Merge Phase.
11: **return** $(w_0, w_1, \ldots, w_{n-1})$

and for any $n = 2^m > 2$, it holds that

$$\begin{aligned} M(n) &= 2 \cdot M\left(\tfrac{n}{2}\right) + 2n - 1, \\ A(n) &= 2 \cdot A\left(\tfrac{n}{2}\right) + \tfrac{1}{2} \cdot n \cdot \log_2(n) + \tfrac{1}{2} \cdot n. \end{aligned}$$

By induction we get

$$\begin{aligned} M(n) &= 2 \cdot n \cdot \log_2(n) - \tfrac{3n}{2} + 1, \\ A(n) &= \tfrac{1}{4} \cdot n \cdot (\log_2(n))^2 + \tfrac{3}{4} \cdot n \cdot \log_2(n) \end{aligned}$$

Chapter 4

CPU

In this chapter we briefly discuss the implementation of the additive FFT in binary fields algorithm and the finite field multiplication operation on which it relies. We rather state the functionality of our implementation so it can be compared to the GPU finite field arithmetics and implementation. We clarify that the main scope of this work is not a CPU implementation of neither additive FFTs over binary fields or these fields' multiplication and this chapter is given for completeness.

4.1 Finite Field Arithmetics

In this section we discuss the implementation of finite field arithmetics needed to compute the additive FFT in $\mathrm{GF}\left(2^{64}\right)$.

4.1.1 Element Representation on CPU

A $\mathrm{GF}\left(2^{64}\right)$ field element represented in the standard basis is a binary polynomial whose degree is at most 63. All operations are performed modulo an irreducible polynomial of degree 64, that will be denoted by $r(x)$ Each element in $\mathrm{GF}\left(2^{64}\right)$ is known to be equivalent to a unique polynomial modulo $r(x)$. Each element e will be represented in the polynomial basis using the polynomial

$$p_e(x) = \sum_{i=0}^{63} c_e^i x^i$$

4.1.2 Finite Field Library API

To support the implementation of the FFT algorithm on CPU we have implemented the following operations for elements in $\mathrm{GF}\left(2^{64}\right)$,

Addition Given two elements a, b the representation of $a + b$ is $c_a \oplus c_b$. The implementation of addition in $\mathrm{GF}\left(2^k\right)$ is just bitwise XOR of the two elements.

Multiplication See section 3.1 for a full theoretical description of this operation. Implementation is given in section 4.1.3.

Squaring Implemented as a multiplication of an element by itself.

Exponentiation Implemented using the repetitive squaring and multiplying algorithm.

Inversion The inverse of an element a is an element b s.t. there exists a polynomial $q(x)$ for which $a(x) \cdot b(x) + q(x) \cdot r(x) = 1$.The inversion is implemented by implementing the extended euclidean algorithm to find this b. The inversion operation is used only n times when evaluating an FFT of a subspace of size 2^n and due to the very limited use of this operation, it was implemented in a very naive manner.

4.1.3 Implementation of multiplication in GF(2^{64})

GF(2^n) multiplication has received considerable attention (cf. [J. 86, E. 96]) and is implemented efficiently for CPU in popular software libraries like NTL [V. 03] and MPFQ [1]. Moreover, in large part because of the importance of GF (2^n) multiplication, Intel introduced in 2010 a dedicated CPU instruction called `CLMUL` which performs GF $(2)[x]$ ring multiplication of polynomials of degree up to 64 in 7–14 cycles [Fog16].

Both NTL and MPFQ use this dedicated instruction. This instruction can be used to multiply polynomials of higher-degree, thereby supporting GF (2^n) multiplication for values $n > 64$ (cf. [C. 12] for one such implementation).

Algorithm 6.1 shows how to perform finite field multiplication in binary fields with elements being represented in the standard basis and the irreducible polynomial is 2-Gapped. The multiplication in GF(2^n) by this algorithm is composed of three multiplications of polynomials in the ring GF $(2)[X]$ of degrees up to $k-1$ and to additions of such polynomials. The multiplication of such polynomials can be implemented using the `CLMUL` instruction that was mentioned before. The implementation is detailed in algorithm 4.2.

Notation 2. Given a polynomial $p(x) = \sum a_k x^k$ we will denote by $p_i^j(x) = \sum_{k=i}^{j} a_k x^{k-i}$

Note that the multiplication by x^{32} in line 2 will be implemented by an arithmetic shift-left of the bits.

The NTL library proposed the same implementation shown here when performing multiplication with a 2-Gapped irreducibles in binary fields. While NTL's implementation is more general, we focused in our CPU implementation on the specific field of GF$\left(2^{64}\right)$. The implementation, using `CLMUL` instruction is described in algorithm 4.2.

Our C++ implementation of algorithm 4.2 cuts NTL's implementation by half. A full comparison and performance analysis is detailed in chapter 8

[1] http://mpfq.gforge.inria.fr/doc/doc.html

Algorithm 4.1 Multiplication in GF(2^n)

Input:

- $a(x), b(x)$ of degree at most $n-1$ in $\mathbb{F}_2[X]$.
- $r(x) = x^n + r_1(x)$, 2-Gapped polynomial in $\mathbb{F}_2[X]$ of degree n.

Output: $h(x) = (a(x) \odot b(x)) \mod r(x)$

1: $h(x) \leftarrow a(x) \odot b(x)$
2: $h(x) \leftarrow h_0^{3n/2-1}(x) \oplus h_{3n/2}^{2n-1}(x) \odot r_1(x) \odot x^{n/2}$
3: $h(x) \leftarrow h_0^{n-1}(x) \oplus h_n^{3n/2}(x) \odot r_1(x)$
4: **return** $h(x)$

Algorithm 4.2 2-Gapped Multiplication in GF$\left(2^{64}\right)$ using `CLMUL`

Input:

- $a(x), b(x)$ of degree at most 63 in $\mathbb{F}_2[X]$.
- $r(x) = x^{64} + r_1(x)$, 2-Gapped polynomial in $\mathbb{F}_2[X]$ of degree 64.

Output: $h(x) = (a(x) \cdot b(x)) \mod r(x)$

1: $h(x) \leftarrow \texttt{CLMUL}(a(x), b(x))$
2: $h(x) \leftarrow h_0^{95}(x) + \texttt{CLMUL}\left(h_{96}^{127}(x), r_1(x)\right) \cdot x^{32}$
3: $h(x) \leftarrow h_0^{63}(x) + \texttt{CLMUL}\left(h_{64}^{95}(x), r_1(x)\right)$
4: **return** $h(x)$

4.2 Parallel FFT and inverse FFT implementation

We implemented the additive FFT and IFFT algorithms of Gao and Matteer [S. 10]. The implementation was parallelized to a large number of cores using openMP [B. 07]. Unfortunately, despite being very fast for a single thread, the implementation didn't scale-up for large number of cores. The reason for this lack of scalability of this implementation is left out of the scope of this work.

Chapter 5

GPU - Introduction of Register Cache

5.1 Introduction of GPUs[1]

The *Graphics Processing Unit* (GPU) is a prarallel machine that runs many threads in parallel. Threads are the basic units of execution in it that process words of size $\mathcal{B}$. Each thread has local memory in the form of registers. The number of registers in the GPU is limited and they are partitioned evenly among running threads. Threads are grouped into warps. A warp is a set of $\mathcal{W}$ threads that operates in lock, i.e., at a given step all threads in the warp execute the same instruction. A thread-block is a set of warps that can share a dedicated memory used for for communication between threads of the same thread block. The set of instructions is fixed and called PTX-ISA. A set of thread-blocks is called a grid and is the largest unit of computation we are interested in, representing all running threads. It has global memory of practically unlimited size but accessing it is slower than accessing the shared memory or local memory (registers). Each thread also has a unique thread-ID which it has access to. These IDs are distributed among threads in a way that all threads of the same warp posses $\mathcal{W}$ consecutive IDs.

Grouping threads into warps has a major significance not only on the computational model itself but on global memory accesses efficiency as well. Global memory accesses are issued by the device in the form of *transactions* where each transaction reads from or writes to a large number of addresses in a single burst. All load and store operations issued by threads of the same warp are coalesced by the device to minimize the number of transactions required to perform the requested operations. The more scattered the load/store addresses are, the more transactions will be needed to perform the load/store operation, therefore each GPU programmer must make an effort to make his global

[1]Based on nVidia's white papers of the Fermi and Kepler architectures.
(`http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf`)
(`http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`)

memory accesses as coalesced as possible. GPU is consisted of many ALUs. Each warp is executed on a single ALU. The ALUs can execute the same instruction over $\mathcal{W}$ inputs. At each time step each ALU performs an instruction with input begin sent by each of the threads of the warp. ALU executes the instruction on all inputs in parallel and returns each result to the relevant thread. This kind of computation in which a single instruction is executed, in parallel, on many inputs is called *SIMD* (single instruction - multiple data). Since ALUs are SIMD, the processing time increases if threads diverge on the instruction executed at a given time step. Thus, warps whose threads all give the same

be given to maintaining a non-divergent execution of all warps.

Intra-warp communication is very common and can be done in two ways,

1. Shared Memory, to which one thread can write data that, later on, other threads can read.

2. Shuffle, which is a special instruction executed by all threads in the same warp where thread i shares a value v_i stored in its register and also states a number t_i of a thread in the warp. The call $shuffle(v_i, t_i)$ will make thread t_i write the value v_{t_i} into a register of thread i.

The main complexity measures we seek to optimize are minimizing the communication complexity between the threads and memory, by efficiently using the registers and minimizing accesses to shared and global memory. We also wish to minimize the parallel running time, measured in number of parallel instructions. This will be achieved by minimizing divergent executions and having all global memory accesses coalesced. We shall first look at multipoint multiplication over finite fields of characteristic 2, and later on we will examine the interpolation and multi-point evaluation problems, also known as FFT and inverse-FFT. In this latter case we care about evaluation over affine subspaces.

The key to the efficient implementation of the finite field multiplication is a novel performance optimization methodology that we call *register cache* and is discussed in chapter 5.2. This methodology allows to speed up an algorithm that uses shared memory for caching its input by transforming it to use per-thread registers instead. We show how to replace shared memory accesses with the `shuffle` intra-warp communication instruction, and thereby significantly reduce and even entirely eliminate shared memory accesses. We thoroughly analyze the register cache approach and characterize its benefits and limitations.

We apply the register cache methodology to the implementation of the binary finite field multiplication algorithm on GPUs. We achieve up to 138× speedup for fields of size 2^{32} over the popular highly optimized Number Theory Library (NTL) [V. 03] which uses specialized `CLMUL` CPU instruction, and over 30× for larger fields of size below 2^{256}. Our register cache implementation enables up to 50% higher performance compared to the traditional shared-memory based design.

Next, we deeply elaborate an implementation of additive FFT and inverse additive FFT over affine subspaces that achieves high occupancy, meaning that the processing hardware in the GPU chip is idle in a negligible part of the computation time. All implementations are non-divergent and have no non-coalesced global memory accesses.

5.2 Intra-warp register cache

On-die shared memory is commonly used for data sharing among threads in the same $\mathcal{TB}$. One common practice is to optimize input data reuse, whereby the kernel input is first prefetched from the global memory into the shared memory, thus saving global memory access cost on the following accesses.

In this section we focus on the *register cache*, a design methodology whose goal is to improve kernel performance by transforming computations to use registers instead of shared memory. We use private registers in each thread as a distributed storage, effectively implementing a layer of user-managed cache for the threads in the same warp with the help of the `shuffle`() instruction.

The benefits of using registers and `shuffle`() are well known in SIMD architectures [B. 14], and are embraced in GPU computing [nVi15, V. , P. 15, A. 11, S. 16, G. 15]. The `shuffle`()-based design removes the $\mathcal{TB}$-wise synchronization overhead associated with the use of shared memory, and allows higher effective memory bandwidth to the data stored in registers. However, the existing uses of `shuffle`() are application-specific and offer no guidance for the design of the algorithm. Here we suggest a systematic approach to constructing `shuffle`()-based algorithms, aiming specifically to optimize applications with significant input data reuse.

Problem setting We consider a common application scenario in which threads prefetch the shared $\mathcal{TB}$ input into shared memory and then access it repeatedly. Our goal is to reduce the use of shared memory as much as possible by identifying sharing and access patterns among the threads of a single warp, and replacing certain or all shared memory accesses by `shuffle`().

Overview We start with a shared memory-based implementation. The following steps accomplish the kernel transformation to use registers instead.

1. Identify warp inputs in shared memory.
2. Distribute inputs across warp threads such that each thread stores some part of the shared input in its registers. The optimal distribution is application dependent.
3. Logically break computations into two interleaving bulk-synchronous phases: communication and computation. The communication phase corresponds to the shared memory accesses in the original implementation. The computation phase is similar to the original implementation, but uses only the data in local registers.

Communication phase We now describe the communication phase transformations in greater detail.

1. For each thread, declare the data to be read from other warp threads. We refer to each access as a `Read`(var, tid) operation, such that tid is the thread to read from, and var is the remote variable holding the data, both determined by the

data distribution.

2. For each thread, compile the list of local variables required for the other threads by observing `Read` operations issued by them. Declare each such variable using `Publish`(var) operations.
3. Align `Read` and `Publish` operations in each thread and across the threads, such that (a) there is one `Read` for each `Publish` in each thread, and (b) there is one `Publish` for the value in the remote thread for each local `Read`. This step might require duplicating some calls to achieve perfect alignment, and/or redistribution of the inputs to reduce *conflicts*, i.e., when aligned `Read` requests from different threads need different variables from the same thread. Replace `Read-Publish` tuples with `shuffle`() calls.

5.2.1 Example: 1D k-stencil

We now illustrate this scheme using a 1D k-stencil kernel. We then apply the same principles to the finite field multiplication in Section 6.3.

1D k-Stencil Given an input array $a_0, \ldots, a_{n-1}$, the output of a k-stencil kernel is an array $b_0, \ldots, b_{n-1}$ such that $b_i = \frac{\sum_{j=i-k}^{i+k} a_i}{2k+1}$, assuming $a_i = 0$ for $i < 0$ or $i \geq n$. k is also called a *window size*. Note that each input element is read $2k + 1$ times during computation. Thus, any implementation must cache the input in order to exploit data reuse.

In what follows we use $k = 1$ for clarity, and remind that $\mathcal{W}$ =32 threads per warp.

Shared memory implementation We consider the following implementation: (1) copy input from global memory into a temporary array in shared memory by using all the $\mathcal{TB}$ threads; (2) wait until the input is fully stored in shared memory; (3) compute one output element; (4) store the results in global memory.

We follow the register cache methodology suggested above to eliminate shared memory accesses.

Step one: Identify warp inputs Given that i is the index of the output element computed by thread 0 in a warp, the warp calculates the output elements $i, \ldots, i + 31$, and depends on 34 input elements $i - 1, \ldots, i + 32$, denoted as `input` array.

Step two: Determine input distribution We use a round-robin distribution of `input` arrays among the threads, as illustrated in Figure 5.1. In this scheme, `input[i]` is assigned to thread j=$i \mod 32$, where j is the thread index in the warp. Thread 0 and thread 1 each store two elements, while all the other threads store only one. We denote the first cached element as $r[0]$ and the second as $r[1]$. Observe that this distribution scheme mimics the data distribution across banks of shared memory.

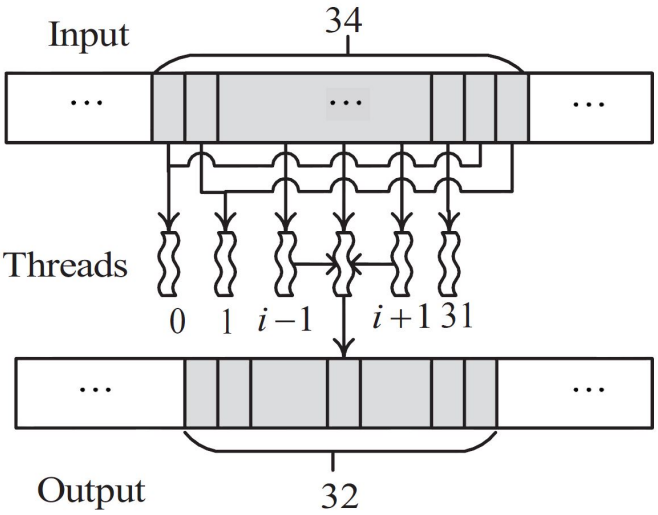


Figure 5.1: Input distribution in 1-stencil computation

Tid	0	1	2-29	30	31
Iteration 1			R(0,i) P(0)		
Iteration 2	R(0,1) P(1)	R(0,2) P(0)	R(0,$i+1$) P(0)	R(0,31) P(0)	R(1,0) P(0)
Iteration 3	R(0,2) P(1)	R(0,3) P(1)	R(0,$i+2$) P(0)	R(1,0) P(0)	R(1,1) P(0)

Table 5.1: Read (R) and Publish (P) operations in each iteration of the 1D 1-stencil computation. Tid denotes the thread index in a warp.

Step three: Communication and computation We identify three communication phases – one for each input element read by each thread. Table 5.1 lists all `Read` (R) and `Publish` (P) operations performed by each thread. `Read`(i, j) indicates a read from thread j of its element $r[i]$. The first communication phase is entirely local, and provided for clarity.

We now merge Publish-Read tuples into `shuffle`(). At this point computations in a warp do not use shared memory. All that remains is to efficiently compute thread and register indexes in the `shuffle`() calls while avoiding divergence.

The complete implementation is in Listing 1.

5.2.2 Analysis

Bank conflicts and `shuffle`() conflicts One of the main challenges of the register cache design is to transform the `Publish` and `Read` operations into `shuffle`() calls. In particular, if there are two or more threads performing
`Read`(var, tid), such that tid is the same and var is different, this is called a *conflict*, since thread tid may fulfill these requests only in multiple `Publish` calls.

A natural question is whether such register cache conflicts are more likely than the conflicts in shared memory in the original implementation. We argue that this is not

Listing 1 1-Stencil implementation using the register cache.

```
#define REGISTER_ARRAY_SIZE 2
#define FILTER_SIZE 1
__global__ void kstencilShuffle(
        int* in,
        int* out,
        int size){
    int threadInput[REGISTER_ARRAY_SIZE];
    int threadOutput = 0, reg_idx, tid_idx;
    int lindex = threadIdx.x & (WARP_SIZE - 1);
    int gindex =
        threadIdx.x + blockIdx.x * blockSize.x;
// PREFETCH. note: in is padded by FILTER_SIZE
    int lowIdx = gindex - FILTER_SIZE;
    int highIdx = lowIdx + WARP_SIZE;
     threadInput[0] = input[lowIdx];
     threadInput[1] = input[highIdx];

//First iteration - data available locally
    threadOutput+=threadInput[0];

//COMMUNICATE + COMPUTE
    reg_idx=(lindex==0)? 1 : 0 ;
    tid_idx=(lindex+1) & (WARP_SIZE -1);
    threadOutput+=
        __shfl(threadInput[reg_idx],tid_idx);

//COMMUNICATE + COMPUTE
    reg_idx =
        (lindex == 0 || lindex == 1) ? 1 : 0 ;
    tid_idx = (lindex+2) & (WARP_SIZE -1);
    threadOutput+=
        __shfl(threadInput[reg_idx],tid_idx);
    output[gindex] = threadOutput / FILTER_SIZE;
}
```

the case. Consider the round-robin input distribution we used in the k-Stencil example. This distribution mimics the distribution of data across the banks in shared memory, because, to the best of our knowledge, the number of banks in NVIDIA GPUs is the same as the number of threads in a warp. Thus, when using the round-robin distribution, the number of register cache conflicts will be exactly the same as the number of shared memory conflicts.

Moreover, register cache might make it possible to reduce the number of conflicts by using an alternative, application-optimized distribution of inputs. We leave this optimization question for future work.

Performance improvement over shared memory The use of a register cache may significantly improve application performance. The main benefits come from lower latency of `shuffle`() operations versus shared memory accesses [nVi15], and higher bandwidth to registers compared to shared memory [V.].

As an illustration, we compare the performance of shared memory and register cache implementations of the k-Stencil kernel. We find that the register cache implementation achieves 64% higher throughput compared to the shared memory version for input sizes of 2^{27} elements.

Thread coarsening One common technique in program optimizations is *thread coarsening* [A. 13]. This technique increases the number of outputs produced by each thread, and thus enables some of the data to be reused across iterations by storing it in registers.

In the case of the register cache, thread coarsening is sometimes *required* in order to achieve the desired performance improvements. The reason lies in the small number of threads sharing the cache. Since the register cache is limited to the threads of a single warp, only the inputs necessary for the warp threads are prefetched and cached. However, the input reuse might occur *across the warps*. For example, for the $k = 1$-Stencil kernel, the value `array`[0] in warp i is the same as `array`[31] in warp $i - 1$; however, both warps read it from the global memory. Thus, assuming the maximum of 32 warps in a $\mathcal{TB}$, one $\mathcal{TB}$ in a register cache implementation performs $34 \times 32 = 1088$ global memory accesses, which is 6% more than the global memory accesses in a shared memory implementation with the same $\mathcal{TB}$ size. Moreover, the number of redundant memory accesses grows with k, reaching 88% for $k = 16$.

Thread coarsening helps reduce the effect of redundant global memory accesses. In Figure 5.2 we show the performance improvement due to computing more outputs per thread (2,4,8 and 16) for the implementations using register cache and shared memory, for different values of k. We see that the improvement due to thread coarsening is almost negligible for the shared memory version, but it is significant for the register cache. We note that with a single output per thread the shared memory version is actually 1.8-2 times faster than the one using register cache for all k (not shown in the graph). However with two and more outputs per thread, the register cache version is faster.

High data reuse As with any cache, the effect of the register cache is amplified with higher data reuse. Figure 5.3 shows the relative performance of the register cache implementation of k-Stencil over the shared memory implementation for different k, as a proxy for evaluating different amounts of data reuse. The speedup achieved by the register cache is about 10% higher for $k = 15$ than for $k = 1$. Each thread computes 16 outputs.

5.2.3 Limitations

Access pattern known at compile time The register cache design may work only for a shared memory access pattern known at compile time. The main reason is that a thread must `Publish` its data exactly when the other threads need it, which requires static provisioning of the respective
`shuffle`() calls. For memory accesses determined at runtime, such provisioning is impossible.

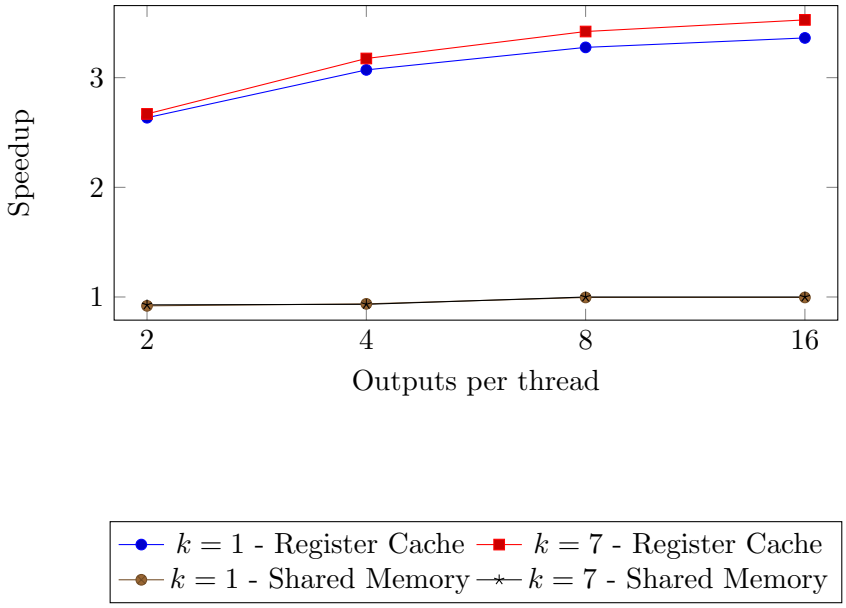


Figure 5.2: Speedup obtained from coarsening in the computation of $1 - Stencil$ and $7 - Stencil$ for register cache and shared memory implementation

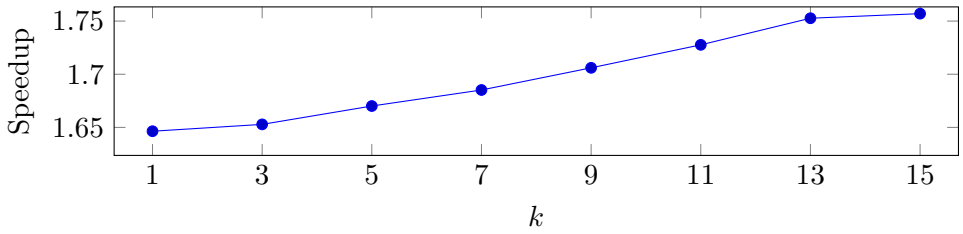


Figure 5.3: Speedup of the shuffle-based k-Stencil implementation over the shared memory-based implementation as a function of k

Register pressure The register cache uses additional registers, and increases register pressure in a kernel. Even though recent NVIDIA GPUs increase the number of hardware registers per $\mathcal{TB}$, the register pressure poses a hard limit on the number of registers available for caching and must be considered to avoid spillage.

Chapter 6

GPU - Finite Field Multiplication

In this chapter the GPU implementation of finite field multiplication in GF(2^n) is given. We present a sequential finite field multiplication algorithm, and then present a parallelization for this algorithm dedicated to the GPU architecture when a large number of multiplications takes place concurrently. Next we show how additional throughput can be achieved by applying our register cache method shown in chapter 5.2 as well as reducing shared-memory consumption and traffic. Finally, we evaluate the performance of our implementation.

6.1 Sequential finite field multiplication

We now provide an efficient algorithm (Algorithm 6.1) for finite field multiplication, one that reduces field multiplication to a small number of polynomial multiplications; it requires a special standard basis, induced by a 2-*gapped polynomial*, defined next. In this section we use the following notation: given a polynomial $h(x) = \sum_{i=0}^{m} h_i x^i$, we define $h_a^b(x) = \sum_{i=a}^{b} h_i x^{i-a}$.

Definition 6.1.1 (2-Gapped Polynomial)**.** A polynomial $r(x)$ is *2-Gapped* if the degree of its second-largest term is at most $\lfloor \frac{\deg(r(x))}{2} \rfloor$, i.e., if $r(x) = x^n + r_1(x)$ with $\deg(r_1(x)) \leq \lfloor \frac{n}{2} \rfloor$.

Algorithm 6.1 performs GF(2^n) multiplication by reducing it to 3 GF (2) [x] ring multiplications. Thus, *the performance of field multiplication is determined almost entirely by the complexity of multiplication of polynomials in the ring of polynomials.* Therefore, in the rest of the work we focus on the problem of fast polynomial multiplication on GPUs.

6.1.1 The CPU `CLMUL` instruction

Finite field arithmetic, in particular GF(2^n) multiplication, has received considerable attention (cf. [J. 86, E. 96]) and has efficient CPU implementations in popular software

Algorithm 6.1 Multiplication in GF(2^n)

Input:

- $a(x), b(x)$ of degree at most $n-1$ in $\mathbb{F}_2[X]$.
- $r(x) = x^n + r_1(x)$, 2-gapped polynomial in $\mathbb{F}_2[X]$ of degree n.

Output: $h(x) = (a(x) \odot b(x)) \mod r(x)$

1: $h(x) \leftarrow a(x) \odot b(x)$
2: $h(x) \leftarrow h_0^{3n/2-1}(x) \oplus h_{3n/2}^{2n-1}(x) \odot r_1(x) \odot x^{n/2}$
3: $h(x) \leftarrow h_0^{n-1}(x) \oplus h_n^{3n/2}(x) \odot r_1(x)$
4: **return** $h(x)$

Algorithm 6.2 Naïve polynomial multiplication

Input:

$a(x), b(x)$ of degree at most $n-1$.

Output: $c(x) = a(x) \odot b(x)$

1: **for** $i = 0, \ldots, n-1$ **do**
2: $\quad c_i \leftarrow 0$
3: $\quad$ **for** $j = 0, \ldots, i$ **do**
4: $\quad\quad c_i \leftarrow c_i \oplus a_j \odot b_{i-j}$
5: **for** $i = n, \ldots, 2n-2$ **do**
6: $\quad c_i \leftarrow 0$
7: $\quad$ **for** $j = i, \ldots, 2n-2$ **do**
8: $\quad\quad c_i \leftarrow a_{n-1+i-j} \odot b_{j-n+1}$
9: **return** $c(x) = \sum_{i=0}^{2n-2} c_i \cdot x^i$

libraries like NTL [V. 03] and MPFQ [1]. Moreover, in large part because of the importance of GF(2^n) multiplication, Intel introduced in 2010 a dedicated CPU instruction set extension `CLMUL`, which performs GF$(2)[x]$ ring multiplication of polynomials of degree up to 64 in 7–14 cycles [Fog16].

Both NTL and MPFQ use this dedicated instruction. This instruction can be used to multiply polynomials of higher degree, thereby supporting GF(2^n) multiplication for values $n > 64$ (cf. [C. 12] for one such implementation).

6.1.2 Sequential polynomial multiplication

The complexity of polynomial multiplication has been extensively studied. The number of bit operations performed by naïve Algorithm 6.2 is $O(n^2)$. More sophisticated algorithms by Karatsuba [KO63] and by Schonhage and Strassen [SS71, D. 91] are asymptotically faster, requiring $O(n^{\log_2 3})$ and
$O(n \log n \log\log n)$ bit operations, respectively.

In this work we use the naïve Algorithm 6.2 because it is the fastest for polynomials of degrees below 1000 [M. 05] and its simplicity makes it a prime starting point for study.

[1] http://mpfq.gforge.inria.fr/doc/doc.html

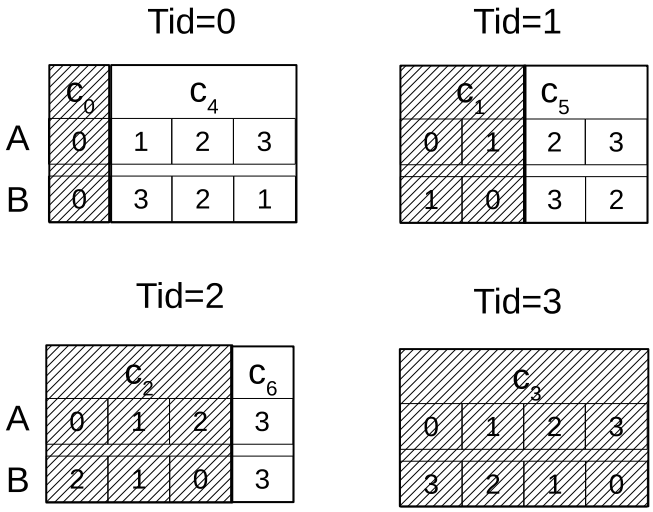


Figure 6.1: Illustration of the access pattern of the multiplication algorithm for $\mathrm{GF}(2^4)$ with $\mathcal{W}$ =4. Each frame encloses the indexes of rows in A and B accessed for computing the respective rows c_i specified on the top. Tid denotes the thread index in the warp.

The following simple equation, which explicitly computes coefficients of the output polynomial, will be used later to balance work in the GPU.

$$c_k = \begin{cases} \sum_{i=0}^{k} a_i \cdot b_{k-i} & k \leq n-1 \\ \sum_{i=k}^{2n-2} a_{n-1+k-i} \cdot b_{i-n+1} & k > n-1 \end{cases} \tag{6.1}$$

6.2 Parallel polynomial multiplication

We consider the problem of performing multiplication of a large number of pairs of polynomials.

A naïve, purely data-parallel approach is to assign a single multiplication of two polynomials to one thread. Here, each polynomial of degree $n-1$ is represented as a bit array of size n, where the i^{th} element represents the coefficient of x^i in the polynomial.

This solution is highly inefficient, however. On a platform with $\mathcal{B}$-bit registers and ALUs, performing single-bit operations uses only $1/\mathcal{B}$ of the computing capacity. We therefore develop an alternative algorithm which eliminates this inefficiency.

6.2.1 Bit slicing

We reorganize the computation such that one thread performs bit-wise operations on $\mathcal{B}$ bits in regular registers, effectively batching multiple single-bit operations together. This technique, which packs multiple bits for parallel execution, is often called *bit-slicing* [Wik].

To employ bit-slicing for polynomial multiplication, we first introduce a new data structure, a *chunk*, to represent multiple polynomials, and then reformulate the multiplication algorithm using chunks.

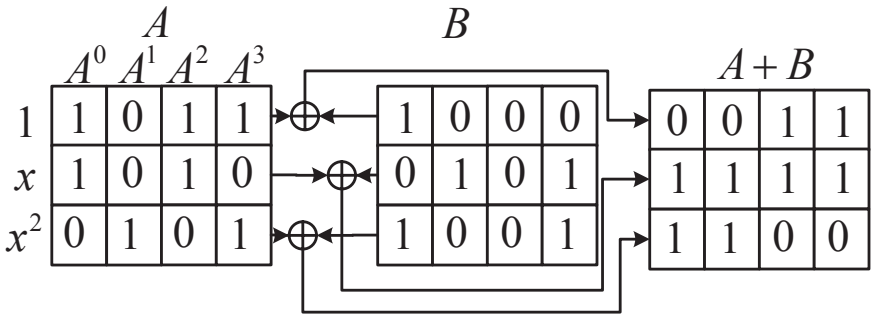


Figure 6.2: Polynomial addition in 4-bit chunks. Computing the output chunk requires 3 bit-wise XORs, each performing 4 concurrent $\oplus$ operations.

Definition 6.2.1 (Chunk)**.** A chunk is an $n \times \mathcal{B}$ matrix M of bits that represents a set of $\mathcal{B}$ polynomials $P(i)$, $i \in \{0, \ldots, \mathcal{B}-1\}$ of degree less than n. We denote the j^{th} column in M by M^j, and the i^{th} row by M_i. M^j represents the coefficients of the j^{th} polynomial in the set. In other words, $A(i) = \sum_{i=0}^{n-1} M_i^j x^i$.

To explain how to compute using chunks, we first consider polynomial addition. It is easy to see that it can be performed by bit-wise XOR of the respective rows of the input chunks A and B. Thus, a single $A_i \oplus B_i$ computes the i^{th} coefficients for all $\mathcal{B}$ output polynomials at once. Figure 6.2 shows two input chunks A, B, and the chunk representing their sum $A \oplus B$. Each chunk represents 4 polynomials of degree 3. For example, A^1 represents polynomial x^2. Figure 6.2 also shows an example of polynomial addition using chunks, assuming $\mathcal{B} = 4$.

Similarly, it is straightforward to extend the single-bit polynomial multiplication Algorithm 6.2 to use chunks. This is done by replacing the references to individual bits in lines 2,4,6 and 8 with the references to chunk rows, and replacing single-bit operations with bit-wise operations.

6.2.2 Parallel polynomial multiplication using chunks

We show how to parallelize chunk-based polynomial multiplication. We seek a parallel algorithm that enables efficient, divergence-free execution by the threads of a single warp, which is the key to high performance on GPUs.

A simple parallelization whereby one thread computes one row in the output chunk is inefficient due to divergence among the threads. As we see from Eq. 6.1, different coefficients in the output polynomial require different numbers of computations. For example, computing the coefficient of x^2 requires only three $\oplus$ operations, while computing the one for x^3 requires four. Thus, different threads in a warp would perform different numbers of operations, resulting in divergence.

The key to achieve load-balanced execution among the threads is to realize that *pairs* of coefficients require exactly the same number of computations *in total*, as we show below.

Denote by $Add(k)$ and $Mul(k)$ the number of $\oplus$ and $\odot$ operations respectively to compute the k^{th} coefficient in the output polynomial. From Eq. 6.1 we derive that $Add(k) = \min\{k, 2n-2-k\}$, $Mul(k) = \min\{k+1, 2n-2-k+1\} = Add(k)+1$. Therefore $Add(k)$ and $Mul(k)$ are symmetric around $n-1$. Consequently, for each $0 \leq k < n$ $Add(k) + Add(k+n) = n-2$, $Mul(k) + Mul(k+n) = n$

We conclude that the number of computations needed to compute both coefficients k and $k+n$ together is exactly the same for all k. Therefore, allocating such pairs of coefficients to be computed by each thread will balance the load perfectly among the threads. Note that computations always interleave bitwise $\oplus$ and $\odot$ operations; therefore there is no divergence as long as the number of such operations in all threads is the same.

In summary, our parallel polynomial multiplication algorithm allocates each thread in a warp to compute one or more pairs of rows $(k, k+N)$ in the output chunk. Each thread computes the coefficients of $\mathcal{B}$ polynomials at once, thanks to bit-slicing.

We illustrate the execution of the algorithm for $\mathrm{GF}(2^4)$ and $\mathcal{W} = 4$ threads per warp as an example in Figure 6.1.

Implementation The implementation closely follows the algorithm. We dedicate one warp to compute $2\mathcal{W}$ rows in the output chunk C. All the rows in the input are accessed by all the threads, and therefore they are prefetched into shared memory. Figure 2 lists the implementation for a single warp, assuming $\mathcal{W}$ =N=32. For clarity we split the implementation into two separate loops (line 15 and 22), each computing one output row. This leads to divergence in practice, so in the real implementation these two loops are merged.

Limitations The algorithm achieves divergence-free execution when invoked for polynomial multiplication in $\mathrm{GF}(2^N)$ when $N|\mathcal{W}$, i.e., 32, 64, 96. We leave the question of efficient multiplication of polynomials of other degrees to future work.

6.3 Polynomial multiplication using register cache

In this section we apply the register cache methodology presented in Section 5.2 to speed up ring multiplication (Listing 3) and compare it (here and later) to the less efficient and simpler shared memory implementation (Listing 2). To describe the register cache optimizations, we focus on a single warp performing multiplication of polynomials of degree n=$\mathcal{W}$ =32. We then discuss the application of this method to polynomials of higher degree.

We start with the shared memory implementation described in Section 6.2.2.

Step one: Identify warp inputs in shared memory Since each warp is dedicated to the calculation of a single product of two chunks, each warp reads only its input

Listing 2 Multiplication of polynomials of degree 32 in a warp using shared memory.

```
1  __global__ void multiply_shmem(
2          int* A, B, C,
3          int N)
4  {
5      __shared__ int sA[32];
6      __shared__ int sB[32];
7      int output=0;
8      int lindex = threadIdx.x & (WARP_SIZE - 1);
9
10 // PREFETCH
11     sA[lindex]=A[lindex];
12     sB[lindex]=B[lindex];
13     __syncthreads();
14
15     for (int i=0;i<=lindex;i++){
16         int a = sA[i];
17         int b = sB[lindex-i];
18         output ^= a&b;
19     }
20     C[lindex]=output;
21     output=0;
22     for (int i=lindex+1;i<N;i++){
23         int a = sA[i];
24         int b = sB[N-1+lindex-i];
25         output ^= a&b;
26     }
27     C[lindex+N]=output;
28 }
```

chunks.

Step two: Distribute inputs among warp threads The rows in chunks are distributed in a round-robin fashion across the warp threads. For each of the two input chunks, thread ℓ stores all the chunk rows t such that $\ell \equiv t \mod w$. Conveniently, since $\mathcal{W} = n$, thread i stores rows A_i and B_i of the respective chunks.

Step three: Split the algorithm into communication and computation steps Each thread communicates with the other threads to obtain the operands of each $\odot$ operation. Therefore, each $\odot$ is a computation step that is preceded by a communication step in which the operands are received. We refer to two such steps together as *an iteration*, because they correspond to one iteration of the loops in lines 15 and 22 in Listing 2.

We first determine the data accessed by each thread. We derive this from the accesses to shared memory in lines 16-17 and 23-24 in Listing 2. Due to the round-robin data distribution we use, and since the number of rows in each chunk equals the number of threads, the indexes in shared memory coincide with the warp indexes of the threads holding the data.

Now we derive which data must be published by each thread in each iteration. Figure 6.1 is useful to reason about this. We see that the value of A_i, stored in thread i, is needed by all the threads only in iteration i, and hence each thread must publish it in iteration i.

B_i, however, is read by different threads in different iterations. For example, B_0 is

Listing 3 Multiplication of polynomials of degree 32 in a warp using the register cache.

```
__global__ void multiply_reg_cache(
        int* A, B, C,
        int N)
{
    int a_cached, b_cached, output=0;
    int lindex = threadIdx.x & (WARP_SIZE - 1);

// PREFETCH
    a_cached=A[lindex];
    b_cached=B[lindex];

    for (int i = 0 ; i < N ; i++)
    {   //COMMUNICATE
        int a = __shfl(cached_a,i);
        int b = __shfl(cached_b,lindex-i);
    //COMPUTE
        if (i <= lindex) output ^= a&b;
    }
    C[lindex]=output;
    output=0;
    for (int i = 0; i < N ; i++){
        int a = __shfl(cached_a,i);
        int b = __shfl(cached_b,N-1+lindex-i);

        if (i > lindex) output ^= a&b;
    }
    C[lindex+N]=output;
}
```

used by thread 0 in the first iteration, thread 1 in the second, and so on. Thus, thread i must publish B_i in each iteration.

The computation in each iteration remains the same as in the shared memory version.

Replacing each communication step with shuffles To use `shuffle`(), we must align each `Read` and `Publish` operations in each communication step. To simplify, we consider the case in which we first align all accesses to B and then to A.

Aligning accesses to B is straightforward, because (1) each thread publishes its single cached value and reads one value in every iteration, and (2) no two threads require two different values at once from the same thread (which would result in a conflict).

The accesses to A cause a problem, because each thread publishes only in one iteration, but reads in each iteration. The solution is to simply duplicate the `Publish` operation to each iteration, even though it is redundant.

The complete algorithm is presented in Listing 3, side-by-side with the shared memory implementation in Listing 2 for comparison.

6.4 Extending to polynomials of larger degrees

We now extend the register cache-based multiplication implementation described in the previous section to polynomials of larger degrees. Doing so requires us to cope with the challenge of limited register space.

The shared memory algorithm in Listing 2 can be extended to up to $n = 1024$ by adding more warps, each using the same code structure. The register cache, however, is

applicable only within a single warp. Therefore such a simple extension does not work for the optimized algorithm.

However, extending the register cache for higher degree polynomials is problematic in other ways as well. Caching these large polynomials requires more register space. Thus, at a certain threshold n_0, high register pressure results in register spillage to global memory, thereby rendering the register cache method described above inapplicable. We found empirically that the threshold is $n_0 = 64$.

In order to efficiently multiply polynomials of degree $n > 64$, we develop a hybrid solution that uses the efficient register cache-based implementation for multiplying polynomials of lower degree. The idea is to use the lower-degree multiplication as a building block for multiplying polynomials of higher degrees, at the expense of employing shared memory.

The full description of this algorithm is omitted for lack of space. But we now explain the main idea behind it, by showing how to multiply degree-64 polynomials using multiplication of degree-32 polynomials as a building block.

Let $a(x) = \sum a_i x^i$ and $b(x) = \sum b_i x^i$ be two polynomials of degree 64 that we wish to multiply. Denote the efficient procedure for multiplying two polynomials of degree 32 by `mult32()`. We can represent $a(x) = a_0(x) + x^{32}a_1(x)$, where $a_0(x) = \sum_{i=0}^{31} A_i x^i$ and $a_1 = \sum_{i=32}^{63} x^i$. Observe that a_0 and a_1 are two polynomials of degree at most 31. Using the same representation for $b(x)$, we obtain $a(x) \odot b(x) = (a_0(x) + x^{32}a_1(x)) \odot (b_0(x) + x^{32}b_1(x)) =$
$\texttt{mult32}(a_0(x), b_0(x)) + x^{32}\texttt{mult32}(a_1(x), b_0(x)) +$
$x^{32}\texttt{mult32}(a_0(x), b_1(x)) + x^{64}\texttt{mult32}(a_1(x), b_1(x))$.

There are many possible implementations of this idea and those we are aware of use shared memory. We choose to implement one such solution that uses two warps. The first warp computes $\texttt{mult32}(a_0, b_0)$ and $\texttt{mult32}(a_1, b_0)$, and the second one computes $\texttt{mult32}(a_0, b_1)$ and $\texttt{mult32}(a_1, b_1)$. Since the input is reused across the warps, it is stored in shared memory. In addition, each warp stores its output in shared memory, so the two warps can combine the results of $\texttt{mult32}(a_0, b_1)$ and $\texttt{mult32}(a_1, b_0)$.

We use the same principle to implement multiplication for polynomials of higher degree.

6.4.1 Performance comparison of the different designs

We would like to compare the relative speedup offered by the hybrid algorithm over the purely shared memory implementation, and over the implementation that uses the register cache only. Comparing these three designs is possible only for $n \leq 64$ because, as mentioned, register pressure in the register cache version results in register spillage.

In our implementation, the naive shared memory version runs in two warps. The hybrid `mult32`-based implementation uses the `mult32` function internally, and uses shared memory to share input and intermediate outputs between warps. Finally, the

Listing 4 Ring multiplication of just of large degrees using ring multiplication of chunks of degree 32 as a building block.

```
// Rounds up N to the smallest multiplication
// of 32 larger than or equals to N.
#define ROUNDED(N) ((((N)+31)/32)*32)

// Brief:     Performs ring multiplication of chunks A and B.
// Input:     A,B - Chunks of degree N stored using ROUNDED(N)
//             entries in shared memory each.
//             B is stored right after A in shared memory.
//            tempChunk - A chunk of degree 64, stored in shmem.
//             unique chunk for each warp.
//            myIdxInGroup - The index of a thread within all threads
//             that cooperate in the multiplication of A and B.
//            myIdxInWarp - Index of the thread within its warp.
//            warpInGroup - Index of this thread's warp within
//             all warps that cooperate in the multiplciation of
//             chunks A and B.
// Output:    Ring multiplication of A and B stored in chunk C.
template<unsigned int N>
__device__ inline void finiteFieldMultiply(
    unsigned int A[ROUNDED(N)],
    unsigned int B[ROUNDED(N)],
    unsigned int C[2 * ROUNDED(N)],
    unsigned int temp[64],
    unsigned int myIdxInGroup,
    unsigned int myIdxInWarp,
    unsigned int warpInGroup)
{
    for (unsigned int i = 0 ; i < ROUNDED(N)/32 ; ++i)
    {
        // Each warp multiplies two chunks of degree 32:
        //         1) In A: entries (32*warpInGroup,...,32*warpInGroup + 31)
        //        2) In B: entries (32*i,...,32*i + 31)
        // Output is written to tempChunk.
        multiply32Ring(tempChunk, &A[32 * warpInGroup], &B[32 * i]);

        //
        C[32*(i+warpInGroup) + myIdxInWarp] ^= tempChunk[myIdxInWarp];
        __syncthreads();

        // Then write the upper 32 entries to the shared memroy.
        C[32*(i+warpInGroup) + myIdxInWarp + 32] ^= tempChunk[myIdxInWarp + 32];
    }
}
```

optimized degree-64 multiplication uses register cache natively, without shared memory. In this implementation each thread stores 4 input coefficients and produces 4 outputs.

The results of the comparison are presented in Table 6.1 and demonstrate the benefits of using register cache. We observe that the shared memory (shmem) implementation is about 3.5 times slower than the one using register cache (rcache). The hybrid version (mult32) achieves 2.6 times faster execution over shmem, and about 30% slower than the optimal rcache version.

These results also indicate that the best building block for the hybrid algorithm is the multiplication kernel of the largest degree that fits in the register cache. Therefore, we use n-64 polynomial multiplication and evaluate its performance in Section 8.

6.4.2 Application to larger fields

The shared memory based multiplication requires $16n$ bytes of shared memory. In a GPU with up to 48KB of shared memory per $\mathcal{TB}$ for full occupancy (as NVIDIA

Version	Throughput (mult/s $\times 10^9$)	Shared memory accesses	Reg/Thread
shmem	1.04	16384	25
mult32	2.7	512	30
rcache	3.6	0	32

Table 6.1: Performance of three different implementations of 64-degree polynomial multiplication.

Titan-X), we are limited to fields of size $< 2^{3072}$. With the register cache we use half the amount of shared memory, and therefore can implement multiplication in fields as large as $\mathrm{GF}\left(2^{6144}\right)$.

However, we do not implement it for fields larger than $\mathrm{GF}\left(2^{2048}\right)$. For larger fields the hybrid algorithm outlined here with asymptotic running time $O(n^2)$ becomes relatively inefficient when compared to the more sophisticated Karatsuba algorithm, as detailed in Section 8.

6.4.3 Using shared memory only for the output

We now present another optimization intended to reduce the amount of shared memory allocated for a ring multiplication. Therefore, this optimization reduces the amount of shared memory allocated to finite field multiplication as well. This optimization uses the degree-32 ring multiplication of chunks based on register cache and does not work when a degree-32 ring multiplication based on shared memory is used.

In the previous section we showed that to perform polynomial multiplication of chunks of degree n using the degree 32 polynomial multiplication as a building block. We have also stated there that the degree 32 polynomial multiplication can be performed either by shared memory based multiplication or using the shared-memory free version that is shuffle based. In both ways the presented implementation stores in shared memory 2 chunks of degree n and an additional chunk of degree $2n$ that stores the output of the multiplication which is in total $\frac{n\mathcal{B}}{2}$ bytes of memory.

In this section we show how we reduce the amount of shared memory when using the shuffle-based version of the degree 32 polynomials multiplication by half. We recall that this version is shared memory free and utilizing the characteristic is crucial in the improvement.

The multiplication will be in place, which means that the output of the multiplication will be written straight into where the input was written in shared memory.

First, since the output is degree $2n$ chunk and we wish to write it exactly where the input resided in the beginning of the algorithm we the chunks in shared memory that store the inputs will be consecutive in the memory. This requirement is reasonable and can be easily programmed, we will not discuss this requirement along the rest of the algorithm and assume that chunk a is followed in memory by chunk b.

Recall the algorithm in listing 4 in which the used degree 32 polynomial multiplication building block is the shuffle based one. In this algorithm each warp that participates in the multiplication of two chunks stores, distributively, a chunk of degree 32. We know that all entries in the input chunk a will be stored in one of these warps and we know that after this single reading from a no additional reads will be done from A at all along the algorithm. Therefore, we can use A to store the lower half of the output (coefficients of $x^0, \ldots, x^{n-1}$) and B to store the upper half of the output. In the first step of the iteration, we take the first 32 entries of b and load them distributively into the registers of all warps. At this moment the first 32 entries of b will never be read again. Therefore the first $n + 32$ at this moment are free for output to be written to them. At this point each warp multiplies its part of a by the first 32 entries of b that are stored in its registers and the total output of all warps exactly fits into the first $n + 32$ entries of the output, the result will still be written in two phases as described in previous section, each warp with 64-degree polynomial it wants to add to the output first adds the lower 32 entries of the output it stores to the correct entries in the shared memory, then a barrier is applied and then the upper 32 entries each warp holds will be added to the correct entries in the shared memory. The full implementation is given in listing 5.

Notice that if we didn't use the shuffle-based communication, we would be able to write our result to c without overriding entries from a subchunk of b that we still need for the multiplication since we don't store them in our registers. We can also use this ring multiplication as a building block in the ring multiplication based finite field multiplication as described in algorithm 6.1 to cut in half the shared memory consumption of the finite field multiplication as well.

Listing 5 In-place ring multiplication of chunks.

```
// Rounds up N to the smallest multiplication
// of 32 larger than or equals to N.
#define ROUNDED(N) ((((N)+31)/32)*32)

// Brief:     Performs ring multiplication of chunks A and B.
// Input:     A,B - Chunks of degree N stored using ROUNDED(N)
//             entries in shared memory each.
//             B is stored right after A in shared memory.
//            myIdxInGroup - The index of a thread within all threads
//             that cooperate in the multiplication of A and B.
//            myIdxInWarp - Index of the thread within its warp.
//            warpInGroup - Index of this thread's warp within
//             all warps that cooperate in the multiplciation of
//             chunks A and B.
// Output:    Result is stored as a chunk in the same memory
//            A and were stored in.
template<unsigned int N>
__device__ inline void finiteFieldMultiply(
    unsigned int A[ROUNDED(N)],
    unsigned int B[ROUNDED(N)],
    unsigned int myIdxInGroup,
    unsigned int myIdxInWarp,
    unsigned int warpInGroup)
{
    unsigned int my_a;
    unsigned int my_b;
    unsigned int my_c[2];

    // READ step: Each warp reads coefficients
    //         (warpInGroup * 32,..., warpInGroup * 32 + 31)
    //        into registers.
    my_a = A[warpInGroup * 32 + myIdxInWarp];

    // All entries of A are stored in registers, we nullify
    // the shared memory in which A was stored.
    A[warpInGroup * 32 + myIdxInWarp] = 0;

    for (unsigned int i = 0 ; i < ROUNDED(N)/32 ; ++i)
    {
        for (unsigned int j = 0  ; j < 4 ; ++j)
        {
            my_c[i] = 0;
        }

        // READ step - Reading coefficients (32*i,...,32*i + 31).
        my_b= B[32*i + myIdxInWarp];
        __syncthreads();

        // First warp nullifies the entries of B that
        // were read in this iteration.
        // They will now be used to store the output.
        if(warpInGroup == 0)
        {
            B[32*i + myIdxInWarp] = 0;
        }
        __syncthreads();

        // Each warp multiplies to chunks of degree 32 it stores.
        // Output is stored in my_c.
        // Multiplication is shuffle based.
        multiply32Shuffle(my_c, my_a, my_b);

        // Each warp distributively stores in my_c a chunk
        // of degree 64.
        // To avoid write access collisions, first add
        // lower 32 entries to the result in shared
        // in shared memory and syncrhonize.
        A[32*(i+warpInGroup) + myIdxInWarp] ^= my_c[0];
        __syncthreads();

        // Then write the upper 32 entries to the shared memroy.
        A[32*(i+warpInGroup) + myIdxInWarp + 32] ^= my_c[1];
    }
}
```

Chapter 7

Implementation of the FFT algorithm on GPU

In this section we will depict in detail an efficient parallel implementation of algorithm 3.5 on the GPU architecture. First, we introduce the general outline of the implementation and the main phases that compose it, then we will elaborate the implementation of each phase separately.

7.1 Outline of the Implementation

Algorithm 3.5 is recursive. Unfortunately, recursion would have severely impaired the implementation's performance. Instead the algorithm's implementation begin with a series of d *splitting* iterations within a loop. At the first iteration the input is a polynomial g where $2^d \leq \deg(g) < 2^{d+1}$ represented as a sequence of $\frac{2^{d+1}}{\mathcal{B}}$ chunks with k^{th} element in the series of chunks is a finite field element that represents g_{k-1} - the coefficient of x^{k-1} in g. In figure 7.1 an array of chunks G is presented. At its first entry, elements $g_0, \ldots, g_{\mathcal{B}-1}$ reside. At the second entry $g_{\mathcal{B}}, \ldots, g_{2\mathcal{B}-1}$ and so on.

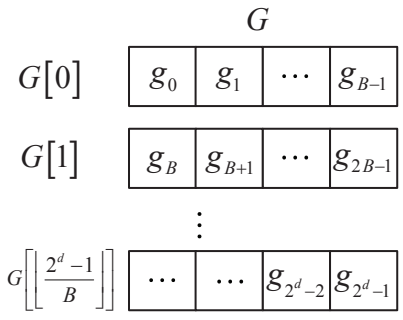


Figure 7.1: Storage of Coefficients of Input FFT Polynomial in Chunks

We will denote the input polynomial g as $g_{0,0}$ which along the first iteration will be processed as described in algorithm 3.5 and will be detailed in this section. At the end of algorithm 3.5 the polynomial is split into two polynomial on which FFT is computed

recursively. In our implementation, at the end of the iteration the polynomial is split into two polynomials $g_{1,0}, g_{1,1}$ where $2^{d-1} \leq deg(g_{1,i}) < 2^d$ and FFT is evaluated on both of them over the same subspace as described in algorithn 3.5. Those evaluations will be in parallel.

The next iterations are implemented in the same manner, the input to the i^{th} iteration will be a series of polynomials $g_{i,0}, \ldots, g_{i,2^i-1}$ where $2^{d-i} \leq deg(g_{i,j}) < 2^{d-i+1}$that are represented as an array of $\frac{2^{d+1}}{\mathcal{B}}$ chunks. The polynomials will be stored in the same array of chunks, such that the first 2^{d-i+1} elements in will represent the first polynomial, the following 2^{d-i+1} will represent the second polynomial and so on. The GPU will process in parallel $g_{i,0}, \ldots, g_{i,2^i-1}$ as described in 3.5 in 3 steps,

1. Shift phase (Algorithm 3.5, step 2) in which we take the polynomial $g_{i,j}(x)$ and a basis element β and calculate the new polynomial $s_{i,j}(x) = g_{i,j}(\beta x)$.

2. Taylor Expansion Phase (Algorithm 3.5, step 3) the which we calculate the Taylor Expansion of the polynomial at $\left(x^2 - x\right)$. The output to this phase is in the same structure as the input, composed of pairs of elements where the k^{th} pair is the linear function $a_{i,j,k} + b_{i,j,k} \cdot x$ which is the coefficient of $\left(x^2 - x\right)^i$ in the calculated Taylor Expansion and $s_{i,j} = \sum_{k=0}^{2^{i-1}-1} \left(\alpha_{i,j,k} + \beta_{i,j,k} \cdot x\right) \cdot \left(x^2 - x\right)^i$

3. Shuffle Phase (Algorithm 3.5, step 4) in which we take the Taylor Expansion calculated in the previous step and calculate from it the two new polynomials $g_{i+1,2j}, g_{i+1,2j+1}$ by performing the *shuffle permutation* on the input, taking all evenly indexed elements (i.e. all $\alpha_{i,j,k}$) and putting them, in order, at the first half of the output so they will represent the polynomial $g_{i+1,2j}$ and the oddly indexed element (i.e. all $b_{i,j,k}$) and putting them, in order, at the second half of the output so they will represent the polynomial $g_{i+1,2j+1}$.

At the end of the iteration, from each polynomial $g_{i,j}$ two new polynomials are created,$g_{i+1,2j}$ and $g_{i+1,2j+1}$. These polynomials will be part of the input of the next iteration.

At the end of all iterations we have 2^d linear functions (i.e polynomials of degree at most 1). $g_{d,0}, \ldots, g_{d,2^{d-1}}$, then we will perform in parallel the linear evaluation phase (Algorithm 3.5, step 1) for $g_{d,j}$. This evaluation will give an output denoted by $e_{d,j}$ the evaluation of linear function $g_{d,j}$ over a subspace with two elements $\{0, u\}$ where $e_{d,j}$ is represented as two elements. First the evaluation of the linear function over 0 and the over u. The output of the whole phase is represented with $\frac{2^{d+1}}{\mathcal{B}}$ chunks composed as the concatenation of all evaluations.

Now we will iteratively "fold" the recursion, with another loop of d *merge* iterations, the input to each iteration is a series of evaluations $e_{i,0}, \ldots, e_{i,2^i-1}$ which are the evaluations of $g_{i,0}, \ldots g_{i,2^i-1}$ and in parallel for each consecutive pair of evaluations $e_{i,2j}, e_{i,2j+1}$ wes will calculate $e_{i-1,j}$ in the merge phase (Algorithm 3.5, step 10).

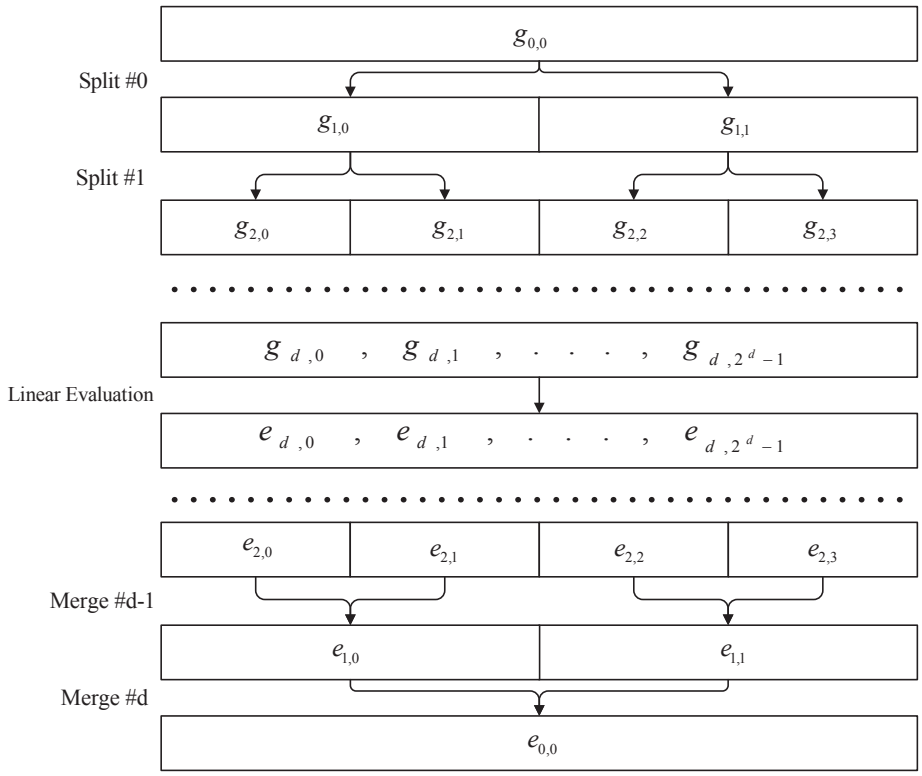


Figure 7.2: Outline of the FFT Algorithm

In figure 7.2 an outline of the algorithm as described above is presented, the flow of the algorithm is described in figure from the top to the bottom and in figure 7.3 a single split iteration is depicted.

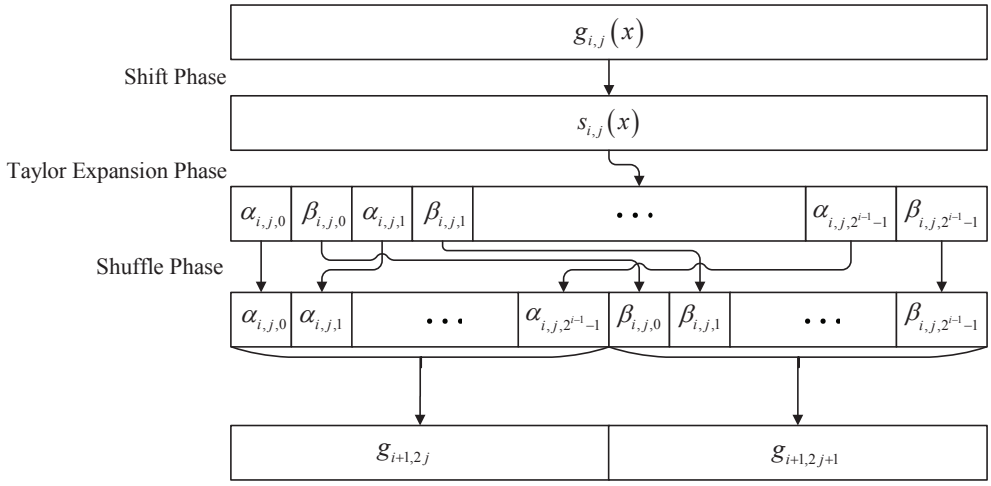


Figure 7.3: Outline of a Single Split Iteration

Along the algorithm there are many uses of affine subspaces composed of subspaces B and G and affine shifts s_B and s_G (using the notation from 3.5). The shifts and the elements spanning the subspaces will be kept in a designated array and since they are

not a function of the input polynomial they will be pre-calculated, assuming the affine space is known beforehand. We will also maintain for each subspace an array of chunks such that the i^{th} finite field element in the array of some subspace B is $B[i]$.

Notice that also D and s_D are used, but only as inputs to the recursive calls so each D and s_D are in turn the B and s_B of the next recursive step.

In the following sections we will depict the details of implementation of each phase.

7.2 Set Up for GPU

At the beginning the algorithm gets as input an array of finite field elements. As previously stated, assume that each finite field element $e = \sum_{i=0}^{n-1} a_i x^i$ in GF(2^n) is represented as an array of $n/8$ bytes where j^{th} bit in i^{th} entry represents a_{8i+j}.

Our goal in the setup phase is to change the representation of the given polynomial into an array of chunks. The kernel that does that assigns a warp for each $\mathcal{B}$ elements in the input polynomial. So warp i will be responsible for grouping elements $\mathcal{B} \cdot i$ to $\mathcal{B} \cdot (i+1) - 1$ into a single chunk . Thread of index j in warp i will be responsible for writing rows $j, j + \mathcal{W}, \ldots$ of that chunk.

First, in order for the GPU to be able to process the given array, we copy the whole polynomial into the GPU.

The algorithm is quite simple, at the beginning thread t in the warp will copy element $t, t + \mathcal{W}, \ldots$ from the elements which its warp is responsible for into the shared memory into a designated array. After that thread t extracts the i^{th} bit from all elements, writing the i^{th} bit of the j^{th} element into the j^{th} bit of the i^{th} row of the output answer, for $i = t, t + \mathcal{W}, \ldots$ and $j = 0, 1, ..., \mathcal{B} - 1$.

After the algorithm is done, we remove original polynomial from GPU as we won't need it anymore and we will copy to the GPU memory the arrays of chunks for each subspace used along the algorithm as described above.

7.3 Shift Phase

In this phase we are given as input an array of finite field elements stored in chunks, representing a series of 2^t polynomials $g_{t,0}, \ldots, g_{t,j}$ of degree at most $2^d - 1$. At the beginning of the algorithm, $t = 0$ hence we have a single polynomial, at each recursive call each polynomial is split into two polynomials as described in the shuffle phase (section 7.5).

For each polynomial $g_{t,j}(x) = \sum_{i=0}^{2^d-1} a_i x^i$ we have to calculate $s_{t,j}(x) = g_{t,j}(\beta_m x)$ where β_m is an element spanning affine subspace B over which we evaluate $g_{t,j}(x)$. By calculating $g_(\beta_m x)$ we mean having in memory the polynomial $g_{t,j}(\beta_m x)$ represented in chunks, so the i^{th} finite field element will represent the coefficient of x^i of that

polynomial. Notice that,

$$s_{t,j} = g_{t,j}(\beta_m x) = \sum_{i=0}^{2^d-1} a_i(\beta_m x)^i = \sum_{i=0}^{2^d-1} \beta_m^i a_i x^i$$

The new coefficient of x^i is $a_i \cdot \beta_m^i$, so our algorithm will multiply the i^{th} coefficient by β_m^i. As β_m is known beforehand and is independent of the input polynomial, we will precompute an array of chunks A of length $\max(1, \lfloor 2^d/\mathcal{B} \rfloor)$ with 2^d finite field elements such that the i^{th} finite field element stored there will be β_m^i.

If $2^d < \mathcal{B}$ then this array will be composed of a single chunk and the i^{th} coefficient (i.e. β_m^i) will be element $i, i+2^d, \ldots$. For example if $\mathcal{B} = 32$ and $2^d = 8$ then a_0 will be in elements $0, 8, 16, 24$ of the chunk and a_5 will be in elements $5, 13, 21, 29$ of the chunk. In the algorithm it self each warp number i is responsible for multiplying chunk $B[i]$ by chunk $A[i \mod 2^d]$ using the GPU implementation of algorithm 6.1 given at chapter 6.

7.4 Taylor Expansion Phase

In this phase we get as input an array of chunks B representing 2^t polynomials $s_{t,0}, \ldots, s_{t,2^t-1}$ of degree $< 2^d$ each ($d \geq 2$). The goal of this phase is to calculate the Taylor Expansion at $\left(x^2 - x\right)$ of each of the given polynomials using Algorithm 3.4. That is, we would like to find linear functions $h_{i,j,0}(x), \ldots, h_{i,j,2^{d-1}-1}$ such that $s_{i,j}(x) = \sum_{k=0}^{2^{d-1}-1} h_{i,j,k}(x) \cdot \left(x^2 - x\right)^k$ and $h_{i,j,k}(x) = \alpha_{i,j,k} + \beta_{i,j,k} \cdot x$.

The format of the requested output is a sequence of pairs $\alpha_{i,j,k}, \beta_{i,j,k}$, in chunks, ordered by k. The first $\mathcal{B}$ elements will be in the first chunk, the next $\mathcal{B}$ elements will be in the second chunk and so on.

The Taylor Expansion algorithm is implemented as follows. Given a polynomial $g_{i,j}(x)$ of degree $< 2^d$ write it as

$$p(x) = t_0(x) + x^{2^{d-2}} t_1(x) + x^{2^{d-1}} t_2(x) + x^{3 \cdot 2^{d-2}} t_3(x) \quad , \deg(p_i) < 2^{d-2} \tag{7.1}$$

So,

$$T(p, 2^d) \quad = \left(T\left(t_0 + (t_1 + t_2 + t_3)\, x^{2^{d-2}}, 2^{d-1}\right), T\left((t_2 + t_3) + t_3 \cdot x^{2^{d-2}}, 2^{d-1}\right)\right) \tag{7.2}$$

Figure 7.4 demonstrates how t_0, t_1, t_2 and t_3 compose $s_{i,j}(x)$ and how the Taylor Expansion is calculated. First, we add for each k the k^{th} element in t_3 to the k^{th} element in t_2. Then, we add the k^{th} element in the sum of $t_2 + t_3$ to the k^{th} element in t_1. We split the elements into two halves and calculate iteratively and in parallel the same algorithm on both halves as long as each contains at least 4 elements. We will now get into the deep details of what each thread performs along the algorithm.

The algorithm is performed in three different kernels, to deal with the following three situations that come up along the algorithm, in any of which the amount of data

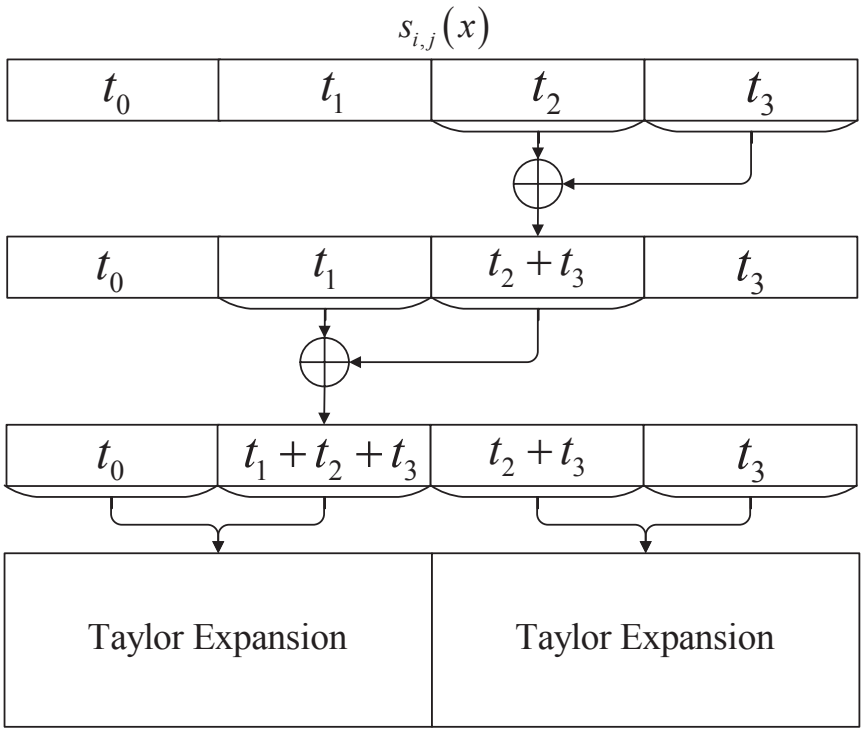


Figure 7.4: Outline of the Taylor Expansion Algorithm

each warp has to process is different but all implement the same basic idea presented above.

1. The first kernel will assume that $2^d/\mathcal{B} \geq 4$, so each $p_i(x)$ is represented using at least 4 chunks. The kernel is repeated until the polynomials don't meet the criterion stated above. Iteratively each polynomial takes $c = 2^d/\mathcal{B}$ chunks and $c/4$ warps will be processing it. Each polynomial $p(x)$ in each iteration will be written as in Equation 7.1 so t_0 is represented using the first $c/4$ chunks, the next $c/4$ chunks represent t_1 and so on. Warp number i among all warps that process that same polynomial $p(x)$ will take the i^{th} chunk representing t_3, t_2 and t_1, denoted by $D[i], C[i]$ and $B[i]$ respectively and thread j will perform for all $k \in R_j$

   (a) $C[i]_k \leftarrow C[i]_k \oplus D[i]_k$. $(t_2 \leftarrow t_2 + t_3)$

   (b) $B[i]_k \leftarrow B[i]_k \oplus C[i]_k$. $(t_1 \leftarrow t_1 + t_2)$

2. The second kernel assumes that $2^d/\mathcal{B} = 2$, so in this case $p_i(x)$ is represented by exactly two chunks A, that stores t_0 and t_1 and B, that stores t_2 and t_3 as described in Equation 7.1. The kernel is invoked once and the output of it are polynomials that are represented using exactly one chunk. Each thread j will perform for all rows number $k \in R_j$ the following,

   (a) Add (Xor) the most significant $\mathcal{B}/2$ bits of B_k into the least significant $\mathcal{B}/2$ bits of B_k.

(b) Add (Xor) the least significant $\mathcal{B}/2$ bits of B_k into the most significant $\mathcal{B}/2$ bits of A_k.

3. The third kernel assumes that $2^d/\mathcal{B} \leq 1$, so in this case $p_i(x)$ is represented in a single chunk A that stores 2^p polynomials, one of them is $p_i(x)$. In this case for each chunk will be processed by a single warp. The kernel is invoked repeatedly until the polynomials are of degree 2. Each thread j will perform for all rows number $k \in R_j$ the following,

   (a) Add (Xor) the bits that represent t_3 of all polynomials within the chunk with the bits that represent the t_2 of the same polynomial.

   (b) Add (Xor) the bits that represent t_2 of all polynomials within the chunk with the bits that represent the t_1 of the same polynomial.

   These operations can be done efficiently with simple bit-wise operations.

Once all executions of the third kernel are done, the phase is finished and the output is given as described above.

7.5 Shuffle Phase

On this phase we are given as input the output of previous phase,

$$\begin{pmatrix} \alpha_{t,0,0} & \beta_{t,0,0} & \alpha_{t,0,1} & \beta_{t,0,1} & \cdots & \alpha_{t,0,2^d-1} & \beta_{t,0,2^d-1} \\ \alpha_{t,1,0} & \beta_{t,1,0} & \alpha_{t,1,1} & \beta_{t,1,1} & \cdots & \alpha_{t,1,2^d-1} & \beta_{t,1,2^d-1} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \alpha_{t,2^t-1,0} & \beta_{t,2^t-1,0} & \alpha_{t,2^t-1,1} & \beta_{t,2^t-1,1} & \cdots & \alpha_{t,2^t-1,2^d-1} & \beta_{t,2^t-1,2^d-1} \end{pmatrix}$$

Where each line represents the Taylor-Expansion of one polynomial. The output of this phase is $g_{t+1,2i}(x)$ and $g_{t+1,2i+1}$ for each polynomial $g_{t,i}(x)$, define as,

$$g_{t+1,2i}(x) \quad = \sum_{j=0}^{2^d-1} \alpha_{t,i,j} x^j, \quad g_{t+1,2i+1}(x) \quad = \sum_{j=0}^{2^d-1} \beta_{t,i,j} x^j$$

So in each polynomial $p_i(x)$ in this phase we will move the $\alpha_{i,j}$ to the first half of elements and the $\beta_{i,j}$ to the last. The expected output is the following,

$$\begin{pmatrix} \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,2^d-1} & \beta_{0,0} & \beta_{0,1} & \cdots & \beta_{0,2^d-1} \\ \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,2^d-1} & \beta_{1,0} & \beta_{1,1} & \cdots & \beta_{1,2^d-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_{2^t-1,0} & \alpha_{2^t-1,1} & \cdots & \alpha 2^t - 1, 2^d - 1 & \beta_{2^t-1,0} & \beta_{2^t-1,1} & \cdots & \beta_{2^t-1,2^d-1} \end{pmatrix}$$

In other words, we perform a permutation on each line taking all even indexed elements to the first half with respect to their original order and taking all odd indexed elements

to the second half with respect to their original order as well. This permutation $\pi_d : [2^{d+1}] \to [2^{d+1}]$ is defined as follows,

$$\pi_d(i) = \begin{cases} \frac{i}{2} & i \text{ is even} \\ 2^d + \frac{i-1}{2} & \text{otherwise} \end{cases}$$

To permute these elements which reside inside chunks, we will perform the permutation for all rows (polynomials) in parallel. (e.g. to move the second element in the third chunk to the fourth element in the second chunk we have to move the second bit in all rows of the third chunk to the fourth bit of all rows of the second chunk). At first thought, the easiest and fastest algorithm to perform this permutation is allocating more memory space and copying, in parallel, element i to its index $\pi_d(i)$ in the allocated array. However, this turns out to be a bad solution in the given representation with chunks because of two main reasons,

1. Processing each bit of each element separately will ignore the ability of each thread to process $\mathcal{B}$ bits at the same time, impairing the performance.

2. Each row of each chunk at the output is affected by several different rows and chunks in the input. The same holds for each row of each chunk of the input, affecting several rows and chunks at the output. If different threads will process different rows and chunks concurrently, means of synchronization will have to be considered, this will impair the performance as well.

The algorithm we suggest is based on the following three sub-phases,

1. Let $\sigma_2, \sigma_4, \sigma_8 \dots \sigma_{\frac{\mathcal{B}}{2}}$ be the following permutations over $[\mathcal{B}]$,

$$\sigma_d(i) = \begin{cases} i - \frac{d}{2} & \lfloor \frac{2i}{d} \rfloor \equiv 2 \,(\mod 4) \\ i + \frac{d}{2} & \lfloor \frac{2i}{d} \rfloor \equiv 1 \,(\mod 4) \\ i & \text{otherwise} \end{cases}$$

   As shown in figure 7.5 the permutation σ_d is applied on the elements of the chunk denoted by A by partitioning the chunk into parts of $2d$ elements. Each part is then partitioned into 4 sections. The permutation exchanges the elements in the second and third sections of each part.

   Let us now explain what each permutation does. Assume $2^{d+1} \geq \mathcal{B}$ and let elements $\alpha_0, \beta_0, \dots, \alpha_{\mathcal{B}/2-1}, \beta_{\mathcal{B}/2-1}$ be some elements in the same chunk, which belong to the same polynomial. Our goal in the first part is to permute the elements within the chunk to achieve the order, $\alpha_0, \dots, \alpha_{\mathcal{B}/2-1}, \beta_0, \dots, \beta_{\mathcal{B}/2-1}$. The computation will be done in several steps, at the beginning of each step we assume the input is as follows,

$$A_0, B_0, A_1, B_1, \dots,$$

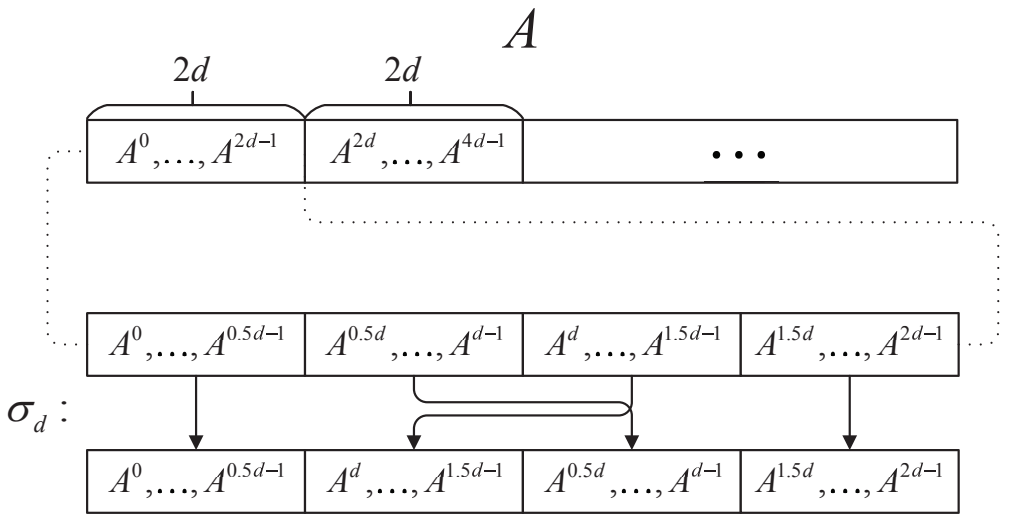


Figure 7.5: Applying Permutation σ_d on a Chunks' Elements

Where A_i and B_i are blocks of q alphas and betas respectively such that

$$A_i = \alpha_{i\cdot q}, \ldots, \alpha_{(i+1)\cdot q-1}$$
$$B_i = \beta_{i\cdot q}, \ldots, \beta_{(i+1)\cdot q-1}$$

After applying permutation σ_{2q} the blocks' order will be $A_0, A_1, B_0, B_1, A_2, A_3, \ldots$. Applying permutation σ_{4q} the blocks' order will be $A_0, \ldots, A_3, B_0, \ldots, B_3, A_4, \ldots$. At the beginning each block will be composed of exactly one element, after repetitively applying permutations $\sigma_2, \sigma_4, \ldots$ we will get the output as requested above.

The implementation of σ_i can be done using bitwise operations. The following code is a possible implementation of the permutation in C programming language on the bits of row i of chunk C.

$\overbrace{\texttt{d=}\overbrace{\underbrace{\texttt{11}\ldots\texttt{1}}_{q-\text{bits}}\underbrace{\texttt{00}\ldots\texttt{0}}_{3q-\text{bits}}\underbrace{\texttt{11}\ldots\texttt{1}}_{q-\text{bits}}\underbrace{\texttt{00}\ldots\texttt{0}}_{3q-\text{bits}}\ldots}^{\mathcal{B}\text{ bits}}}^{}$

```
C[i]=(C[i]&d) | (C[i]&(d>>3q)) | ((C[i]&(d>>2q))<<q) | ((C[i]&(d>>q))>>q)
```

If $2^{d+1} < \mathcal{B}$ only permutations $\sigma_2, \ldots, \sigma_{2^{d-1}}$, and the next sub-phases will not be executed at all.

Figure 7.6 gives an example for the flow of this phase for a single chunk when $\mathcal{B} = 16$. The phase is composed of three steps, in which we apply permutations σ_2, σ_4 and σ_8 consecutively to achieve the goal of this phase - having all α_i elements kept in order in the first half of the chunk and all β_i elements ordered in the second half.

2. On the second sub-phase we assume that $2^{d+1} \geq 2\mathcal{B}$ so each row in the input matrix of the shuffle phase is represented using at least two chunks. After the first sub-phase each chunk contents $\mathcal{B}/2$ alpha elements followed by $\mathcal{B}/2$ beta elements.

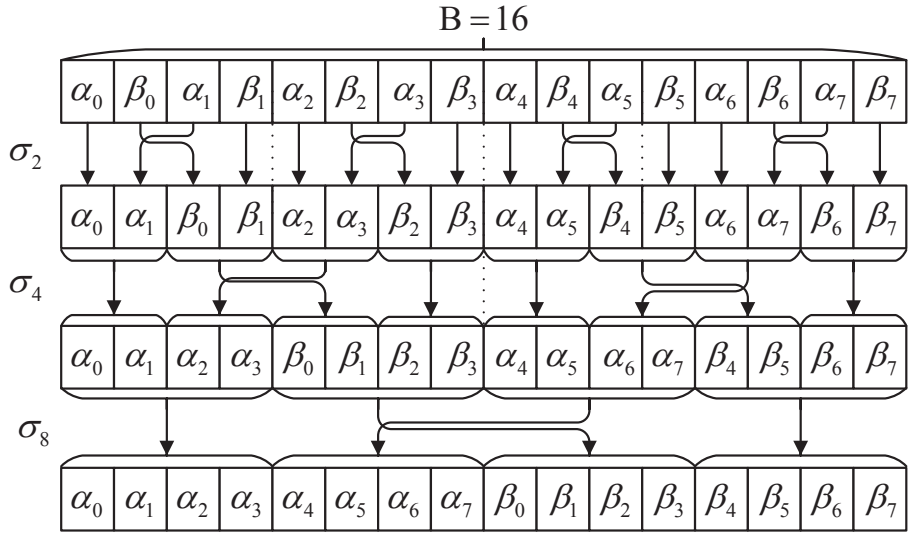


Figure 7.6: Applying Permutation π_8 on a Chunks' Elements

In this sub-phase we wish to change each to consecutive chunks such that the first chunk will contain the alpha elements of both chunks and the second chunk will contain the beta elements from both chunks. The implementation can be done as follows. For each pair of two chunks A and B we will launch a single warp. Thread j in that warp will perform for each row $i \in R_j$ a swap between the $\mathcal{B}/2$ MSBs of A_i LSBs of B_i. Performing this swap, in parallel on all rows of chunks A and B will give the requested output.

3. On the third sub-phase we assume that $2^{d+1} > 2$ so each row in the input matrix spanned over at least 4 chunks. Otherwise, this sub-phase is not performed. At the end of previous sub-phase it is assured that the input to this sub-phase is a series of chunks $A_0, B_0, A_1, B_1, \ldots$ such that A_i contains $\alpha_{i\mathcal{B}}, \ldots, \alpha_{(i+1)\mathcal{B}-1}$ and B_i contains elements $\beta_{i\mathcal{B}}, \ldots, \beta_{(i+1)\mathcal{B}-1}$. On this phase we will allocate a new array that will be the output array of the same length as the input array. The i^{th} chunk of the j^{th} row of the matrix after sub-phase 2 will be copies the $s_d(i)$ chunk of the j^{th} row at the output array.

7.6 Linear Evaluation Phase

In this phase we have as input many linear functions represented within chunks. Each chunk C will contain a series of pairs of elements $\alpha_{t,j,0}, \beta_{t,j,0}$ that represent the linear function $h_{t,j}(x) = \alpha_{t,j,0} + \beta_{t,j,0}x$. In this phase we will evaluate each $g_{t,j}$ over the affine subspace of two elements, $s, s+b$.

The requested output is replacing each α_i with $h_{t,j}(s)$ and each β_i with $h_{t,j}(s+b)$ to obtain $e_{t,j}$ evaluation of $h_{t,j}$ over the affine subspace $s, s+b$. In this phase each warp will perform two finite field multiplications of chunks. The first will multiply an input

chunk C by a chunk containing the elements $0, s, 0, s, 0, s, \ldots$ this will multiply each β_i by s, adding the result to α_i will give $h_i(x)$. The second finite field multiplication of chunks will multiply the input chunk by $0, b, 0, b, 0, b, \ldots$. This will multiply each β_i by b. Adding $h_i(s)$ that has already been calculated to $\beta_i \cdot b$ will give us $h_i(s+b)$, that will finish the calculation in this phase.

In figure 7.7 the flow of the linear evaluation phase over a single chunk is presented where $\mathcal{B} = 16$.

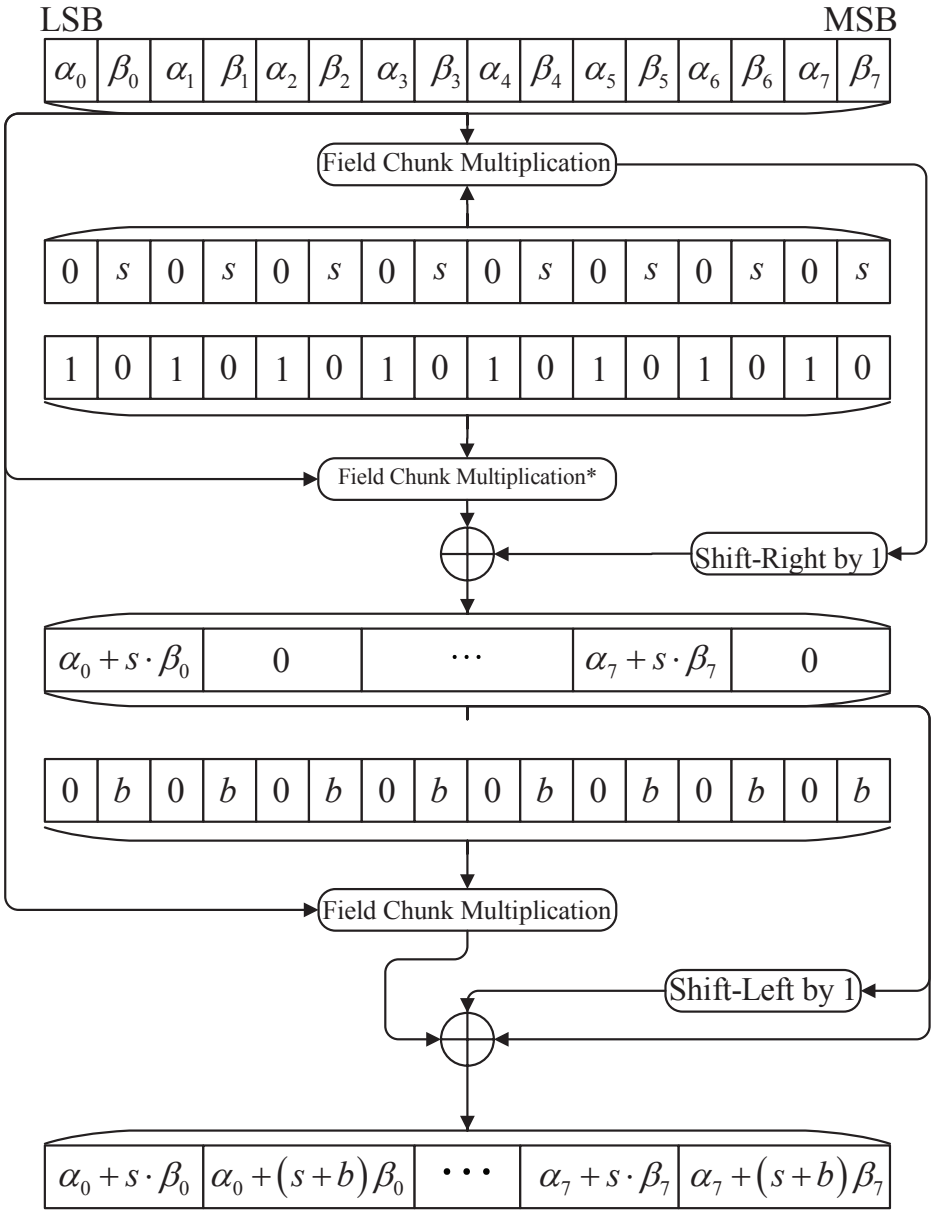


Figure 7.7: Linear Evaluation Phase Applied Over a Single Chunk

7.7 Merge Phase

The input of this phase are 2^t evaluations $e_{t+1,0}, \ldots, e_{t+1,2^t-1}$ of some polynomials over the same affine subspace $S = s_D + D$ in this phase we will take each pair of evaluations $e_{t+1,2j}$ and $e_{t+1,2j+1}$ and treat them as the evaluations of some polynomials $g_{t+1,2j}(x)$ and $g_{t+1,2j+1}(x)$ respectively as described in Algorithm 3.5 where each $e_{t+1,2j}$ and $e_{t+1,2j+1}$ are the U and V (as in algorithm 3.5) of some FFT invocation, respectively. The evaluations are kept in memory in an array of chunks. The i^{th} element in each evaluation $e_{t+1,j}$ is $g_{t+1,j}(s_D + D[i])$ where $e_{t+1,j}$ is the corresponding evaluation of $g_{t+1,j}(x)$ over the affine subspace $s_D + D$.

In this phase, as described in Algorithm 3.5, we will take each pair of $e_{t+1,2j}$ and $e_{t+1,2j+1}$ with evaluations $u_0, \ldots, u_{2^d-1}$ and $v_0, \ldots, v_{2^d-1}$. These will be used to compute for the polynomial $g_{t,j}$ the evaluation of it, $w_0, \ldots, w_{2^{d+1}-1}$ such that for each $0 \leq i < 2^d$,

$$w_i \leftarrow u_i + (s_D + D[i]) \cdot v_i$$
$$w_{2^d+i} \leftarrow w_i + v_i$$

Assuming we know beforehand the subspace of the FFT we will keep in memory for each affine subspace $s_D + D$ for each merge iteration (as described in section 7.1) of the algorithm an array of all elements in that subspace, in order. One of the following cases will happen in each level of recursion,

1. The number of elements in the subspace is bigger than or equals to B (number of elements that can be stored in a single chunk). In that case, the elements will be kept in an array of chunks.

2. The number of elements in the subspace is smaller than B. In that case, a single chunk will be allocated for the storage of elements in the subspace. Let 2^d be the number of elements in that subspace, then the chunk will be filled such that each group of consequent 2^{d+1} elements will contain first 2^d zero-elements and then 2^d subspace elements. The reason for this storage will be explained next.

The implementation of this phase will be taking all elements in $e_{t+1,2j+1}$ multiplying them by the elements in the affine subspace $s_D + D$ which we already have in memory in multi-point fashion such that the v_i will be multiplied by $s_D + D[i]$. The output will be added to the corresponding u_i in $e_{t+1,2j}$. The result after the addition will be added to v_i in $e_{t+1,2j+1}$ to obtain $e_{t,j}$.

To do this a single warp will be launched for each chunk in which elements of $e_{t+1,2j+1}$ exist.

In the case in which a subspace has $e < \mathcal{B}$ each chunk in our input has both $e_{t+1,2j}$ elements and $e_{t+1,2j+1}$. Therefore, elements we will keep all elements of the subspace in a single chunk in the following way. The first $2e$ elements will be e zero field-elements and e elements will be the elements of the subspace. The next $2e$ elements will be the

same and so on. Now, after multiplication of each chunk of evaluations by that chunk the result will be first right-shifted e times. As for each v_i that was multiplied by some subspace element, the results should be added to u_i that resides in this case in the same chunk e elements before v_i. Notice that after the addition each v_i not be changed since all u_i elements were multiplied by zero.

Chapter 8

Performance

In this chapter we will present the performance of various implementations for the FFT, inverse FFT and finite field multiplication algorithm.

The source code of the finite field multiplication is available online [1]. We incorporate the algorithms in Section 6.3 into the finite field multiplication implementation according to Algorithm 6.1.

Methodology We use GeForce® GTX TITAN-X GPU, and a Supermicro Server with 2x6 Intel® Xeon® E5-2620 v2 @ 2.10GHz CPUs with 64GB of RAM. For each measurement we perform five executions, remove the highest and lowest results, and compute the average of the remaining three. We observe negligible standard deviation, less than $< 4\%$. Hyperthreading and CPU power management are disabled to achieve reproducible CPU performance. Each experiment uses random data for its input. As a CPU baseline we use NTL version 8.1.2 [V. 03], which is a highly-optimized single-core CPU library for finite field arithmetics that uses `CLMUL` CPU intrinsics for polynomial multiplication.

Speedup over CPU for GF(2^{32}) and GF(2^{64}) Our implementation for GF(2^{32}) and GF(2^{64}) employs optimized register cache implementations of n=32 and n=64 polynomial multiplication respectively. We emphasize that we apply the same optimizations the NTL does when 2-gapped polynomials are used, and that the NTL implementation is based on the `CLMUL` instruction.

Figure 8.1 shows the results. The GPU implementations for GF(2^{64}) and GF(2^{32}) are up to **99** $\times$ and **138** $\times$ faster than NTL's CPU multiplication for inputs exceeding 2^{26} elements.

We observe that the speedups are not constant. The reason lies in the variability in the NTL performance, which drops by about 15% for larger inputs. The GPU implementation performance keeps rising until it plateaus out for inputs exceeding 2^{25} elements.

[1] `https://github.com/HamilM/GpuBinFieldMult`

The peak throughputs of GPU implementations are **3.15** and **2.09** billion finite field multiplications per second for $\mathrm{GF}(2^{32})$ and $\mathrm{GF}(2^{64})$ respectively. Note that these throughputs are slightly lower than the throughput of the respective polynomial multiplication, because finite field multiplication involves multiple polynomial multiplications.

Register cache vs. shared memory We compare two implementations for multiplication in $\mathrm{GF}(2^{64})$: with shared memory and with register cache. This experiment seeks to evaluate the impact of our register cache optimization on the end-to-end application performance. We observe that the register cache version is 50% faster than the shared memory version. As expected, the performance boost is smaller than in the pure polynomial multiplication case reported in Table 6.1.

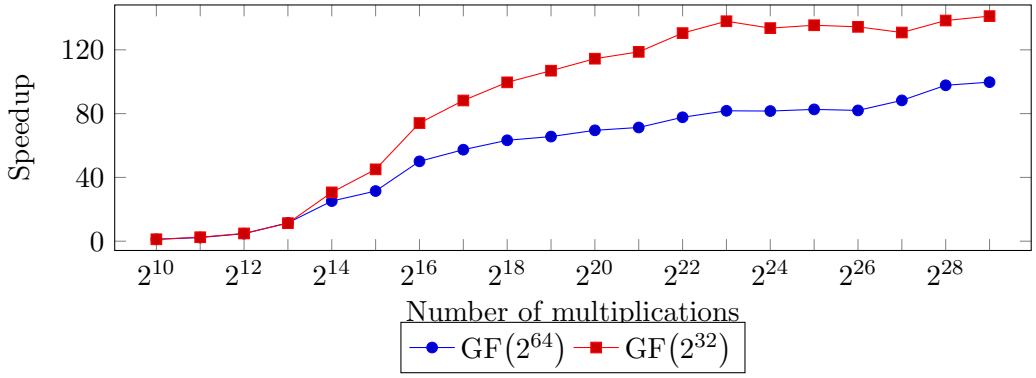


Figure 8.1: Speedup of register cache multiplication in $\mathrm{GF}(2^{64})$ and $\mathrm{GF}(2^{32})$ over NTL

Performance for larger fields We evaluate the performance of the finite field multiplication in fields of higher degrees. Here we incorporate our hybrid implementation for polynomial multiplication described in Section 6.4, using the n=32 polynomial multiplication as its building block. We measure the performance for fields from $\mathrm{GF}(2^{32})$ to $\mathrm{GF}(2^{2048})$. We use 2^{23} elements per input.

Figure 8.2 shows the speedup of our implementation over NTL. We achieve significant speedups for smaller fields, but when fields grow larger our speedup diminishes (to 2.17× in $\mathrm{GF}(2^{2048})$). The reasons are found in the NTL implementation. For fields smaller than $\mathrm{GF}(2^{64})$, NTL uses the `CLMUL` intrisics, which allow only multiplication of n=64 degree polynomials; the implementation is therefore inefficient for these fields. Our GPU implementation does not suffer from this limitation. However, for larger fields NTL uses a different hybrid algorithm (Karatsuba), which is asymptotically faster than the quadratic algorithm we use. The problem of implementing the Karatsuba algorithm on GPUs is in the difficulty to balance the load across threads. We leave the implementation of a GPU Karatsuba for future work.

Performance for other fields Figure 8.3 shows the performance of our GPU implementation for $\mathrm{GF}(2^N)$ where $N \neq 2^n$. As expected, we observe the step function,

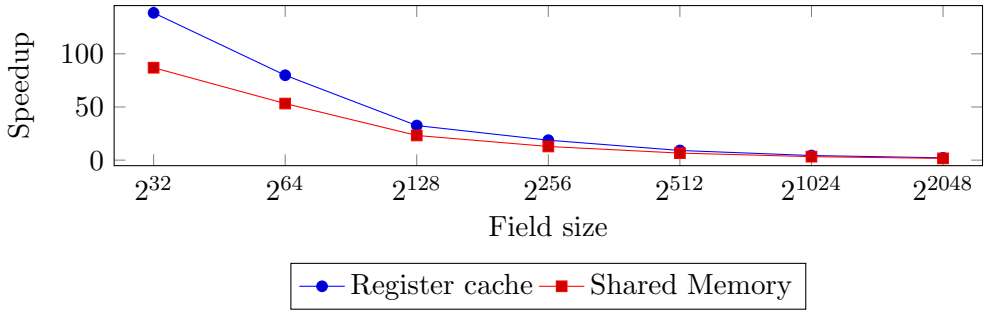


Figure 8.2: Speedup over NTL for varying field sizes

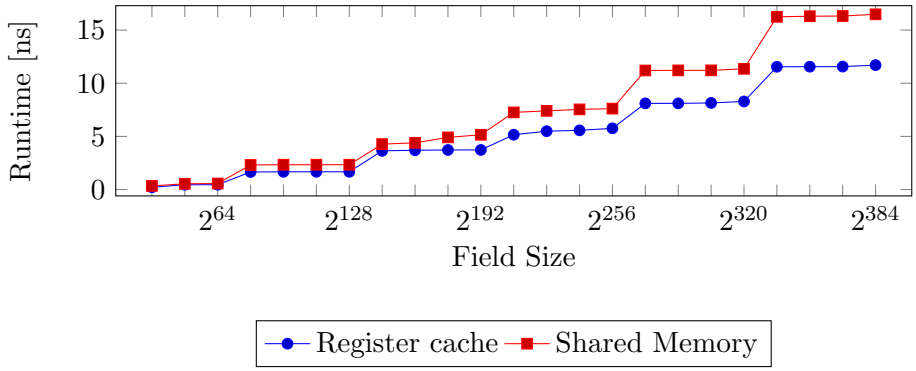
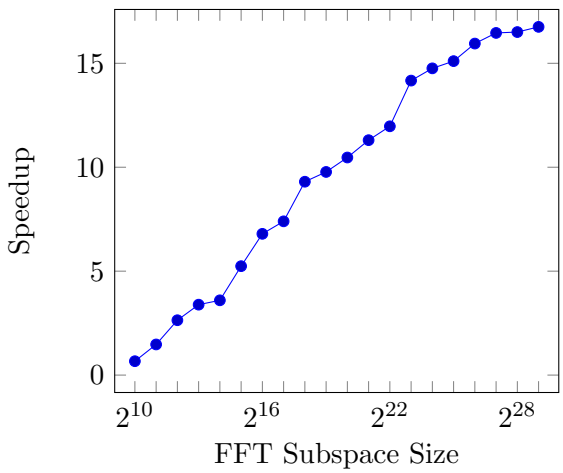


Figure 8.3: Finite field multiplication performance for $\mathrm{GF}(2^N)$ where N is not a power of 2.

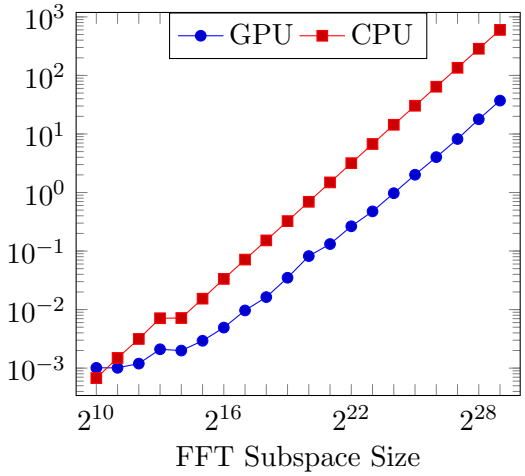
where in each step the inputs are processed by the same number of warps. The number of warps in our implementation employed in $\mathrm{GF}(2^N)$ is $\lceil \frac{N}{64} \rceil$.

Considering alternative CPU implementations In all our experiments we use a single-threaded NTL implementation for CPU as the performance baseline. NTL natively supports multiplication of a single pair of elements and uses `CLMUL` instruction. One could argue, however, that extending NTL to support multiplication of many pairs in a batch, as we do in GPUs, might open additional optimization opportunities, e.g., bit-slicing techniques like those proposed in Section 6.2. Thus, it would become possible to use the AVX vector instruction set instead of `CLMUL`, potentially improving NTL performance.

We now show why `CLMUL` implementation is superior. In the AVX instruction set [Fog16] a single 512-bits wide AND and XOR takes 1 cycle each. Therefore, using our bit-slicing algorithm, we can multiply 512 pairs of polynomials of degree 64 in $2 \times (64^2) = 8192$ cycles. Note that this estimate is rather optimistic, as we ignore the time to reorganize the input bits to allow vectorized execution. On the other hand, each CLMUL instruction multiplies a single pair of polynomials of degree 64 in 3.5 cycles (latency 7 cycle, throughput=2) [Fog16]. Therefore, 512 polynomials can be multiplied in $3.5 \times 512 = 1792$ cycles alone, much faster then the bit-sliced AVX-based implementation.

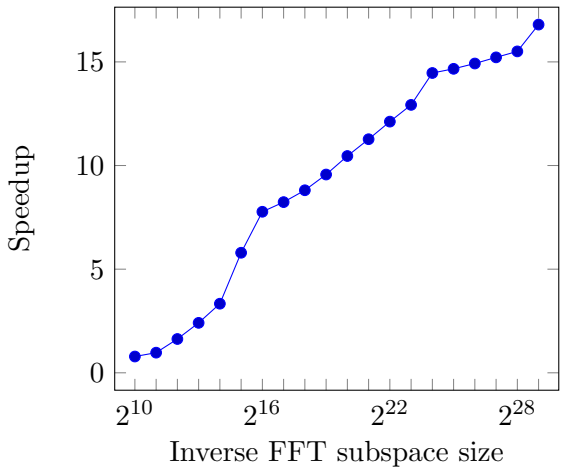


(a) Speedup of the GPU FFT implementation over the CPU FFT implementation as a function of the subspace size

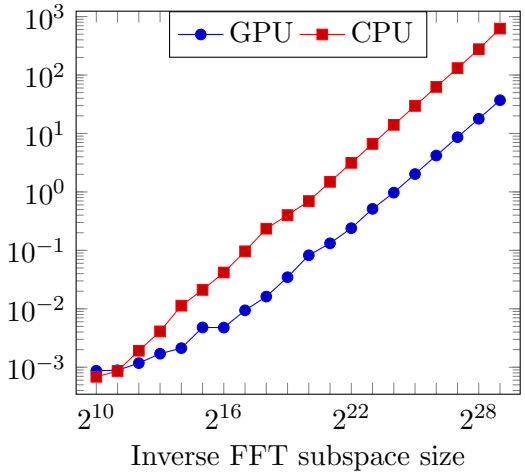


(b) Time to compute the FFT as a function of the subspace size

Figure 8.4: Comparison of GPU and a single threaded CPU implementation for FFT



(a) Speedup of the GPU inverse FFT implementation over the CPU inverse FFT implementation as a function of the subspace size



(b) Time to compute the inverse FFT as a function of the subspace size

Figure 8.5: Comparison of GPU and a single threaded CPU implementation for inverse FFT

8.1 FFT and Inverse FFT

In this section we will present the performance of our GPU implementation for the FFT and inverse FFT over multiple affine subspace dimensions.

Figures 8.4 and 8.5 present the runtime and the speedup of the GPU implementation over the serial CPU implementation for FFT and inverse FFT respectively.

Due to the symmetry of the algorithms, both figure portray the same picture. In

both algorithms the GPU implementation achieved a speedup of 16. In the TITAN-X architecture the GPU implementation achieves a maximal subspace size of 2^{29} and is limited by the global memory size of 12 GBs. For the maximal input size of 2^{29} the GPU achieves a running time of 37 seconds for both FFT and inverse FFT.

Chapter 9

Conclusion and Open Questions

9.1 Conclusions

GPU Scales GPGPU programming with cuda gives more power to the programmer in memory hierarchy management. This manual management of registers and shared memory can prevent some phenomena on CPU that may cause lack of scalability like false sharing or excessive cache miss rates. However, with great power comes great responsibility as one has to take many considerations and restrictions into account when manually managing his memory hierarchies and this management can be a quite a burden to the programmer.

Utilizing CPU and GPU Instruction-Set To achieve high performance and throughputs, applications should use the state of the art SIMD instructions in the instruction set and not be restricted to old, classical instructions. The key to the high performance of the CPU implementation of finite field multiplication is the use of the PCLMUL instruction. Particularly the SSE instruction sets make CPU SIMD programming simpler and very efficient.

Warp Locallity in GPU The register cache method presented in chapter 5.2 can be used in a wide variety of applications as discussed in chapter 1 to achieve a modular and scalable primitive for intra-warp communication. This method can accelerate computations which are warp-centric, i.e - the computation can be broken into somewhat larger parts in which each warp in independent. Several uses for this method are given in chapter 5.2. In chapter 6 we presented the main use of this method in this work - accelerating finite field multiplication in binary fields. With this method being applied, additional throughput of 50% is measured.

9.2 Some open questions

Is Gao and Matteer's Algorithm Inherently unscalable? In section 3.2 we presented an additive FFT algorithm for affine subspaces over finite fields, originally published by Gao and Matteer [S. 10] with a little addition making it compatible for affine subspaces as well. In section 4.2 we presented an implementation of this algorithm in CPU architecture and the performance of this implementation as discussed in section 8 point that this implementation does not scale because of high cache miss-rate along with high NUMA traffic. Is implementing this algorithm on CPU is possible in a scalable manner? Can one avoid the excessive miss-rates? If not, what properties of the algorithm or CPU architecture cause it?

Can the GPU Hardware Support a Warp-Level Cache? In chapter 5.2 the model of register-cache method is presented. This model behaves as a virtual warp-level cache, in term that each thread reads several values from the shared/global memory, stores them in registers, and shares them using shuffle with any threads wishes to access to these values, by that reducing shared/global-memory traffic. This cache is not only manually configured but is also not an inherent part of a large set of computations, as shown in chapters 1 and 5.2. We raise a question whether it is possible that the GPU architecture will support a such warp-level cache as parts of its hardware and whether this level of cache is manually managed by the user or automatically by the GPU.

Can the register cache application can be automatized? In chapter 5.2 we present the model of the register-cache method alongside a sime use-case of it. The programmer in our case should be always aware of the usage of register cache when reading and writing to memory. Implementing an automatic tool that in compilation-time can substitute memory-accesses with the corresponding shuffles to achieve register cache will relieve the programmer of this burden and perhaps will be able to distribute the data between threads in an optimal fashion that minimizes the number of shuffles performed for a single memory access.

Bibliography

[A. 10] A. E. Cohen and K. K. Parhi. GPU Accelerated Elliptic Curve Cryptography in $GF(2^m)$. In *IEEE 53rd International Midwest Symposium on Circuits and Systems*, pages 57–60, Aug 2010.

[A. 11] A. Davidson, and J. D. Owens. Register packing for cyclic reduction: A case study. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, pages 4:1–4:6. ACM, 2011.

[A. 13] A. Magni, C. Dubach, and M. F. P. O'Boyle. A Large-scale Cross-architecture Evaluation of Thread-coarsening. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 11:1–11:11. ACM, 2013.

[AS98] Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs: a new characterization of NP. *Journal of the ACM*, 45(1):70–122, 1998. Preliminary version in FOCS '92.

[B. 07] B. Chapman, G. Jost, and R. V. D. Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[B. 14] B. Catanzaro, A. Keller, and M. Garland. A decomposition for in-place matrix transposition. *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 193–206, 2014.

[BSGH+06] Eli Ben-Sasson, Oded Goldreich, Prahladh Harsha, Madhu Sudan, and Salil Vadhan. Robust pcps of proximity, shorter pcps, and applications to coding. *SIAM J. Comput.*, 36(4):889–974, December 2006.

[C. 12] C. Su, and H. Fan. Impact of Intel's new instruction sets on software implementation of GF(2)[x] multiplication. *Inf. Process. Lett.*, 112(12):497–502, June 2012.

[D. 82] D. F. Elliott, and K. R. Rao. *Fast Transforms: Algorithms, Analyses, Applications*. Computer Science and Applied Mathematics Series. Academic Press, 1982.

[D. 91] D. G. Cantor, and E. Kaltofen. On fast multiplication of polynomials over arbitrary algebras. *Acta Informatica*, 28(7):693–701, 1991.

[Din06] Irit Dinur. The PCP theorem by gap amplification. In *Proc. 38th ACM Symp. on Theory of Computing*, pages 241–250, 2006.

[E. 96] E. D. Win, A. Bosselaers, S. Vandenberghe, P. D. Gersem, and J. Vandewalle. A fast software implementation for arithmetic operations in GF(2^n). In *Proceedings of the International Conference on the Theory and Applications of Cryptology and Information Security: Advances in Cryptology*, pages 65–76. Springer-Verlag, 1996.

[E. 08] E. Ben-Sasson, and M. Sudan. Short PCPs with polylog query complexity. *SIAM Journal on Computing*, 38(2):551–607, 2008. Preliminary version appeared in STOC '05.

[Eis50] G. Eisenstein. Lehrsätze. *Journal für die reine und angewandte Mathematik*, 39:180, 1850.

[F. 95] F. Ergün. Testing Multivariate Linear Functions: Overcoming the Generator Bottleneck. In *Proceedings of the Twenty-seventh Annual ACM Symposium on Theory of Computing*, STOC '95, pages 407–416, New York, NY, USA, 1995. ACM.

[Fog16] A. Fog. Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. Available at `http://www.agner.org/optimize/instruction_tables.pdf`, 1996-2016. [Online; accessed 28-Mar-2016].

[G. 98] G. Seroussi. Table of low-weight binary irreducible polynomials. In *HP Labs Technical Reports*, pages 98–135, 1998.

[G. 14] G. Shay, and M. E. Kounavis. Intel(R) carry-less multiplication instruction and its usage for computing the GCM mode - rev 2.02. Intel Corporation, April 2014.

[G. 15] G. L. Steele Jr., and J. B. Tristan. Using butterfly-patterned partial sums to optimize GPU memory accesses for drawing from discrete distributions. *CoRR*, abs/1505.03851, 2015.

[Gao93] S. Gao. Normal bases over finite fields, 1993.

[Gau66] C. F. Gauss. Nachlass: Theoria Interpolationis methodo nova tractata. (German/Latin) [deduction: Interpolation theory by a new method]. In *Carl Friedrich Gauss, Werke*, volume 3, pages 265–303. Konigleichen Gesselschaft der Wissenschaften, Göttingen, Germany, 1866. Posthumous publication of undated, and previously unpublished, work done about October/November 1805, according to historical evidence presented in [HJB85]. Gauss' work predates Fourier's work of 1807 on the representation of functions as infinite series of trigonometric functions, but due to opposition by Lagrange, Fourier did not publish it until 1822.

[Hen88] K. Hensel. Ueber die Darstellung der Zahlen eines Gattungsbereiches fur einen beliebigen Primdivisor. *Journal Fur Die Reine Und Angewandte Mathematik*, 1888:230–237, 1888.

[HJB85] Michael T. Heideman, Don H. Johnson, and C. Sidney Burrus. Gauss and the history of the Fast Fourier Transform. 34(3):265–277, September 1985.

[I. 60] I. S. Reed, and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[J. 65] J. W. Cooley, and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Math. Comput.*, 19:297–301, 1965.

[J. 86] J. L. Massey, and J. K. Omura. Computational method and apparatus for finite field arithmetic. US patent number 4587627. May 1986.

[J. 98a] J. Daemen, and V. Rijmen. AES proposal: Rijndael. Available at `http://jda.noekeon.org/JDA_VRI_Rijndael_V2_1999.pdf`, 1998. [Online; accessed 28-Mar-2016].

[J. 98b] J. L. Massey. The Discrete Fourier Transform in Coding and Cryptography. *IEEE Inform*, 1998.

[J. 03] J. Von-Zur Gathen, and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 2 edition, 2003.

[J. 12] J. Luo, K. D. Bowers, A. Oprea, and L. Xu. Efficient software implementations of large finite fields GF(2^n) for secure storage applications. *Trans. Storage*, 8(1):2:1–2:27, February 2012.

[J. 13] J. S. Plank, K. M. Greenan, and E. L. Miller. Screaming fast Galois field arithmetic using Intel SIMD instructions. In *11th USENIX*

Conference on File and Storage Technologies, pages 298–306, February 2013.

[K. 12] K. Leboeuf, R. Muscedere, and M. Ahmadi. High performance prime field multiplication for GPU. In *IEEE International Symposium on Circuits and Systems*, pages 93–96, May 2012.

[Kil92] J. Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing, May 4-6, 1992, Victoria, British Columbia, Canada*, pages 723–732, 1992.

[KO63] A. Karatsuba and Y. Ofman. Multiplication of Many-Digital Numbers by Automatic Computers. Doklady Akad. Nauk SSSR145, 293-294, 1962. Translation in Physics-Doklady 7, 595-596, 1963.

[L. 88] L. Babai, and S. Moran. Arthur-merlin games: A randomized proof system, and a hierarchy of complexity class. *J. Comput. Syst. Sci.*, 36(2):254–276, April 1988.

[L. 90] L. Babai, L. Fortnow and C. Lund. Non-deterministic exponential time has two-prover interactive protocols. *computational complexity*, 1(1):3–40, 1990.

[L. 91] L. Babai, L. Fortnow and L. A. Levin, and M. Szegedy. Checking computations in polylogarithmic time. In *Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing*, STOC '91, pages 21–32, New York, NY, USA, 1991. ACM.

[LRRY78] B. Gold L. R. Rabiner and C. K. Yuen. Theory and application of digital signal processing. *IEEE Transactions on Systems, Man, and Cybernetics*, 8(2):146–146, Feb 1978.

[M.] M. J. Mohlenkamp. A fast transform for spherical harmonics. *Journal of Fourier Analysis and Applications*, 5(2):159–184.

[M. 05] M. Arabi. *Comparison of Traditional, Karatsuba and Fourier Big Integer Multiplication. B.Sc. Thesis.* University of Bath, May 2005.

[Mat08] T. Mateer. *Fast Fourier Transform Algorithms with Applications.* PhD thesis, Clemson, SC, USA, 2008. AAI3316358.

[Mic94] S. Micali. CS proofs (extended abstracts). In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*, pages 436–453, 1994.

[Mie08] Thilo Mie. Polylogarithmic two-round argument systems. *J. Mathematical Cryptology*, 2(4):343–363, 2008.

[nVi15] nVidia. Kepler Tuning Guide. `http://docs.nvidia.com/cuda/kepler-tuning-guide/index.html`, 2015. [Online; accessed 26-Jan-2016].

[P. 15] P. Enfedaque, F. Auli-Llinas, and J. C. Moure. Implementation of the DWT in a GPU through a register-based strategy. *IEEE Transactions on Parallel and Distributed Systems*, 26(12):3394–3406, Dec 2015.

[R. 89] R. C. Mullin, I. M. Onyszchuk, S. A. Vanstone, and R. M. Wilson. Optimal normal bases in $gf(p^n)$. *Discrete Appl. Math.*, 22(2):149–161, February 1989.

[R. 97a] R. Lidl and H. Niederreiter. *Finite Fields*. (2nd ed.), Cambridge University Press, 1997.

[R. 97b] R. Lidl, and H. Niederreiter. *Finite Fields*. Number v. 20, pt. 1 in Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1997.

[R. 02] R. D. Kent, and C. Read. *The Acoustic Analysis of Speech*. Singular/Thomson Learning, 2002.

[S. 89] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. 18(1):186–208, February 1989.

[S. 98] S. Arora, C. Lund, R. Motwani and M. Sudan, and M. Szegedy. Proof verification and the hardness of approximation problems. *J. ACM*, 45(3):501–555, May 1998.

[S. 10] S. Gao, and T. Mateer. Additive fast fourier transforms over finite fields. *IEEE Trans. Inf. Theor.*, 56(12):6265–6272, December 2010.

[S. 11] S. Kalcher, and V. Lindenstruth. Accelerating Galois field arithmetic for Reed-Solomon erasure codes in storage applications. In *IEEE International Conference on Cluster Computing*, pages 290–298, Sept 2011.

[S. 16] S. Ashkiani, A. Davidson, U. Meyer, and J. D. Owens. GPU Multisplit. *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 12:1–12:13, 2016.

[SS71] A. Schonhage and V. Strassen. Schnelle multiplikation grosser zahlen. *Computing*, 7(3-4):281–292, 1971.

[V.] V. Volkov, and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *International Conference for High Performance Computing, Networking, Storage and Analysis, Nov 2008.*, pages 1–11.

[V. 03] V. Shoup. NTL: A library for doing number theory. Avilable at `http://www.shoup.net/ntl`, 2003. [Online; accessed 28-Mar-2016].

[Wel67] P. D. Welch. The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short, Modified Periodograms. *IEEE Transactions on Audio and Electroacoustics*, 15(2):70–73, 1967.

[Wik] Wikipedia. Bit slicing — Wikipedia,. [Online; accessed 27-Mar-2016].

פים באותה הקבוצה ללא גישה נוספת לזיכרון משותף או גלובאלי. בעבודה זו אנו מציגים מתודולוגיה יוניפורמית ומלאה הנקראת "מטמון אוגרים" המיועדת להאצת חישובים ב- GPU ועוסקת במטמון מבוסס אוגרים וכן מוצגים מספר רב של אלגוריתמים לארכיטקטורת ה- GPU אותם ניתן להאיץ באמצעות שימוש במתודולוגיה זו.

את כל הדרכים אליהן מתפצל ביצוע הקוד, דבר שיפגע באופן אנוש בזמני הריצה. בנוסף, כל שלושים ושתיים קבוצות, שהן אלף עשרים וארבעה חוטים, מתקבצות ליחידה חישוב אחת שנקראת "בלוק". ל- GPU יש מספר מוגבל חומרתית של מעבדים כאשר כל מעבד בכל זמן נתון מריץ בלוק של חוטים. בכל מעבד יש יחידת עיבוד וכן "זיכרון משותף" שבו, תוך שימוש בפקודות סנכרון מתאימות, יכולים חוטים מאותו הבלוק לשתף מידע. זיכרון זאת הוא קטן יחסית ושימוש בו עשוי להקשות על מימוש כפלים בשדות הרחבה בינאריים מדרגה גבוהה לכן נשתדל להמעיט בשימוש בו. בנוסף יש היררכיית זיכרון גלובאלית, בה נמצא הקלט בתחילת הריצה של התכנית והפלט נכתב אליו בסוף הריצה של התכנית. הגישה להיררכיית הזיכרון הגלובאלית היא איטית ובעבודה זו מוצג מימוש בו נמנענו לחלוטין פרט לקריאת קלט ולכתיבת פלט מגישה לזיכרון זה. התכנות לארכיטקטורה זו מתבצעת על ידי סדרה של תכניות כאשר כל תכנית מקבלת קלט, מבצעת עליו חישובים ואת הפלט שלה מעבירה לתכנית הבאה. לפי-כך, לא ניתן לבצע תקשורת בין חוטים שאינם מאותו הבלוק, אלא על ידי כתיבת פלט בתכנית אחת על ידי חוט כלשהו וקריאת הקלט בתוכנית העוקבת על ידי חוט אחר. אילוץ נוסף עליו יש לשמור הוא שבמהלך גישה לזיכרון הגלובאלי של חוטים מבלוק כלשהו, על הכתובות הנ-קראות מהזיכרון להיות "מרובבות", כלומר להיות קרובות זו לזו, זאת כיוון שהקריאה מהזכרון הגלובאלי מתבצעת באופן טרנאזאקציוני של מספר גדולים של כתובת זיכרון סמוכות. לכן, אם הכתובות הנקראות רחוקות זו מזו יידרשו מספר רב של טרנאזאקציות לביצוע הקריאה.

מימוש התמרות הפוריה ב- GPU הוא נטול פיצולים וכל הגישות בו לזיכרון הגלובאלי הן מרובבות. התפוקה של ה- GPU נמצאה כגבוהה פי שישה-עשר בבדיקות שנעשו. על מנת לממש את ההתמרות ב- GPU נעשה מימוש לכפל מהיר בשדה סופי. בהמימוש הוא מקבילי גם כן וכל קבוצת חוטים אחראית על כפל של רצף של איברים בשדה סופי ברצף איברים אחר. מניתוח מדויק עלה כי מספר הפעולות שהמימוש מבצע בזמן הכפל של שני איברים בשדה סופי תלוי בין היתר במספר המקדמים השונים מאפס בפולינום האי-פריק. לפי-כך, נוספה דרישה לפולינומים האי-פריקים שהשתמשנו בהם במימוש שיהיה יהיו פנטנומיאלים, כלומר בעלי 5 מקדמים שונים מאפס לכל היותר. מחקרים קודמים שנעשו בנושא מראים כי לכל צורך פרקטי ידוע על פולינומים בינאריים אי-פריקים מרווחים ופנטומיאליים וכן קיימות השערות שטרם הופרכו באשר לקיומם בכל שדה בינארי גדול מספיק. באמצעות פולינומים אלו המימוש של הכפל בשדה הסופי נתן תפקודה גבוה בסדר גודל ביחס למימוש הכפל ב- CPU.

מתוך מחקר מעמיק יותר שנעשה באשר לאופן הגישות של חוטים שונים לזיכרון שנמצאים באותה קבוצה עולה כי הם ניגשים בהפרשי זמן קצרים למקומות זהים בזיכרון הגלובאלי, שהגישות אליו הן איטיות. לכן, פותח ייצוג אחר בזיכרון לאיברים בשדה הסופי. הייצוג הינו מבוזר ומאפשר לאיברים המוכפלים על ידי אותה קבוצת חוטים להישמר באוגרים שלהם, מה שאמור להקנות זמני גישה קטנים יותר לזיכרון ולחסוך מספר רב של גישות לזיכרון הגלובאלי. על מנת לשתף תוכן שנמצא באוגר של חוט מסוים בקבוצה עם חוט אחר באותה הקבוצה השתמשנו בפקודת הערבוב (shuffle) של המעבד. בדרך זו הצלחנו להוריד כשליש מזמן הריצה הדרוש לכפל של איברים בשדה הסופי.

העיקרון המנחה באמצעותו הצלחנו להאיץ את מימוש הכפל בשדות סופיים ב- GPU היה הור-דת סך התעבורה לרמות זיכרון איטיות (זיכרון משותף וגלובאלי) ויצירת רמת מטמון חדשה באופן וירטואלי על ידי שמירת ערכים באוגרים של חוטים ושיתוף שלהם בין חוטים נוס-

מנת שהמימוש יהיה יעיל ככל הניתן, המימוש הינו מקבילי גם כן. בנוסף, במסגרת עבודה זו נבחנו שתי ארכיטקטורות שונות בהן מומש האלגוריתם, CPU ו- GPU. על מנת לעמוד במטרת המחקר, ובהינתן האתגר הגדול ביותר במימוש אלגוריתמי התמרות פוריה אדיטיביים מעל שדות סופיים, יש לצמצם תחילה את סך זמן הריצה הדרוש לכפל בשדה סופי.

צמצום סך זמן הריצה הדרוש לכפל בשדה סופי באלגוריתמים לחישוב התמרות פוריה אדיטיביות נעשה במחקר זה בשתי דרכים. הראשונה, היא צמצום מספר הכפלים שנעשים על ידי האלגוריתם והשני היא צמצום סך הזמן הדרוש לביצוע פעולת כפל אחת.

על מנת לצמצם את מספר הכפלים שהאלגוריתם צריך לבצע הוחלט להשתמש באלגוריתם קיים שפורסם בשנת 2010 אך מאפשר לבצע את החישוב אך ורק מעל תתי-מרחבים ולא מעל כל תת-מרחב אפיני בשדה הסופי. בעבודה זו מוצגים מספר שינויים שנעשו לאלגוריתם על מנת לאפשר לו לבצע את החישוב מעל תתי-מרחבים אפינים כלשהם.

האיברים בשדה הסופי ממציין שתיים מיוצגים בבסיס הסטנדרטי, כלומר, כל איבר הוא פולינום עם מקדמים אפס או אחת ודרגתו קטנה מדרגת ההרחבה של השדה הסופי בו עובדים. בשדה זה כל הפעולות מתבצעות מודולו פולינום אי-פריק כלשהו. על מנת לבצע כפל מהיר אם כן, היה צורך בביצוע יעיל של פעולות הכפל של הפולינומים וכן של פעולות המודולו. על מנת לבצע ביעילות את פעולת המודולו בפולינום אי פריק, החלטנו להשתמש בפולינומים אי-פריקים "מרווחים". פולינום ייקרא "מרווח" אם חזקת משתני הפולינום שמקדמיהם שונים מאפס קטנה ממחצית מדרגת הפולינום, פרט כמובן, למקדם השונה מאפס של חזקת משתנה הפולינום הקובע את דרגתו של הפולינום. במקרה בו משתמשים בפולינומים שכאלו הראינו בעבודה זו כיצד ניתן להשתמש בשתי פעולות כפל פולינומים, שהפכו ליעילות באמצעות פקודת המעבד שהוזכרה לעיל, על מנת לבצע פעולת מודלו בפולינום "מרווח" ובכך למעשה לבצע כפל בשדה סופי באמצעות שלוש פעולות כפל פולינומים יעילות. כמו כן, מוצגות תוצאות המראות את קיומם של פולינומים אי-פריקים מרווחים בשדות סופיים ממציין שתיים. המימוש המקבילי של התמרת הפוריה האדיטיבית בארכיטקטורת ה- CPU לא נתן תוצאות סקאלאביליות ולכך מספר סיבות שהינן מחוץ למסגרתה של עבודה זו. השערה שעלתה היא שלמימוש שניתן יש נטיה להיות לא סקאלאבילי על קלטים גדולים כיוון שישנם שלבים באלגוריתם בהם קלטים מסויימים הם פונקציה של מספר ערכים שנמצאים במקומות מרוחקים בזיכרון. גישה למקומות אלו בזיכרון עולה זמן יקר למעבד וגורמת להתנגשויות ברמות זיכרון המטמון השונות במעבד.

בעקבות הקושי שעלה מארכיטקטורת ה- CPU מוצג בעבודה זו מימוש סקאלאבילי מקבילי לארכיטקטורת ה- GPU של התמרת הפוריה האדיטיבית. לפני שנציג את תוצאות המחקר, נציג מודל פשטני של ארכיטקטורת ה- GPU. בארכיטקטורה זו יש מספר גדול מאוד של יחידות חישוב שנקראות "חוטים", לכל חוט יש זיכרון פרטי שנשמר בתוך אוגרים. מספר האוגרים מוגבל ולכן סך הזיכרון שחוט יכול לשמור בזיכרון הפרטי שלו מוגבל. החוטים מחולקים ל"קבוצות" כשכל קבוצה מכילה שלושים ושניים חוטים. הקבוצה פועלת כגוף אחד ולמעשה בכל זמן נתון כל החוטים בקבוצה מבצעים את אותה הפקודה, כתיבת תכנית בה חוטים מאותה הקבוצה לא מבצעים את אותה פקודה, למשל במקרה בו יש התניה בביצוע הפקודה שגורם לפיצול הביצוע בתכנית בין חוטים מאותה הקבוצה, תגרור מצב בו כל החוטים יבצעו

תקציר

העיסוק במימוש התמרות פוריה מהירות החל לפני כחמישים שנים ומאז הפך להיות חלק בלתי נפרד מן המחקר המדעי וההנדסי. באופן בסיסי, ניתן לומר שהתמרות פוריה דיסקרטית מתמירה ייצוג של פולינום כסכום פורמלי וסופי של מכפלת מקדמי הפולינום בחזקות של משתנה הפולינום בייצוג של פולינום בתוך הערך המתקבל ממנו על מספר נקודות הגדול בדיוק ב-1 מדרגת הפולינום. הצורך בהתמרות פוריה דיסקרטיות ובשיערוך של פולינומים מעל שדות סופיים נמצא בשימושים אלגבריים שונים דוגמת קידוד ריד-סולומון. שימוש נוסף שפיתוחו הוא אשר הוביל לעבודת מחקר זו הוא פיתוח של מערכת להוכחות שלמות חישובית קצרות. פרוטוקולים אלו מתקיימים בין יישות אחת הנקראת "מוכיח" ליישות אחרת הנקראת "מוודא". המוודא שולח למוכיח תכנית, קלט לתכנית וחסם עליון לזמן הריצה עבור התכנית. המוכיח בתורו מריץ את התכנית שקיבל למשך עד לעצירתה או עד להגעתה לחסם הצעדים ושולח את פלט התכנית אל המוודא. לאחר מכן מוכיח באופן הסתברותי ובאמצעות הוכחה קצרה כי הוא אכן ביצע את החישוב והפלט ששלח למוודא הוא הפלט שהתקבל מהחישוב. המוודא מוודא באופן יעיל את ההוכחה ומחליט האם לקבל או לדחות אותה. במהלך יצירת ההוכחה המוודא מייצר מעקב אחר תוכן הזיכרון לאורך ריצת התכנית ושולח קידוד כלשהו שלו למוודא במ-סגרת ההוכחה. הקידוד מחושב באמצעות התמרות פוריה מעל שדות סופיים ממציין שתיים. מבדיקות שנעשו במסגרת הפיתוח נראה כי צוואר הבקבוק המרכזי בחישוביו של המוכיח הם בשלב חישוב שיערוך הפולינום מערך תתי המרחבים האפיניים, חישוב שאפשר לבצע ביעילות באמצעות התמרות פוריה דיסקרטיות מעל שדות סופיים. התמרות פוריה היו בראשיתן מו-טיפליקטיביות (כפליות). הן הסתמכו על קיומה של תת-חבורה כפלית של שורשי יחידה מסדר כלשהו בשדה כלשהו. עם זאת, לשדות מסוימים ובפרט לשדות סופיים, אין שורשי יחידה מכל סדר שהוא, לצורך כך החל העיסוק בהתמרות פוריה אדיטיביות. התמרות פוריה אדיטיביות מתבססות על קיומן של תת-חבורות חיבוריות. התמרות פוריה אדיטיביות מתאימות במיוחד לשדות סופיים, שכן שדות אלו הינם בעלי מבנה אדיטיבי ובפרט בשדות סופיים ממציין 2, ישנן חבורות חיבוריות מכל גודל שהוא חזקה של 2. החקר של חישוב התמרות פוריה אדיטיביות בשדות סופיים החל בשנות השמונים. באותה העת פורסמו מספר אלגוריתמים בנושא אך מימושים של אותם אלגוריתמים הניבו ביצועים שאינם מניחים את הדעת. אחת הסיבות העי-קריות לכך הייתה שאלגוריתמים אלו דורשים מספר גבוה מדי באופן אסימפטוטי של כפלים בשדה סופי. מבדיקה עלה כי כפלים אלו צורכים את רוב זמן הריצה. אלגוריתמים אחרים שהוצעו אמנם נותנים מספר קטן יותר אסימפטוטית של כפלים בשדה הסופי, אך דורשים שהשדה הזו יהיה בעל תכונות מסויימות מאוד. מטרתו הראשית של מחקר זה הייתה מימוש יעיל של התמרות פוריה אדיטיביות מעל תתי מרחבים אפיניים בשדות סופיים ממציין 2. על

המחקר נעשה בהנחייתם של פרופסור אלי בן-ששון ופרופסור מרק זילברשטיין, בפקולטה למדעי המחשב.

אני מודה לקרן ע"ש הילדה ומנשה בן-שלמה ולטכניון על התמיכה הכספית הנדיבה בהשתל-מותי.

אלגוריתמים מקביליים לחישוב התמרות פוריה אדיטיביות

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
מדעי המחשב

מתן חמיליס

הוגש לסנט הטכניון --- מכון טכנולוגי לישראל
תמוז התשע"ו חיפה יולי 2016

אלגוריתמים מקביליים לחישוב התמרות פוריה אדיטיביות

מתן חמיליס