2012

GPU and ASIC Acceleration of Elliptic Curve Scalar Point Multiplication

Karl Bernard Leboeuf
University of Windsor

GPU and ASIC Acceleration of Elliptic Curve Scalar Point Multiplication

by

Karl Leboeuf

A Dissertation
Submitted to the Faculty of Graduate Studies through the Department of Electrical and Computer Engineering in Partial Fulfilment of the Requirements for the Degree of Doctor of Philosophy at the University of Windsor

Windsor, Ontario, Canada
2012

GPU and ASIC Acceleration of Elliptic Curve Scalar Point Multiplication

by

Karl Leboeuf

APPROVED BY:

Dr. A. Reyhani-Masoleh, External Examiner
University of Western Ontario

Dr. L. Rueda
Computer Science

Dr. M. Mir
Electrical and Computer Engineering

Dr. C. Chen
Electrical and Computer Engineering

Dr. R. Muscedere, Co-Advisor
Electrical and Computer Engineering

Dr. A. Ahmadi, Co-Advisor
Electrical and Computer Engineering

Dr. A. Lanoszka, Char of Defense
Political Science

September 26, 2012

Declaration of Co-Authorship / Previous Publication

I. Co-Authorship Declaration

I hereby declare that this thesis incorporates material that is result of joint research, as follows:

This thesis also incorporates the outcome of a joint research undertaken in collaboration with Ashkan Hosseinzadeh Namin under the supervision of Dr. M. Ahmadi, Dr. H. Wu, and Dr. R. Muscedere. The collaboration is covered in Chapter 3 of the thesis. In all cases, schematics, layouts, data analysis and interpretation, were performed by the author.

I am aware of the University of Windsor Senate Policy on Authorship and I certify that I have properly acknowledged the contribution of other researchers to my thesis, and have obtained written permission from each of the co-author(s) to include the above material(s) in my thesis. I certify that, with the above qualification, this thesis, and the research to which it refers, is the product of my own work.

II. Declaration of Previous Publication

This thesis includes 3 original papers that have been previously published in peer reviewed journals and conferences, as follows:

Thesis Chapter	**Publication Title**	**Publication Status**
Chapter 3	High-speed hardware implementation of a serial-in paralle-out finite field multiplier using reordered normal basis	Published [1]
Chapter 4	Efficient VLSI implementation of a finite field multiplier using reordered normal basis	Published [2]
Chapter 7	High performance prime field multiplication for GPU	Published [3]

I certify that I have obtained a written permission from the copyright owner(s) to include the above published material(s) in my thesis. I certify that the above material describes work completed during my registration as graduate student at the University of Windsor.

I declare that, to the best of my knowledge, my thesis does not infringe upon anyones copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis. I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

Abstract

As public information is increasingly communicated across public networks such as the internet, the use of public key cryptography to provide security services such as authentication, data integrity, and non-repudiation is ever-growing.

Elliptic curve cryptography is being used now more than ever to fulfill the need for public key cryptography, as it provides security equivalent in strength to the entrenched RSA cryptography algorithm, but with much smaller key sizes and reduced computational cost.

All elliptic curve cryptography operations rely on elliptic curve scalar point multiplication. In turn, scalar point multiplication depends heavily on finite field multiplication.

In this dissertation, two major approaches are taken to accelerate the performance of scalar point multiplication. First, a series of very high performance finite field multiplier architectures have been implemented using domino logic in a CMOS process. Simulation results show that the proposed implementations are more efficient than similar designs in the literature when considering area and delay as performance metrics. The proposed implementations are suitable for integration with a CPU in order to provide a special-purpose finite field multiplication instruction useful for accelerating scalar point multiplication.

The next major part of this thesis focuses on the use of consumer computer graphics cards to directly accelerate scalar point multiplication. A number of finite field multiplication algorithms suitable for graphics cards are developed, along with algorithms for finite field addition, subtraction, squaring, and inversion. The proposed graphics-card finite field arithmetic library is used to accelerate elliptic curve scalar point multiplication. The operation throughput and latency performance of the proposed implementation is characterized by a series of tests, and results are compared to the state of the art. Finally, it is shown that graphics cards can be used to significantly increase the operation throughput of scalar point multiplication operations, which makes their use viable for improving elliptic curve cryptography performance in a high-demand server environment.

Dedicated to my wife, Christine

Acknowledgements

I would like to thank my supervisors, Dr. Roberto Muscedere and Dr. Majid Ahmadi for their continued help and guidance during my entire time as a graduate student, and also Dr. Wu for helping me with some of the mathematics needed for this work.

Several other people I would like to thank include my committee members, Dr. Rueda, Dr. Mir, and Dr. Chen, for attending my seminars, their constructive comments, and for their advice.

A very special thanks is required for Adria Ballo and Shelby Marchand for helping me with what now seems like a little bit of everything during my time as a graduate student.

In addition to my parents and family, I would also like to thank my friends and colleagues in the RCIM lab at the University of Windsor, which includes (but not limited to!) Ashkan Hosseinzadeh Namin, Golnar Khodabandehloo, Iman Makaremi, Farinoush Saffar, Mahzad Azarmehr, and the Moush-ha.

Contents

List of Tables

List of Figures

List of Abbreviations

ALU	Arithmetic Logic Unit
ANSI	American National Standards Institute
ASIC	Application Specific Integrated Circuit
CMOS	Complimentary Metal-Oxide-Semiconductor
CPU	Central Processing Unit
DLL	Delay-Locked Loop
EC	Elliptic Curve
ECC	Elliptic Curve Cryptography
ECDH	Elliptic Curve Diffie-Hellman
ECDLP	Elliptic Curve Discrete Logarithm Problem
ECDSA	Elliptic Curve Digital Signature Algorithm
EM	Electro-Magnetic
FIPS	Federal Information Processing Standard
FPGA	Field Programmable Gate Array
GNB	Gaussian Normal Basis
GPU	Graphics Processing Unit
HDL	Hardware Description Language
IC	Integrated Circuit

IEEE	Institute of Electrical and Electronics Engineers
NAND	Not-AND
NAF	Non-adjacent form
NB	Normal Basis
NIST	National Institute of Standards and Technology
NMOS	n-Channel MOSFET
NSA	National Security Agency
ONB	Optimal Normal Basis
PLL	Phase-Locked Loop
PMOS	p-Channel MOSFET
RAM	Random Access Memory
RIM	Research In Motion
RNB	Reordered Normal Basis
RNS	Residue Number System
ROM	Read Only Memory
SM	Streaming Multiprocessor
SPICE	Simulation Program with Integrated Circuit Emphasis
SSL	Secure Socket Layer
TLS	Transport Layer Security
TSMC	Taiwan Semiconductor Manufacturing Company
TSPC	True Single Phase Clock
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integration
XOR	Exclusive-OR

CHAPTER 1

Introduction

1.1 Motivation

The amount of private information transmitted over public networks such as the internet is ever-increasing. A truly massive number of people use the internet on a daily basis for business and pleasure, and activities range from checking e-mail and social networking websites, to the use of on-line banking and search engines. The majority of the data traveling over these networks are unencrypted and vulnerable to potential eavesdroppers, however this is starting to change as end users begin to realize that security vulnerabilities can and do affect them. For example, in October 2010, a Firefox web browser extension called "Firesheep" was developed[7, 8], which allowed a (possibly malicious) user on a Wi-Fi network unfettered access to other network users' on-line accounts such as Facebook, Twitter, Hotmail, and virtually any other website that did not use end-to-end encryption.

Within a few months, many major websites began offering the option of end-to-end

encryption, while many others are in the process of adding this functionality. Notably, starting in January 2010, Google began encrypting its e-mail traffic by default, while in February 2012, Twitter has enabled encryption by default for its services as well. Facebook added support for end-to-end encryption in January 2011, and they claim to be working on making this the default setting within a year.

1.1.1 Transport layer security & public key cryptography

End-to-end encryption ensures the confidentiality and integrity of transmitted information by encrypting it before it is placed on a public network, and decrypting it after it arrives at its destination; in practice this is typically achieved with the transport layer security (TLS) protocol. The most time consuming part of a TLS transaction is the underlying public key cryptography operations allowing a server to prove its identity to and exchange a session key with the client.

As end-users are beginning to place greater value on their data security and privacy, the demand for end-to-end encryption and its relatively expensive cryptography operations also increases, placing a greater burden on on-line service providers' servers, driving up costs.

1.1.2 RSA key sizes

Further increasing the computational burden of internet security, the minimum recommended key size for the popular RSA public key cryptography algorithm has recently doubled from 1024 to 2048 bits; the importance for this key strength upgrade is underlined by the fact that in 2009 a group of researchers have successfully factored (broken) a 768-bit RSA modulus in about two thousand CPU-years, or just short of three years of calendar time with the computing resources available to them [9].

1.1.3 Moore's law & serial processing performance

Meanwhile, Moore's law (which states that the number of transistors in an integrated circuit doubles every 18 to 24 months) continues to hold, however the microelectronics industry has effectively hit a wall in terms of CPU power consumption: after a process shrink, hardware designers can use more transistors in an integrated circuit (IC), however these transistors will not necessarily operate any faster than before due to heat dissipation problems.

The challenges of increased demand, larger key size requirements for RSA, and the inability to simply operate existing architectures at higher clock speeds has lead to some interesting developments.

1.2 Solutions

1.2.1 Elliptic curve cryptography

A public key crytosystem employing elliptic curves was independently proposed by Neal Koblitz [10] and Victor Miller [11] in 1987 and 1985, respectively. Elliptic curve cryptography (ECC) can implement the same functionality as RSA while using significantly smaller key sizes [12, 13, 14, 15, 16], which results in fewer clock cycles and reduced hardware costs.

In 2012, the recommended RSA key size is 2048 bits, while it is only 224 bits for ECC. Recommended key sizes for the year 2050 are 7680 for RSA and 384 for ECC. Furthermore, the United States' National Security Agency (NSA) estimates that the computational cost of RSA compared to ECC at the current recommended security level is 6 to 1, while this gap only increases as security requirements and key lengths inevitably grow.

ECC was being adopted rather slowly aside from certain key devices, notably the

Year	Recommended Key Size [14]		RSA Cost : EC Cost [17]
	RSA	Elliptic Curve	
2020	2048	224	6:1
2030	2048	224	6:1
2040	3072	256	10:1
2050	7680	384	32:1

Table 1.1: Comparison of recommended public key sizes and estimated computational cost for elliptic curve and RSA cryptography

RIM Blackberry smartphones. This is likely due to the fact that RSA was already well-established, and also because ECC is often perceived as being patent encumbered. Due to the combination of larger RSA key sizes, and a host of other factors well beyond the scope of this dissertation, ECC will most likely replace RSA public key cryptography for most applications in the coming years [18].

1.2.2 Parallel processing

As for difficulties concerning stagnant CPU clock rates, there are two major approaches the microelectronics industry is taking to put the additional transistors afforded by process miniaturization to work. First, it has always been possible to develop special purpose circuits to accelerate commonly used CPU tasks. As of 2008, for example, Intel and AMD have been including special hardware to accelerate AES block encryption, which is accessed using special instructions [19].

The second major approach for employing more transistors without significantly increasing power consumption is parallel and massively parallel processing. It is not uncommon for a desktop CPU to possess four or even eight physical cores, while graphics processors may possess hundreds.

1.3 Research goals, and the organization of this dissertation

The overarching goal of this work is to improve the operation throughput of elliptic curve scalar point multiplication, which is the key operation used in all elliptic curve cryptography algorithms. An overview of the mathematical foundation upon which elliptic curve cryptography and elliptic curve scalar point multiplication is based is presented in Chapter 2, along with a brief review of important cryptography protocols and standards.

First efforts towards the acceleration of elliptic curve scalar point multiplication began with the VLSI implementation of a recently proposed finite field reordered normal basis multiplier architecture [4], which, in turn, would lead to greatly improved scalar point multiplication performance once integrated into a CPU or SOC. Chapter 3 presents the CMOS $0.18\mu m$ implementation of this multiplier, which uses a combination of custom domino logic and standard VLSI library cells to achieve excellent performance compared to the state of the art. In Chapter 4 the proposed finite field multiplier was further improved upon by making some architectural changes, and by implementing the entire design in domino logic, further reducing its critical path delay as well as its area utilization.

The planned research goals at this stage were as follows:

1. Integrate the proposed multiplier into a CPU core, fabricate it, and measure its results before any further improvements are carried out. This step is critical in order to determine if there are any unforeseen issues relating to the design that require correction, such as antenna effects, hot spots, or EM noise issues.

2. Further generalize the design so that it may carry out finite field multiplication over a wider array of field sizes; essentially allow the proposed multiplier to be used for Gaussian Normal Bases.

At this point, however, some developments transpired which altered the course of this work:

1. Practical limitations: CPUs IP cores were not (and are not) available to the University of Windsor; this eliminates the chance to actually integrate the proposed custom multiplier into a CPU design for use in accelerating cryptography operations. Further, it is impractical to design a custom CPU capable of running even a very minimalist operating system. The ultimate goal of integrating a custom multiplier design into a common CPU is thus out of reach.

2. Limited library components: fabricating and testing the multiplier on its own at the high operating speeds for which it was designed requires components such as the phase locked loops (PLLs) or delay locked loops (DLLs), which are not available in any available library. Creating these components is possible, but time consuming and not considered a research problem.

3. A new and interesting avenue of research for improving scalar point multiplication was made available: graphic processing unit (GPU) acceleration. Until recently, GPUs were used almost exclusively to render graphics for computer games. Now, however, it is possible to use them to accelerate general purpose computations. This aligned well with the major goal of this dissertation, which is to determine a practical way to accelerate the performance of elliptic curve scalar point multiplication, and as such, the remainder of this dissertation focuses on this new and interesting area of research.

Chapter 5 presents a review GPU computing, and following this Chapter 6 proposes a type-II optimal normal basis multiplication algorithm for the GPU. Compared to other GPU-based binary extension field multiplication algorithms, the proposed work performs significantly better in terms of operation throughput in multiplications

per second, however it is not able to surpass a CPU implementation which makes use of some recently released special-purpose instructions.

Noting that the GPU possesses extremely efficient integer multiplication, attention was refocused on developing prime field multiplication algorithms. Chapter 7 presents a NIST-fields multiplication algorithm for GPU that, to the best of the author's knowledge, is the fastest CPU or GPU based multiplication algorithm reported in the literature.

In Chapter 8, a number of analyses were performed on the multiplication stage used by the NIST multiplication algorithm, and it was determined that a very high throughput Montgomery multiplication algorithm could be developed which maintains high operation throughput while allowing reduction over any finite field. This chapter proposes a complete finite field arithmetic library based on the Montgomery multiplication algorithm, which includes a finite field inversion algorithm based on Fermat's little theorem that is suitable for GPU implementations. The resulting library's operation throughput performance is analyzed and compared to the state of the art.

Chapter 9 presents the proposed GPU-based elliptic curve scalar point multiplication algorithm, along with a series of performance analyses which characterize the proposed in terms of operation throughput, latency, and batch size requirements. The proposed scalar point multiplication is compared to the state of the art, and it is shown that it boasts between 5x and 31x greater operation throughput than the next best CPU implementation, and 6x to 7.7x greater operation throughput than the state-of-the-art FPGA implementation.

Finally, the contributions of the work presented in this dissertation are highlighted in Chapter 10, along with some potential future work.

CHAPTER 2

Mathematical Preliminaries

2.1 Introduction

The main goal of the work presented in this dissertation is to improve the operation throughput of the elliptic curve scalar point multiplication operation that is used in internet security protocols such as TLS and SSL which employ elliptic curve cryptography algorithms. As shown in Figure 2.1, scalar point multiplication depends on point addition and point doubling operations, which in turn require fundamental finite field arithmetic operations such as multiplication, addition, subtraction, squaring, and inversion.

This chapter presents a brief, bottom-up summary of elliptic curve cryptography, beginning with its underlying finite field arithmetic (the bottom of the hierarchy in Figure 2.1), followed by elliptic curve group law, scalar multiplication, and high-level elliptic curve cryptography algorithms, which is second from the top in Figure 2.1.

Sections 2.2 and 2.3 review the mathematical concept of groups and fields, fol-

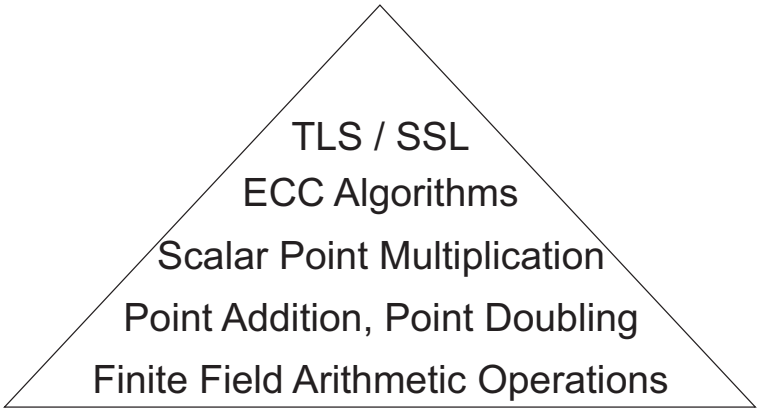


Figure 2.1: Hierarchy of operations for implementing internet security protocols with elliptic curve cryptography

lowed by finite fields and extension fields in sections 2.4 and 2.4.2. In section 2.4.4 several important bases for binary extension fields are presented and compared. Section 2.4.6 summarizes the different fields and bases used in the work presented in this dissertation. Following this, elliptic curves and the elliptic curve group law is introduced in section 2.5. Section 2.6 presents scalar point multiplication, and 2.8 reviews the high level protocols that ultimately carry out security services. Section 2.9 presents some concluding remarks.

2.2 Groups

A group is defined as a set $\mathbf{G}$ together with an operator '$\bullet$' that combines two elements in $\mathbf{G}$ to form a third element also in $\mathbf{G}$, and satisfies the following four properties [20]:

Property 2.1 (Closure)

$a, b \in \mathbf{G}$ implies that $a \bullet b \in \mathbf{G}$

Property 2.2 (Associativity)

$a, b, c, \in \mathbf{G}$ implies that $a \bullet (b \bullet c) = (a \bullet b) \bullet c$

Property 2.3 (Identity element)

There exists an element $0 \in \mathbf{G}$ such that $a \bullet 0 = 0 \bullet a = a$ for all $a \in \mathbf{G}$

Property 2.4 (Inverse element)

For every $a \in \mathbf{G}$ there exists an element $a^{-1} \in \mathbf{G}$ such that $a \bullet a^{-1} = a^{-1} \bullet a = 0$

Abelian groups

An *abelian* group is a group that also satisfies the commutativity property:

Property 2.5 (Commutativity)

A group $\mathbf{G}$ is said to be abelian (commutative) if for every $a, b, \in \mathbf{G}, a \bullet b = b \bullet a$

The most common example of a group is perhaps the integers ($\aleph$) together with the addition ($+$) operation.

2.3 Fields

A field is a set $\mathbb{F}$ together with two operators, often denoted as $\bullet$ and $*$ which combine two elements in $\mathbb{F}$ to form a third element in $\mathbb{F}$ that satisfies the following seven properties in addition to the five abelian group properties [21]:

Property 2.6 (Closure under multiplication)

$a, b \in \mathbb{F}$ implies that $a * b \in \mathbf{G}$

Property 2.7 (Associativity of multiplication)

$a, b, c, \in \mathbb{F}$ implies that $a * (b * c) = (a * b) * c$

Property 2.8 (Distributivity)

$a * (b \bullet c) = a * b \bullet a * ca$ for $a, b, c \in \mathbb{F}$

$(a * b) \bullet c = a * b \bullet a * ca$ for $a, b, c \in \mathbb{F}$

Property 2.9 (Commutativity of multiplication)

$a * b = b * a$ for $a, b, \in \mathbb{F}$

Property 2.10 (Multiplicative identity)

There exists an element $1 \in \mathbb{F}$ such that $a \bullet 1 = 1 \bullet a = a$ for all $a \in \mathbb{F}$

Property 2.11 (No zero divisors)

If $a, b \in \mathbb{F}$ and $a * b = 0$, then either $a = 0$ or $b = 0$

Property 2.12 (Multiplicative inverse element)

If $a \in \mathbb{F}$ and $a \neq 0$, then there is an element a^{-1} in $\mathbf{G}$ such that $a * a^{-1} = a^{-1} * a = 1$

An example of a field is the set of real numbers $\Re$, together with the standard addition ($+$) and multiplication ($\times$) operations.

2.4 Finite fields

A finite field possesses all the properties of a field, with the additional constraint that its set contains a *finite* number of elements [22].

2.4.1 Prime fields

The set of integers $[0, p-1]$ for p prime, together with field operations defined as addition and multiplication modulo p, form a finite field which is denoted as either $\mathbb{F}_p$ or equivalently as GF(p), in honour of Evariste Galois [23]. The field prime 'p' is called the characteristic of the field, and the number of elements in the field (more properly stated as the field *order*) is also p. The fields GF(p) for different primes p may also be referred to as the prime fields.

2.4.2 Extension fields

It is possible to create an m-dimensional vector space from the elements in $\mathbb{F}_p$, which is equivalently denoted as either $\mathbb{F}_{p^m}$ or GF(p^m). This vector space is called an extension field, and it has a field order of p^m, and characteristic p. It can be shown that *all* finite fields are isomorphic (structurally equivalent) to $\mathbb{F}_{p^m}$, and there are many different, mathematically equivalent ways to define a *basis*, that is, a set of linearly independent

vectors that span the space, allowing for different implementation advantages and disadvantages [20].

2.4.3 Polynomial basis arithmetic

The most common basis for extension fields is polynomial (or standard) basis. In this case, the elements are polynomials of degree at most $m-1$ with coefficients in $\mathbb{F}_p$, and operations are polynomial addition and multiplication modulo a primitive degree-m polynomial [20]. If α is a root of the degree-m primitive polynomial which defines the field, a polynomial can be defined as [20]:

$$\{1,\ \alpha,\ \alpha^2,\ \ldots,\ \alpha^{m-1}\} \tag{2.1}$$

and an element A in GF(p^m) can be represented as

$$A = \{a_0 + a_1\alpha + a_2\alpha^2 + \ldots + a_{m-1}\alpha^{m-1}\} \quad a_i \ \in \ \mathbb{F}_p, 0 \leq i < m \tag{2.2}$$

or equivalently using sigma notation as

$$A = \sum_{i=0}^{m-1} a_i\alpha^i, \quad a_i \ \in \ \mathbb{F}_p \tag{2.3}$$

A particularly important family of extension fields are the characteristic-2 or binary extension fields, denoted as $\mathbb{F}_{2^m}$, or GF(2^m). Binary extension fields are of great interest as they are especially well suited for implementation using the binary logic employed by virtually all computer hardware. In this case, the coefficients of A in Equation 2.3 are either 0 or 1, and adding elements A and B in a field over GF(2) is simply an m-bit wide exclusive-or (XOR) operation. Note that this is a carry-less operation, allowing for easy parallel implementation. As previously stated, multiplication over polynomial basis is defined as common polynomial multiplication, modulo

the primitive polynomial that defines the field. While there may exist more than one primitive polynomial over a field GF(p^m) for a specific 'p' and 'm', these different polynomials construct fields that are structurally equivalent or *isomorphic* with respect to each other. It is possible, however, that certain primitive polynomials lead to more efficient implementations, such as the use of all-one polynomials (AOPs), or equally spaced polynomials (ESPs) [24, 25].

Polynomial basis is especially popular for software implementations; compared to alternative bases (such as those presented in the next section), they require far fewer multi-machine-word shift operations, and fewer instructions overall. Emphasizing the popularity of polynomial basis for software implementations, Intel has recently included a special, dedicated instruction "PCLMULQDQ", which is included in the majority of their desktop and server CPUs released since 2010, can be used perform polynomial basis multiplication very efficiently [26].

Although there are a number of different polynomial basis arithmetic algorithms exist for both hardware and software platforms, they are beyond the scope of this work, which concentrates on prime fields, as well as other bases in GF(2^m), such as normal basis, Gaussian normal basis, and reordered normal basis, which possess numerous advantages in hardware and parallel implementations.

2.4.4 Normal basis, optimal normal basis, & Gaussian normal basis

A basis of $\mathbb{F}_{2^m}$ over $\mathbb{F}_2$ of the form $N = \{\beta^{2^0},\ \beta^{2^1},\ \beta^{2^2},\ \ldots,\ \beta^{2^{m-1}}\}$ for an appropriate choice of $\beta \in \mathbb{F}_{2^m}$ is called a normal basis (NB) [22]. A normal basis can be found for every finite field $\mathbb{F}_{2^m}$ [22]. Field elements can be represented as an m-dimensional ordered set $(a_0,\ a_1,\ \ldots,\ a_{m-1})$, with coefficients a_i in $\mathbb{F}_2$, and basis $N = \{\beta,\ \beta^q,\ \beta^{q2},\ \ldots,\ \beta^{q^{m-1}}\}$ spanning $\mathbb{F}_{2^m}$; this is shown compactly using sigma notation as shown in Equation 2.4. In software implementations, NB elements are

represented with an array of $\lceil \frac{m}{w} \rceil$ w-bit binary words, whose bits are interpreted as the coefficients a_i in Equation 2.4.

$$A = \sum_{i=0}^{m-1} a_i \beta^{2^i} \tag{2.4}$$

Normal basis addition over binary extension fields is carried out using the same m-wide XOR operation used by polynomial basis. A squaring operation is especially inexpensive in NB, as it consists of a single circular shift operation, which can be implemented in hardware at virtually no cost. Multiplication is more complicated, and requires a combination of circular shift, and logicical XOR and AND operations.

Normal basis multiplication

Given two elements A and B in normal basis defined as in Equation 2.4, a third element C can be computed as shown in Equation 2.6:

$$C = A \times B = \sum_{i=0}^{m-1} a_i \beta^{2^i} \times \sum_{j=0}^{m-1} b_j \beta^{2^j} \tag{2.5}$$

$$= \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} a_i b_j \beta^{2^i} \beta^{2^j} \tag{2.6}$$

$$= \sum_{k=0}^{m-1} c_k \beta^{2^k} \tag{2.7}$$

Note that the product of the double sum $\sum_{i=0}^{m-1} \sum_{j=0}^{m-1} \beta^{2^i} \beta^{2^j}$ in Equation 2.6 must map to the single sum $\sum_{i=0}^{m-1} \beta^{2^k}$ in Equation 2.7. It is possible to create a multiplication table λ_{ijk} for each combination of $i,\ j$ in Equation 2.6 to determine the sum $\sum_{i=0}^{m-1} \beta^{2^k}$; one method of generating λ_{ijk} is presented in [27]. Shown in the five rightmost columns of Table 2.1 is the multiplication table λ_{ijk} for the finite field GF(2^5) generated by the primitive polynomial $f(x) = x^5 + x^4 + x^2 + x + 1$ and $\beta = \alpha^{11}$, where

α is a root of the primitive polynomial.

Table 2.1: Normal basis multiplication table for $\mathbb{F}_{2^5}$ with primitive polynomial $x^5 + x^4 + x^2 + x + 1$, and $\beta = \alpha^{11}$ where α is a root of the primitive polynomial

i	j	β^{2^i}					β^{2^j}					$\beta^{2^k} = \beta^{2^i} \times \beta^{2^j}$				
		β^{2^4}	β^{2^3}	β^{2^2}	β^{2^1}	β^{2^0}	β^{2^4}	β^{2^3}	β^{2^2}	β^{2^1}	β^{2^0}	β^{2^4}	β^{2^3}	β^{2^2}	β^{2^1}	β^{2^0}
0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0
0	1	0	0	0	0	1	0	0	0	1	0	0	1	1	1	0
0	2	0	0	0	0	1	0	0	1	0	0	1	0	1	1	1
0	3	0	0	0	0	1	0	1	0	0	0	1	1	1	0	1
0	4	0	0	0	0	1	1	0	0	0	0	0	0	1	1	1
1	0	0	0	0	1	0	0	0	0	0	1	0	1	1	1	0
1	1	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0
1	2	0	0	0	1	0	0	0	1	0	0	1	1	1	0	0
1	3	0	0	0	1	0	0	1	0	0	0	0	1	1	1	1
1	4	0	0	0	1	0	1	0	0	0	0	1	1	0	1	1
2	0	0	0	1	0	0	0	0	0	0	1	1	0	1	1	1
2	1	0	0	1	0	0	0	0	0	1	0	1	1	1	0	0
2	2	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0
2	3	0	0	1	0	0	0	1	0	0	0	1	1	0	0	1
2	4	0	0	1	0	0	1	0	0	0	0	1	1	1	1	0
3	0	0	1	0	0	0	0	0	0	0	1	1	1	1	0	1
3	1	0	1	0	0	0	0	0	0	1	0	0	1	1	1	1
3	2	0	1	0	0	0	0	0	1	0	0	1	1	0	0	1
3	3	0	1	0	0	0	0	1	0	0	0	1	0	0	0	0
3	4	0	1	0	0	0	1	0	0	0	0	1	0	0	1	1
4	0	1	0	0	0	0	0	0	0	0	1	0	0	1	1	1
4	1	1	0	0	0	0	0	0	0	1	0	1	1	0	1	1
4	2	1	0	0	0	0	0	0	1	0	0	1	1	1	1	0
4	3	1	0	0	0	0	0	1	0	0	0	1	0	0	1	1
4	4	1	0	0	0	0	1	0	0	0	0	0	0	0	0	1

In [28] it was shown that it is possible to use the λ table's column $k = 0$, while shifting input operands A and B in order to compute the product $C = A \times B$. The individual bits of C may be computed as shown in Equation 2.8:

$$c_k = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} a_{i+k} b_{j+k} \lambda_{ij0} \tag{2.8}$$

Optimal normal basis multiplication

The λ_{ij0} table for the complete multiplication table 2.1, is shown in Table 2.2. Note that the number of non-zero terms in this case is 15; this quantity is called the

Table 2.2: Multiplication table λ_{ij0} for the example in Table 2.1

i \ j	0	1	2	3	4
0	0	0	1	1	1
1	0	0	0	1	1
2	1	0	0	1	0
3	1	1	1	0	1
4	1	1	0	1	1

table's *complexity*, and is denoted C_N; when Equation 2.8 is expanded, it will require C_N terms. Different normal bases may have different complexities, and it would be advantageous to use the normal basis with the minimum possible complexity in order to reduce the number of terms in the expanded form of Equation 2.8.

Mullin et al. did exactly this, and determined that the minimum complexity of a normal basis over a field $\mathbb{F}_{2^m}$ is $C_N = 2m - 1$ [28]. Additionally, they determined when such a basis exists, and how to construct such a basis, which is referred to as an optimal normal basis (ONB) [28]. The authors also note that there are two types of ONB; in practice only type-II ONB is used, as type-I ONB only exist for certain *even* extension degrees (m), as there is some concern that there may be undiscovered methods that exploits for fields with even extension degrees.

To highlight the importance of ONB, Table 2.3 presents the complete multiplication table for the type-II ONB for the same field shown in the previous example, $f(x) = x^5 + x^4 + x^2 + x + 1$, however this time choosing $\beta = \alpha$ instead of $\beta = \alpha^{11}$. This changes the complexity C_N from 15 to $C_N = 2m - 1 = 9$, greatly reducing the number of terms in the expanded form of Equation 2.8; the difference between a randomly chosen NB and an ONB grows significantly for the larger field sizes that are of interest for elliptic curve cryptography.

Table 2.3: Type-II optimal normal basis multiplication table for $\mathbb{F}_{2^5}$ with primitive polynomial $\alpha^5 + \alpha^4 + \alpha^2 + \alpha + 1$, and $\beta = \alpha$

i	j	β^{2^i}					β^{2^j}					$\beta^{2^k} = \beta^{2^i} \times \beta^{2^j}$				
		β^{2^4}	β^{2^3}	β^{2^2}	β^{2^1}	β^{2^0}	β^{2^4}	β^{2^3}	β^{2^2}	β^{2^1}	β^{2^0}	β^{2^4}	β^{2^3}	β^{2^2}	β^{2^1}	β^{2^0}
0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0
0	1	0	0	0	0	1	0	0	0	1	0	0	1	0	0	1
0	2	0	0	0	0	1	0	0	1	0	0	1	1	0	0	0
0	3	0	0	0	0	1	0	1	0	0	0	0	0	1	1	0
0	4	0	0	0	0	1	1	0	0	0	0	1	0	1	0	0
1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	0	1
1	1	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0
1	2	0	0	0	1	0	0	0	1	0	0	1	0	0	1	0
1	3	0	0	0	1	0	0	1	0	0	0	1	0	0	0	1
1	4	0	0	0	1	0	1	0	0	0	0	0	1	1	0	0
2	0	0	0	1	0	0	0	0	0	0	1	1	1	0	0	0
2	1	0	0	1	0	0	0	0	0	1	0	1	0	0	1	0
2	2	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0
2	3	0	0	1	0	0	0	1	0	0	0	0	0	1	0	1
2	4	0	0	1	0	0	1	0	0	0	0	0	0	0	1	1
3	0	0	1	0	0	0	0	0	0	0	1	0	0	1	1	0
3	1	0	1	0	0	0	0	0	0	1	0	1	0	0	0	1
3	2	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1
3	3	0	1	0	0	0	0	1	0	0	0	1	0	0	0	0
3	4	0	1	0	0	0	1	0	0	0	0	0	1	0	1	0
4	0	1	0	0	0	0	0	0	0	0	1	1	0	1	0	0
4	1	1	0	0	0	0	0	0	0	1	0	0	1	1	0	0
4	2	1	0	0	0	0	0	0	1	0	0	0	0	0	1	1
4	3	1	0	0	0	0	0	1	0	0	0	0	1	0	1	0
4	4	1	0	0	0	0	1	0	0	0	0	0	0	0	0	1

Gaussian normal basis

Unfortunately, an optimal normal basis does not exist for all field sizes, which limits its use compared to polynomial basis, where a primitive pentanomial or trinomial can always be found. To ameliorate this, Ash et al. developed the concept of type-T Gaussian normal bases (GNB), and demonstrated that type-I and II GNB are equivalent to type-II and II ONB [29]. Higher types GNB are more computationally expensive (or require greater hardware resources) compared to lower types, however they guarantee the lowest complexity possible for a given field $\mathbb{F}_{2^m}$. A type-T GNB exists for all fields $\mathbb{F}_{2^m}$ except those with m divisible by 8 [30].

2.4.5 Reordered normal basis multiplication

Reordered normal basis (RNB) was proposed by Wu et al. [4], using some ideas from Gao et al.[31]. RNB is a permutation of type II ONB; the complexity of RNB is equivalent to that of ONB, however RNB allows for very regular multiplier architectures which are especially amenable to hardware implementations, as signal routing is greatly simplified. Specifically, RNB multiplication is carried out using Equation 2.14, whose derivation shown below is reproduced here from [4] for completeness.

Theorem 2.1

Let β be a primitive $(2m+1)^{\text{st}}$ root of unity in $\mathbb{F}_{2^m}$ and $\gamma = \beta + \beta^{-1}$ generates a type II optimal normal basis. Then $\{\gamma_i, i = 1, 2, \ldots, m\}$ with $\gamma_i = \beta^i + \beta^{-i} = \beta^i + \beta^{2m+1-i}$, $i = 1, 2, \ldots, m$ is also a basis in $\mathbb{F}_{2^m}$

For $\beta \in \mathbb{F}_{2^m}$ and γ_i as defined in theorem 2.1, define

$$s(i) \triangleq \begin{cases} i \bmod 2m+1, & \text{if } 0 \leqslant i \bmod 2m+1 \leqslant m, \\ 2m+1-i \bmod 2m+1, & \text{otherwise} \end{cases} \tag{2.9}$$

Now, $s(0) = 0, s(i) = s(2m+1-i)$, and $\gamma_i = \gamma_{s(i)}$ for any integer i. As $\gamma_i\gamma_j = \gamma_{i+j} + \gamma_{i-j}$, thus $\gamma_i \cdot \gamma_j = \gamma_{s(i+j)} + \gamma_{s(i-j)}$. Let $B = (b_1, \ldots, b_m) \in \mathbb{F}_{2^m}$ with respect to the basis $[\gamma_1, \gamma_2, \ldots, \gamma_m]$ and $b_0 = 0$ then

$$\gamma_i \cdot B = \sum_{j=1}^{m} b_j \gamma_i \cdot \gamma_j = \sum_{j=1}^{m} b_j(\gamma_{s(i+j)} + \gamma_{s(i-j)}) \tag{2.10}$$

$$= \sum_{j=1}^{m} (b_{s(j+i)} + b_{s(j-i)})\gamma_j \tag{2.11}$$

The final step in the equation above comes from proper substitutions of the subscript variables. The above constant multiplication $\gamma_i \cdot B$ was proposed by Gao et al. in [31]. In order to obtain a general multiplier, let $A = (a_1, \ldots, a_m)$ be an element in

$\mathbb{F}_{2^m}$ with respect to the basis $[\gamma_1,\ \gamma_2,\ \ldots,\ \gamma_m]$, then multiplication of A and B can proceed as follows:

$$A \cdot B = \sum_{i=1}^{m} a_i(\gamma_i \cdot B) = \sum_{i=1}^{m} a_i \sum_{j=1}^{m} (b_{s(j+i)} + b_{s(j-i)})\gamma_j \tag{2.12}$$

$$= \sum_{j=1}^{m} \Big(\sum_{i=1}^{m} a_i (b_{s(j+i)} + b_{s(j-1)}) \Big) \gamma_j \tag{2.13}$$

If the product is written as $C = \sum_{j=1}^{m} c_j \gamma_j$, then

$$c_j = \sum_{i=1}^{m} a_i (b_{s(j+i)} + b_{s(j-i)}), j = 1, 2,\ \ldots,\ m \tag{2.14}$$

The λ table required for NB and ONB multiplication is irregular, and varies with different fields $\mathbb{F}_{2^m}$ and choices for element β, which can complicate hardware designs by adding to routing overhead. This also causes software algorithms to require irregular memory access patterns. RNB multiplication using Equation 2.14 does not require such a table, and some very regular hardware architectures have been proposed to take advantage of this property [4]. RNB squaring requires a simple permutation of the bits making up the operands, which is implemented inexpensively in hardware by "shuffling" wires. Software implementations of RNB squaring could be significantly more expensive without dedicated hardware such as a multi-precision barrel shifter.

2.4.6 Summary of finite fields

Figure 2.2 presents a simple taxonomy of the finite fields presented in this section. To summarize, all finite fields are of the form $\mathbb{F}_{q^m}$ for prime q. If $q = 1$, the finite field may be referred to as a *prime* field; its elements are the integers, and the field operations are integer multiplication and addition modulo the field prime p. If $q > 1$, the field is called *extension field*. Extension fields exist for any q, however fields with

$q = 2$ are of particular interest due to their compatibility with boolean logic; these are called *binary fields*.

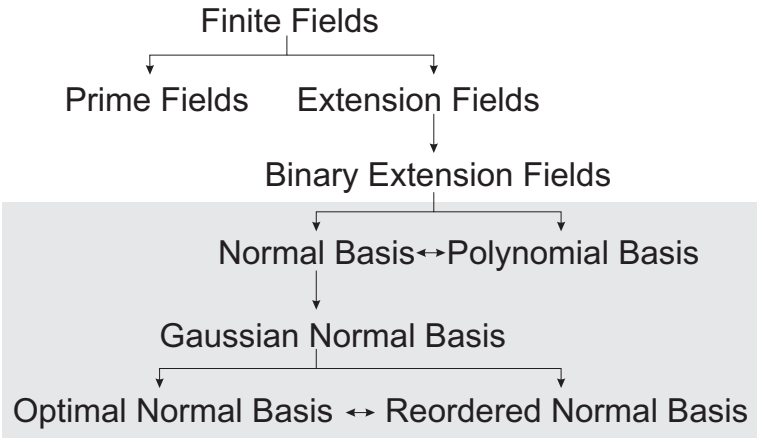


Figure 2.2: Taxonomy of finite fields and bases

Binary fields may be expressed using a number of different bases which are equivalent or *isomorphic*. The different bases discussed in this dissertation are shown in the shaded part of Figure 2.2, and the bi-directional arrows denote that the bases are isomorphic with respect to each other. Polynomial basis is often used for software implementations of binary fields as it requires fewer shifting operations and pre-computations compared to normal basis. Normal basis, which is isomorphic to polynomial basis, is considered inefficient, however a Gaussian normal basis can be found for most applications which significantly reduces the complexity of the multiplication operation. Type-I and type-II Gaussian normal bases are equivalent to an optimal normal basis, which has the least complex multiplication operation of any normal basis. Optimal normal basis only exist for a fraction of possible field sizes, however. Finally, reordered normal basis is a permutation of type-II optimal normal basis that allows for very regular hardware architectures.

2.5 Elliptic curves and elliptic curve group law

Elliptic curves, named as such due to their relationship with elliptic integrals, are defined as the locus of rational points $P = (x, y)$, with x, y elements of some field of characteristic $\neq 2, 3$ which satisfies $y^2 = x^3+ax+b$, $\Delta = -16(4a^3+27b^2) \not\equiv 0$, together with the point at infinity "O" [32]. In the case where the field is of characteristic 2 (i.e. the binary extension fields) an elliptic curve is the locus of points satisfying $y^2 = x^3 + ax^2 + bx + c$ together with the point at infinity O, for a, b, and c $\in \mathbb{F}_{2^m}$ [33].

The elliptic curve $y^2 = x^3+x+3$ over $\Re$ (an infinite field) is shown in Figure 2.3a. Elliptic curves may be defined over any field, including finite fields; the same curve ($y^2 = x^3 + x + 3$) over the finite field $\mathbb{F}_{11}$ is shown in Figure 2.3b.

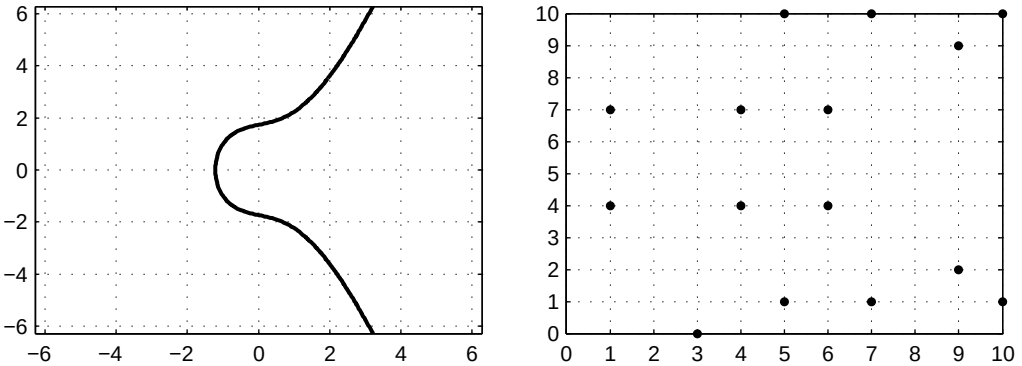


Figure 2.3: The elliptic curve $y^2 = x^3 + x + 3$ over (a) the infinite field $\Re$ and (b) over the finite field $\mathbb{F}_{11}$

Elliptic curve group law

It is possible to define a group law using elliptic curves. For rational points $P(x, y)$ and point at infinity O on the elliptic curve $\mathbf{E}$ over a field $\mathbb{F}_p$, the point O at infinity is the identity element, meaning $P + O = P$. Negatives are formed by changing the sign of the y-element: if $P = (x, y) \in \mathbf{E}$ then $-P = (x, -y)$ and also $P - P = O$.

For two points P and Q on the elliptic curve, $P \neq Q$, addition $P + Q$ is performed by tracing a line between P and Q, finding the point where the line intersects the elliptic curve, and then reflecting that point about the x-axis, as shown in Figure 2.4a. This may also be written as shown in Equation 2.15, where $P = (x_1, y_1), Q = (x_2, y_2), Q, P \in \mathbf{E}, P \neq Q$ and $P + Q = (x_3, y_3)$.

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2 \quad \text{and} \quad y_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1 \tag{2.15}$$

In the event $P = Q$, a point doubling equation is used to compute $2P$: a line tangent to P is drawn, and the point where the line intersects the elliptic curve is found, and reflected about the x-axis as shown in Figure 2.4b. This may also be written as shown in Equation 2.16, where $P = (x_1, y_1), P \in \mathbf{E}$, and $2P = (x_3, y_3)$.

$$x_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1 \quad \text{and} \quad y_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)(x_1 - x_3) - y_1 \tag{2.16}$$

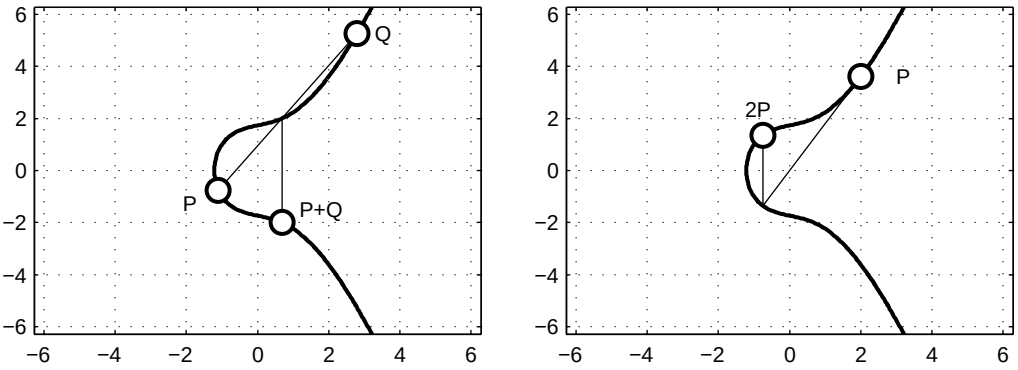


Figure 2.4: (a) Elliptic curve point addition and (b) doubling over the infinite field $\Re$

Note that while the above formulas are for prime fields, similar formulas also exist for elliptic curves over extension fields $\mathbb{F}_{q^m}$.

Elliptic curve group order

Another important aspect of elliptic curve groups over finite fields is their group *order*, that is, the number of rational points that exist on the elliptic curve. The order n can be estimated using Hasse's theorem which states that for a finite field $\mathbb{F}_q$:

$$q + 1 - 2\sqrt{q} \leq n \leq q + 1 + 2\sqrt{q} \tag{2.17}$$

2.6 Scalar Point Multiplication and the Elliptic Curve Discrete Logarithm Problem

With a rule for the addition and doubling of points on an elliptic curve, it is possible to define scalar point multiplication over an elliptic curve $\mathbf{E}$ of order n over a finite field $\mathbb{F}_{p^q}$ as the repeated addition of an elliptic curve point:

$$Q = kP, \text{ for } Q, P \in \mathbf{E} \text{ and } k \in [0, n-1] \tag{2.18}$$

$$= P + P + \ \ldots \ + P \text{ (k times)} \tag{2.19}$$

Given an elliptic curve $\mathbf{E}$ of order n, a base point P on the curve, and a scalar $k \in [0, n-1]$, it is possible to compute $Q = kP$ using an algorithm such as double-and-add. Computing k with knowledge of the curve, Q, and P, on the other hand is believed to be intractable and computationally unfeasible for curves with large order n; this is the elliptic curve discrete logarithm problem (ECDLP) upon which elliptic curve cryptography is based. It is worth noting that the ECDLP has not been proven to be an NP-hard problem [34]. The best known algorithm to solve or *attack* the ECDLP is the "Pollard Rho" method which requires approximately $\sqrt{\pi n/2}$ iterations, and for large enough n, this is wildly impractical.

2.7 Elliptic curve cryptography protocols

While elliptic curve cryptography protocols are beyond the scope of this dissertation, a brief overview of the elliptic curve Diffie-Hellman key exchange and the elliptic curve digital signature algorithm are provided to give a context for how scalar point multiplication is used, and the various standards that specify the various elliptic curve cryptography parameters. Both of these algorithms are based on the ECDLP described in the previous section.

2.7.1 Elliptic curve Diffie-Hellman exchange

The elliptic curve Diffie-Hellman key exchange (ECDH) is used when Alice (A) and Bob (B) need a shared secret, which is often used as a key for a much faster symmetric block encryption algorithm such as AES. The ECDH exchange is described as follows, and more details are provided in [35]:

Alice and Bob must first agree to a set of elliptic curve parameters: the field size m for binary fields (or the field prime p for fields $\mathbb{F}_{\text{l}}$), the specific elliptic curve $\mathbf{E}$ of order n, and a base point $G \in \mathbf{E}$. Alice and Bob also select two different random values d_A and d_B in the interval $[0, n-1]$. Only Alice knows d_A, and only Bob knows d_B; these values serve as Alice and Bob's *private* keys. Finally, Alice and Bob compute $Q_A = d_A G$ and $Q_B = d_B G$, respectively, where Q_A and Q_B serve as *public* keys.

In order to exchange a shared secret, Alice and Bob exchange Q_A and Q_B. Alice now computes $d_A Q_B$ and Bob computes $d_B Q_A$. Since $d_A Q_B = d_A(d_B G) = d_B(d_A G) = d_B Q_A$, Alice and Bob now have a shared secret. Also, because only Bob knows d_B and only Alice knows d_A, any third party intercepting Q_A and Q_B will not be able to reconstruct the shared secret without attacking the ECDLP. It should be noted that generating a public/private key pair requires multiplying a fixed point G on an elliptic curve by an unknown scalar, while computing the shared secret involves the

scalar multiplication of an unknown point.

2.7.2 Elliptic Curve Digital Signature Algorithm

Another major function public key cryptography performs is digital signature, which is used to assert the identity of the author of a message and prevents the message itself from being altered. Two separate algorithms are used in the elliptic curve digital signature algorithm (ECDSA), one for signing a message (algorithm 2.1), the other for verifying it (algorithm 2.2).

Algorithm 2.1: ECDSA Signature Generation [35]

input : Elliptic curve parameters a, b, base point P, curve order n, the field $\mathbb{F}_{q^m}$, private key d, message m
output: Signature (r,s)

1 Select $k \in [1, n-1]$
2 Compute $kP = (x_1, y_1)$
3 Compute $r = x_1 \bmod n$; if $r = 0$ go to step 1
4 Compute $e =$ hash(m)
5 Compute $s = k^{-1}(e + dr) \bmod n$; if $s = 0$ go to step 1
6 Return (r, s)

Algorithm 2.2: ECDSA Signature Verification [35]

input : Elliptic curve parameters a, b, base point P, curve order n, the field $\mathbb{F}_{q^m}$, public key Q, message m, signature (r,s)
output: Acceptance or rejection of the signature

1 Verify that r and s are integers in the interval $[1, n-1]$, if not then return "signature rejected"
2 Compute $e =$ hash(m)
3 Compute $w = s^{-1} \bmod n$
4 Compute $u_1 = ew \bmod n$ and $u_2 = rw \bmod n$
5 Compute $X = u_1P + u_2Q$
6 If $X = O$ (the point at infinity), return "signature rejected"
7 Compute $v = x_1 \bmod n$
8 If $v = r$ return "signature accepted", otherwise return "signature rejected"

As shown in algorithm 2.1, generating a digital signature with the ECDSA uses

unknown scalar point multiplication, while verifying it with algorithm 2.2 must compute the sum of two scalar point multiplications.

2.8 Elliptic curve cryptography standards

To date, a number of standards have been defined for elliptic curve cryptography which specify the underlying field size and elliptic curve parameters. Note that while these standards specify how key exchange and signature operations are to be carried out, implementation details are left up to hardware and software engineers.

Among the most important is the National Institute of Standards and Technology (NIST) Federal Information Processing Standards Publication 186-3 (FIPS 186-3) [35]. This standard defines the approved algorithms for digital signature, which includes DSA (which uses the RSA algorithm) and ECDSA. Additionally, FIPS 186-3 defines specific prime moduli, associated elliptic curves, and elliptic curve base points to be used for ECDSA. The five prime moduli defined by this standard are of special form that allow for simplified arithmetic on computer hardware with machine word sizes of 32 bits. The associated elliptic curves' "a" parameters are fixed at $-3 \bmod p$, as this allows for a slightly more efficient elliptic-curve point doubling operation, while the curves' "b" parameters are pseudo-randomly chosen.

FIPS 186-3 also defines five binary extension fields that are roughly equivalent in security to the prime fields. The field sizes are chosen such that a Gaussian Normal Basis exists, and the field polynomial (either a trinomial or pentanomial) is specified. For each of these binary extension fields, two curves are specified: one is chosen pseudo-randomly, while the other is a "Koblitz" or anomalous binary curve. Koblitz curves have some properties that allow for efficient implementations, however they are beyond the scope of this dissertation.

The American National Standards Institute (ANSI) defines a pair of standards,

X9.62 [30] and X9.63 [36], which detail the algorithms and data structures to be used in the financial services industry to carry out elliptic curve key exchange and digital signature, respectively. Note that these standards specify only the algorithms and data structures, and not which fields or elliptic curve parameters to use.

The United States' National Security Administration (NSA) "Suite B" specifies an entire suite of cryptographic protocols to secure communication up to a "Top Secret" security level [16]. Suite B requires the use of the 256 or 384 bit prime field (and associated elliptic curves and algorithms) specified NIST's FIPS 186-3 standard for digital signature. For key exchange, Suite B uses algorithms specified in the NIST Special Publication 800-56A with the fields and curves specified in FIPS 186-3.

The standards for efficient cryptography group (SECG) has their own standard [37], which is a superset of the FIPS 186-3 standard: in addition to the curves and fields defined in the NIST standard, it also defines at set of "prime" Koblitz curves which allow for some advantages conducting scalar point multiplication, however there is some concern that these curves might also have some still-unknown vulnerabilities [38].

The IEEE has its own standard titled "IEEE P1363", which specifies a number of acceptable algorithms for use in key exchange, digital signature, as well as block encryption. Per the standard, a number of elliptic curve cryptography algorithms may be used including ECDH, ECDSA, as well as some less common ones such as MQV and El-Gamel [39].

The Brainpool Standard Curves are a group of curves specified by a German group [40]. Unlike the NIST curves, they deliberately use randomly chosen field primes and field polynomials, and the elliptic curve parameters a and b are both randomly chosen as well. The primes and curve parameters are random to avoid any undisclosed or not-yet-discovered exploitation of the NIST primes, or by setting the elliptic curve parameter $a = -3$.

2.9 Summary

Security protocols based on elliptic curve cryptography rely on a hierarchy of operations. At the base of this hierarchy are finite field operations which can be implemented using different bases that have different advantages and disadvantages that depend on the implementation.

Rational points on an elliptic curve over finite fields form a group, where the group operation is defined using a chord-and-tangent rule.

Scalar elliptic curve point multiplication can be used to implement public key cryptography services based on the elliptic curve discrete logarithm problem, such as key exchange and digital signature. Elliptic curve cryptography requires much smaller key sizes compared to RSA, and some standards have replaced the use of RSA with elliptic curve cryptography.

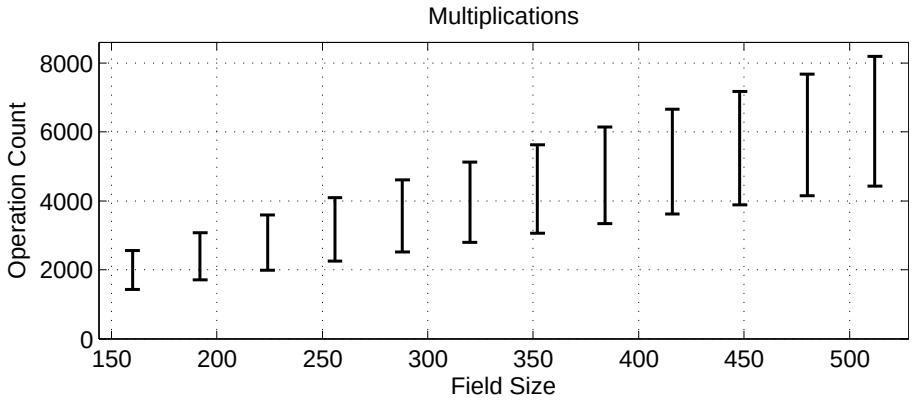
CHAPTER 3

A High-Speed Implementation of a SIPO Multiplier Using RNB

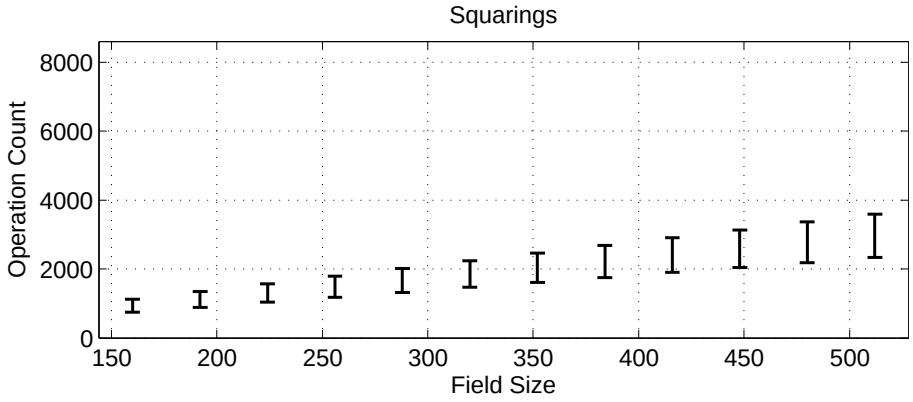
3.1 Introduction

As described in Chapter 1, section 1.3, there are a number of approaches suitable for improving the performance of elliptic curve scalar point multiplication. In turn, scalar point multiplication is dependent on the performance of its underlying field operations. Figure 3.1 presents estimated upper and lower bounds of the different finite field operations required for a single scalar point multiplication over varying field sizes. It is quickly revealed that finite field multiplication has the highest count, while it was seen in Chapter 2 that it is also more computationally expensive than addition or squaring. Improving finite field multiplication will directly lead to improved scalar point multiplication performance.

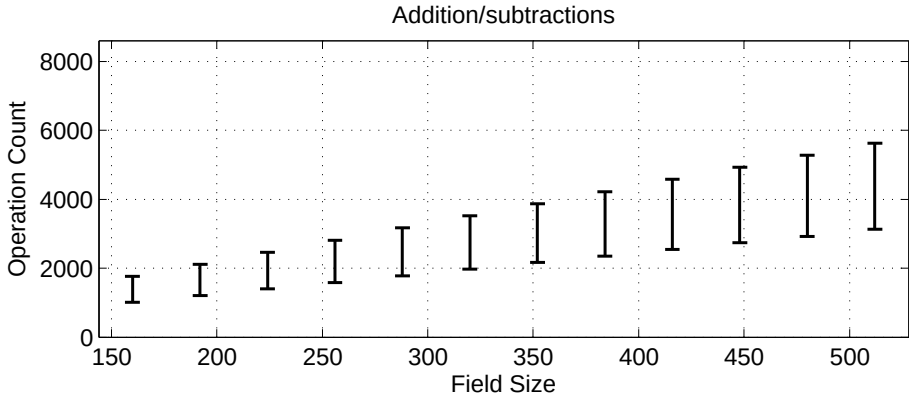
This chapter incorporates the outcome of a joint research undertaken in collaboration with Ashkan Hosseinzadeh Namin under the supervision of Dr. M. Ahmadi, Dr. H. Wu, and Dr. R. Muscedere. In all cases, schematics, layouts, data analysis and interpretation, were performed by the author.



(a) Number of finite field multiplications per scalar point multiplication



(b) Number of finite field squarings per scalar point multiplication



(c) Number of finite field additions & subtractions per scalar point multiplication

Figure 3.1: Estimated range of finite field operation counts for scalar point multiplication over different field sizes

Proposed in this chapter is a a new VLSI implementation for a Serial-In Parallel-Out finite field multiplier using a reordered normal basis. The proposed design could

be integrated into a CPU or SOC, enabling special finite field multiplication instructions to accelerate the performance of elliptic curve scalar point multiplication.

The multiplier was implemented in a .18 μm TSMC CMOS technology using multiples of a custom-designed domino logic block. The domino logic design was simulated, and functioned correctly up to a clock rate of 1.587 GHz, yielding a 99% speed improvement over the static CMOS' simulation results, while the area was reduced by 49%. This multiplier's size of 233 bits is currently recommended by NIST for ECDSA and ECDH algorithms, and is of practical size for use for binary field multiplication in elliptic curve cryptosystems.

The organization of this chapter is as follows: Section 3.2 presents a survey of existing architecture for type-II ONB multiplication. In section 3.3, the design and implementation of the xax-module which is the main building block of the multiplier, is discussed. Section 3.4 presents the implementation of a 233-bit multiplier using the xax-module as the main building block. In section 3.5, comparisons between different VLSI implementations are presented. A few concluding remarks are given in section 3.6.

3.2 A review of existing architectures for ONB type-II multiplication

As described in Chapter 2, section 2.4.5, the type II ONB and reordered normal basis representations of an element are simple permutations of each other. Therefore, RNB multipliers can be used as type-II ONB multipliers and vice versa. Many different architectures have been proposed for multiplication in normal basis, and most architectures fall into one of three main categories: bit-serial, bit-parallel, and word-serial. *Bit-serial* multipliers require m clock cycles to compute an m-bit wide multiplication. They generally boast excellent area utilization, however they require the highest num-

ber of clock cycles compared to other architectures. *Bit-parallel* multipliers are exactly the opposite: hardware is parallelized, significantly increasing area costs, while the computation itself takes a single clock cycle. Finally, *word-serial* multipliers are a compromise between these two extremes. Bits are processed w-bit *words* at a time, requiring $\lceil m/w \rceil$ clock cycles. The complexity of different multipliers available in open literature are listed in Table 3.1.

Table 3.1: Complexities comparison between type-II ONB / RNB Multipliers

Multiplier	# AND	# XOR	# flip -flops	# Clock Cycles	Critical Path Delay	Basis
MO [41]	$2m-1$	$2m-2$	$2m$	m	$T_A + (\lceil \log_2(2m-1) \rceil) T_X$	ONB
IMO [42]	m	$2m-2$	$2m$	m	$T_A + (1 + \lceil \log_2 m \rceil) T_X$	ONB
GG [43]	m	$\frac{3m-1}{2}$	$3m$	m	$T_A + 3T_X$	ONB
Feng [44]	$2m-1$	$3m-2$	$3m-2$	m	$T_A + 4T_X$	ONB
Agnew [45]	m	$2m-1$	$3m$	m	$T_A + 2T_X$	ONB
XEDS [46]	$2m-1$	$2m-2$	$2m$	m	$T_A + (\lceil \log_2(2m-1) \rceil) T_X$	ONB
AEDS [46]	m	$3m-3$	$2m$	m	$T_A + (\lceil \log_2(2m-1) \rceil) T_X$	ONB
w-SMPOI [47]	$\lfloor \frac{m}{2} \rfloor + 1$	$3m$	$3m$	m	$T_A + 3T_X$	ONB
w-SMPOII [47]	m	$m + \lfloor \frac{m}{2} \rfloor$	$3m$	m	$T_A + 3T_X$	ONB
Kwon [48]	m	$\frac{3m+1}{2}$	$3m$	m	$T_A + 2T_X$	GNB
Dong [49]	m	$\frac{3m+1}{2}$	$3m$	m	$T_A + 2T_X$	ONB
PISO [4]	m	$2m-1$	$2m+1$	m	$T_A + (1 + \log_2 m) T_X$	RNB
SIPO [4]	m	$2m$	$3m+1$	m	$T_A + 2T_X$	RNB

In Table 3.1, the first row is the famous Massey-Omura normal basis multiplier, and the second row is the improved version proposed by Gao and Sobelman. The third row shows the architecture proposed by Geisellman and Gollman, which exploits the symmetry property of the normal basis. The fourth and the fifth rows list the architectures proposed by Feng and Agnew respectively. The next two rows represent Reyhani-Masoleh's architecture; the XOR-Efficient Digit Serial as well as the AND-Efficient Digit Serial multipliers, while the next two rows show the Sequential Multipliers with Parallel Output type I and II, also proposed by Reyhani-Masoleh. The next two rows feature designs with similar gate counts and critical path delay

by Kown et al. and Dong et al. Note that Kwon's work works with Gaussian normal basis, and the values listed in the table are for a type 2 GNB, which is equivalent to a type-II ONB. The last two rows list the Serial-In Parallel-Out and Parallel-In Serial-Out reordered normal basis multipliers proposed by Wu. As can be seen from the table, the Agnew architecture (fifth row), As well as the Kwon and Dong architectures (tenth and eleventh rows) and Serial-In Parallel-Out architecture (last row) have the smallest critical path delay compared to other architectures.

The critical path delay of the Wu architecture is the minimum among all except three: The Agnew architecture, which has similar complexity as Wu's, and the work by Dong as well as Kwon's architecture, both of which actually require fewer XOR gates, although these architectures are not for reordered normal basis, and have irregular interconnect patterns that are unsuitable for a domino logic implementation. In this work, the focus is on the SIPO architecture proposed by Wu in [4], which boasts a low critical path delay, while enabling a very regular VLSI implementation due to its use of RNB.

3.3 Design of a practical size multiplier using the xax-module

3.3.1 Multiplier size selection

According to [35], to provide a sufficient level of security, the field size is required to be at least 163 bits for an elliptic curve cryptosystem. As mentioned in Chapter 2, section 2.8, some fields have been recommended by different standards for use, while others were banned. In this work, a field size of 233 was selected for three reasons. First, it is in the range suitable for elliptic curve cryptography. Second, there exists a type-II ONB representation [39], meaning that the RNB also exists. Finally, the

field size is recommended by the NIST as their Digital Signature Standard (DSS) in the ECDSA algorithm [35]. A few other field sizes were also recommended by the standard, but the field size of 233 is the only one such that there exist a type II ONB.

3.3.2 Selected multiplier architecture

The SIPO multiplier proposed in [4] is shown in Figure 3.2. The architecture has been redrawn in Figure 3.3 to show its regularity, and to emphasize the fact that it can be implemented as a serial connection of a single module which is shown in the Figure inside the box. This module, which is made out of three flip-flops, two XOR gates and an AND gate is referred to as an xax-module. The two XOR gates, in addition to the AND gate, create the critical path of the multiplier. One way to reduce the critical path delay is to implement the XOR-AND-XOR function using domino logic [50]. However, careful consideration should be taken into account when designing such circuity [51]. The 233-bit multiplier can be made by replicating the xax-module once for every bit of the multiplier, and serially connecting them together.

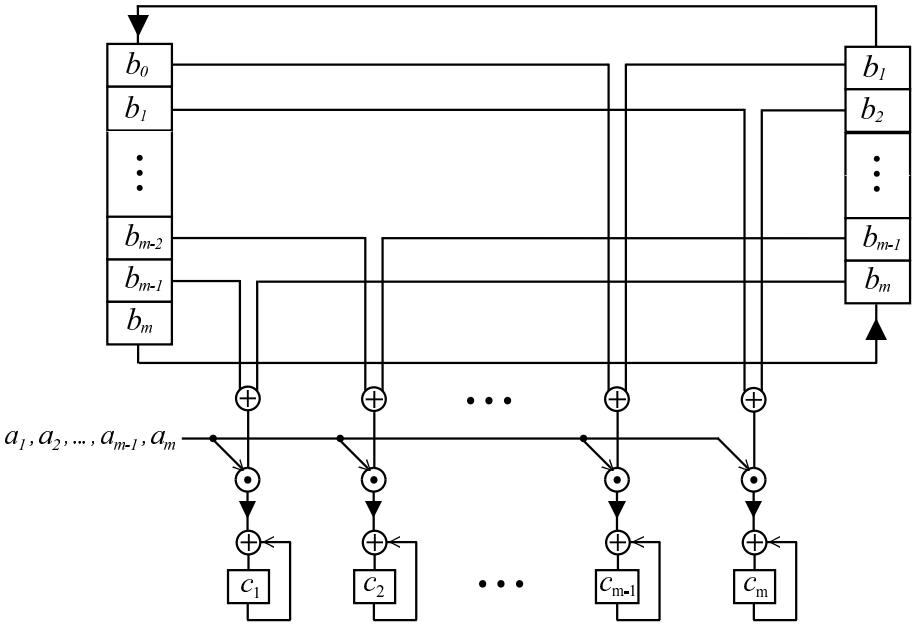


Figure 3.2: Serial-In Parallel-Out RNB multiplier [4]

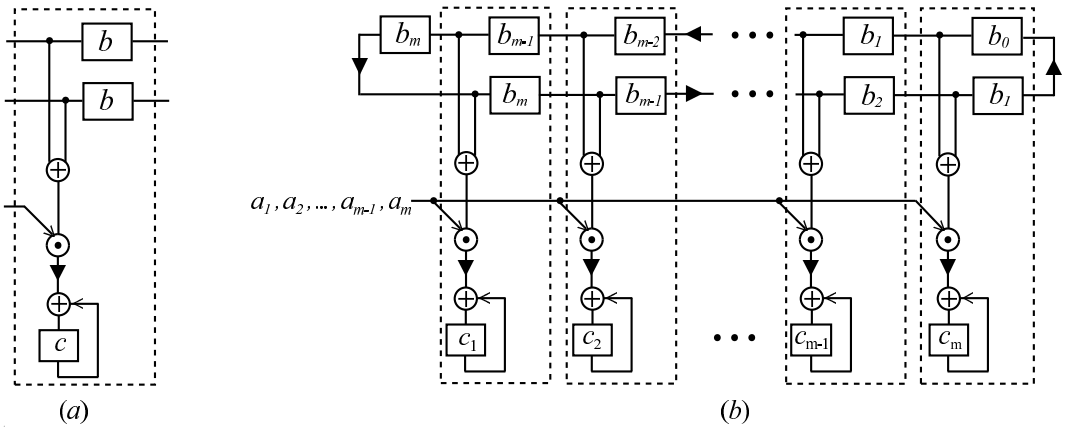


Figure 3.3: xax-module and the SIPO Multiplier Composed of xax-module

3.3.3 Design and implementation of the xax-module

The schematic shown in Figure 3.4 implements the XOR-AND-XOR function in domino logic; it is responsible for implementing the function $((b1 \oplus b2) \cdot a) \oplus c)$. The input signals are referred to in the schematic as "input_b1", for the b1 input, while "input_b1_comp" refers to the logical complement of b1. This convention is used for the rest of the signals present in the schematic. Additionally, the output of the xax-module is labelled "XAX_out".

The number of transistors in the implementation has been reduced while maintaining the same functionality by using the schematic shown in Figure 3.5 for the proposed implementation. In this Figure, the design is quite simple, consisting of 25 transistors (as opposed to 29 in the original design).

In Fig. 3.5 transistor P1 acts as the pull-up network, charging the node Q during the pre-charge state. Transistors N2-13 form the pull down network responsible for discharging node Q when the appropriate combinations of inputs exist. N2-13 connect to the evaluate transistor, N1, which opens a path to ground during the evaluate phase. Transistor P2 is the keeper, reducing the charge leakage effect at node Q.

Transistors P0 and N0 create the output NOT stage, providing the output current drawn from the module. Four NOT gates also exist in the module, which generate

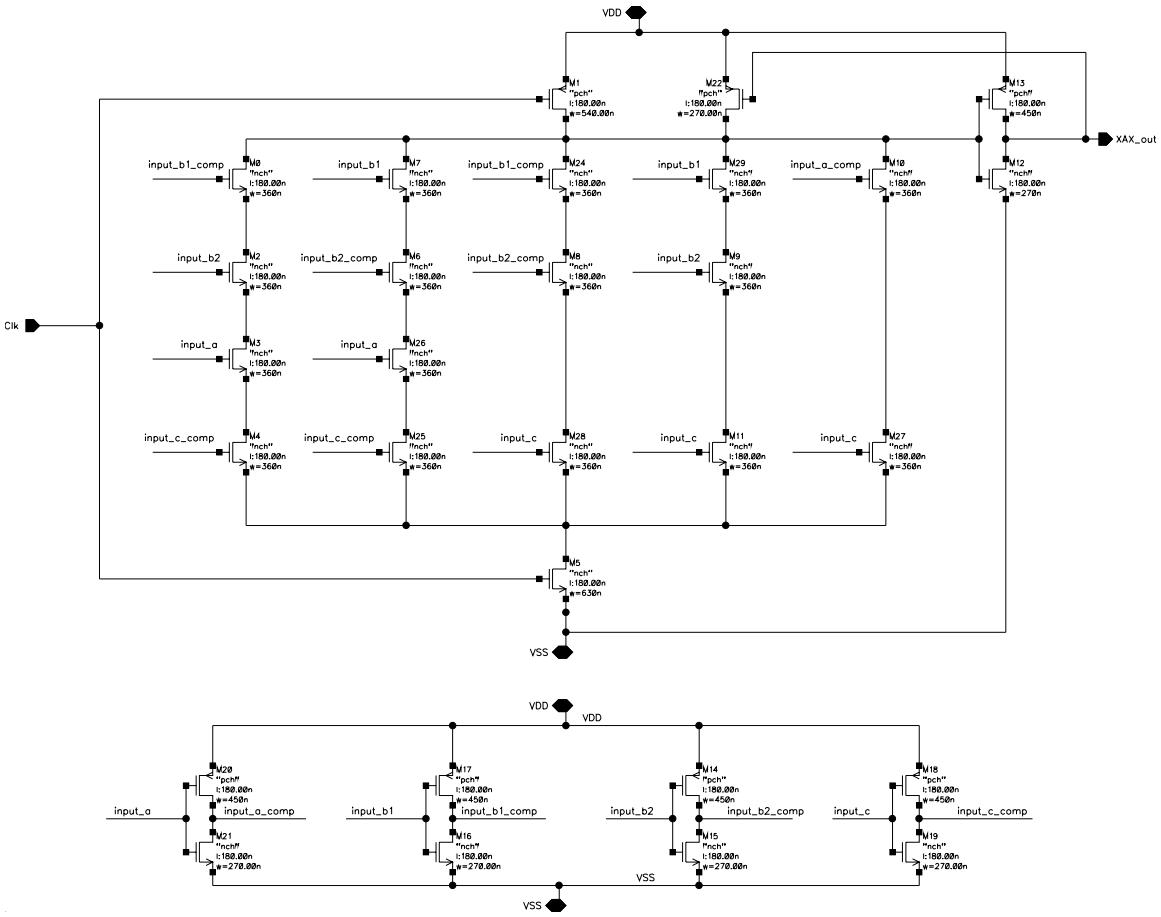


Figure 3.4: XOR-AND-XOR Function Implementation in Domino Logic

the complements of the xax-module's inputs.

To verify the improvement in performance and area utilization both layouts have been implemented, as shown in Figure 3.6. The layout shown on the right side of the Figure has an area of 198.64 μm^2 which relates to the design with 25 transistors, while the other layout implements the 29 transistor design, and has an area of 218.64 μm^2. In these layouts, the three large horizontal wires, from top to bottom, are VDD, VSS, and VDD rails. The section along the top are the four NOT gates used to create the complements of the inputs, while the section along the bottom implements the XOR-AND-XOR function. The layout on the right implements the schematic shown in Figure 3.5, while the layout on the left implements the schematic shown in Figure 3.4. The gap between the VDD and VSS rails was selected to simplify the connection

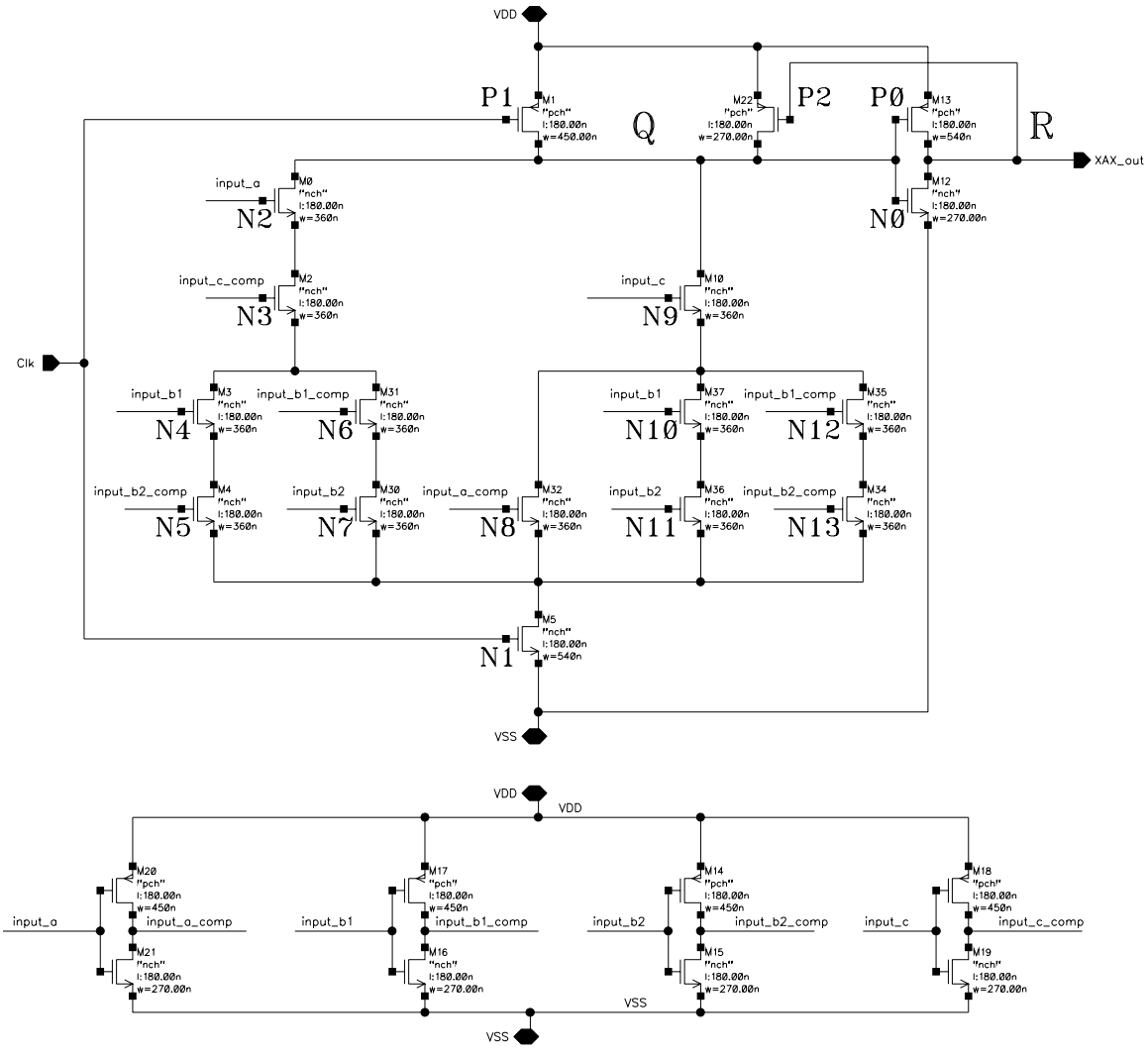


Figure 3.5: A New XOR-AND-XOR Function Implementation in Domino Logic

of the module to three D flip-flops from the standard cell library. The height for the layout was set to be three times the height of a standard-cell: 19.962 μm, since three D flip-flops were to be connected to each domino cell to create the xax-module as it is shown in Figure 3.7.

From the performance point of view, the critical path delay was reduced from 799 ps to 562 ps by using the design with fewer transistors. This difference in critical path delay is largely attributed to the additional capacitance at node "Q" due to the three additional transistors connected to that node.

The flip-flop used in the design of the xax-module was selected from the Virtual

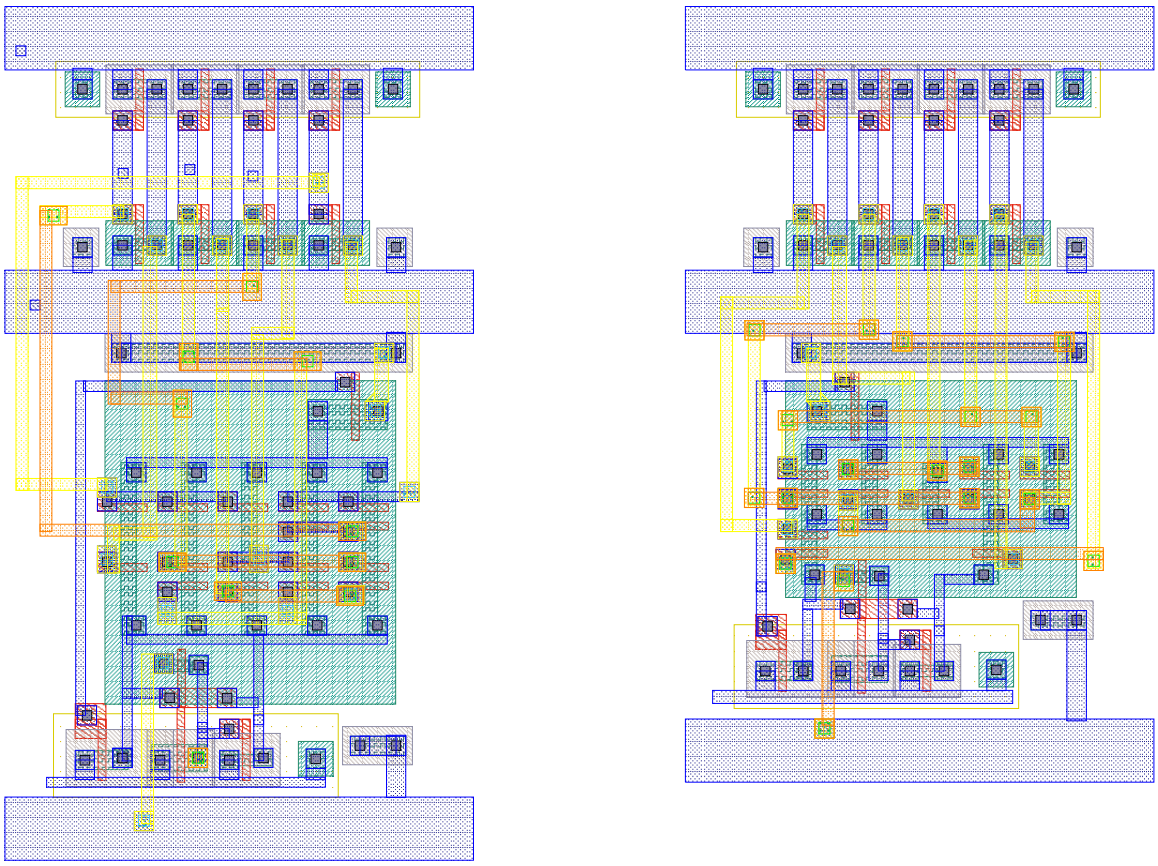


Figure 3.6: Layouts for the XOR-AND-XOR function in domino logic

Silicon CMOS library [52]. Care was taken to select an appropriate flip-flop to interface with the proposed domino-logic cell. A negative-edge triggered flip-flop was needed to maximize the time available for the domino cell to evaluate. Furthermore, it was required to have a hold-time less than or equal to zero, as the domino cell's output becomes invalid immediately after the falling edge of the clock.

The final layout for the xax-module including the XOR-AND-XOR function implementation and three D flip-flops, is shown in Figure 3.7. Note that the D flip-flops shown in this Figure are reproduced as black-box cells since they are the intellectual property of Virtual Silicon. The total area for the xax-module, including the area of three D flip-flops, was measured to be $449.18\,\mu m^2$. The area of the domino-cell on its own is $198.64\,\mu m^2$.

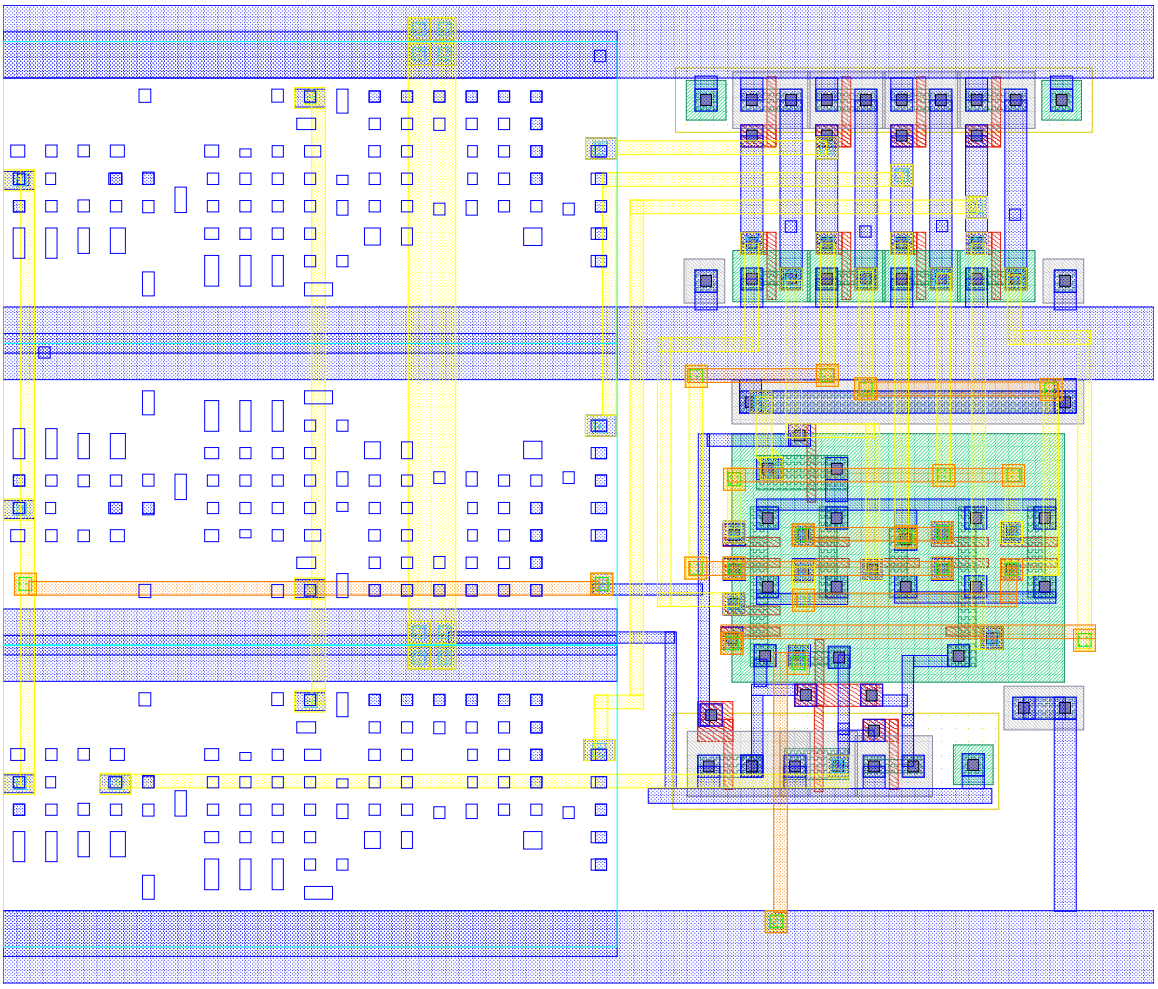


Figure 3.7: Layout for the xax-module

3.3.4 Performance Analysis of the xax-module

In order to assess the performance of the xax-module over a range of process, temperature, and voltage conditions, a set of circuit simulations were carried out. Initially, corner analysis was performed to determine the worst-case delay of the xax-module. Simulation results are tabulated in Table 3.2. As shown, the worst-case delay, which occurs at the "slow-slow" corner at $85°C$ is $933ps$. The best case delay occurs at the "fast-fast" corner at $-30°C$; $370ps$.

Additional simulations were carried out to determine the xax-module's performance as the supply voltage varies. The nominal supply voltage of $1.8V$ was varied by up to $\pm 10\%$, and the xax-module delay was measured. Simulation results are

Table 3.2: Corner analysis simulation results of the xax-module delay

Corner | Temp.	$-30°C$	$0°C$	$25°C$	$55°C$	$85°C$
FF	$370ps$	$415ps$	$460ps$	$515ps$	$579ps$
FS	$404ps$	$450ps$	$495ps$	$555ps$	$623ps$
TT	$454ps$	$510ps$	$562ps$	$640ps$	$721ps$
SF	$543ps$	$620ps$	$690ps$	$790ps$	$910ps$
SS	$579ps$	$655ps$	$725ps$	$819ps$	$933ps$

shown in Table 3.3. The data in the table indicates that the delay varies by no more than 10% for different supply voltage variations.

Table 3.3: xax-module delay for different variations in supply voltage

Supply Voltage	$1.62V$ (-10%)	$1.71V$ (-5%)	$1.80V$ (0%)	$1.89V$ $(+5\%)$	$1.98V$ $(+10\%)$
xax-module delay	$620ps$	$588ps$	$562ps$	$542ps$	$526ps$

Charge sharing is a common issue in domino logic circuits. The charge sharing at node Q was addressed by using the keeper transistor P2. The charge sharing problem also exists at the source of transistor N3, and at the source of transistor N9. One way to address this would be to add additional keeper transistors to these nodes. This, however, greatly complicates the layout, leading to a larger area utilization, more routing, and lower performance. The set of circuit simulations shown in this section demonstrates that acceptable performance is still achievable over a wide range of conditions.

3.3.5 Design and implementation of the 233-bit multiplier using the xax-module

The block diagram design of the 233-bit multiplier is shown in Figure 3.8. As shown, the main part of multiplier architecture can be implemented by connecting the xax-modules serially. 13 xax-modules were used in each row, for each of 18 rows to create the complete multiplier. This row/column distribution was chosen to give the design

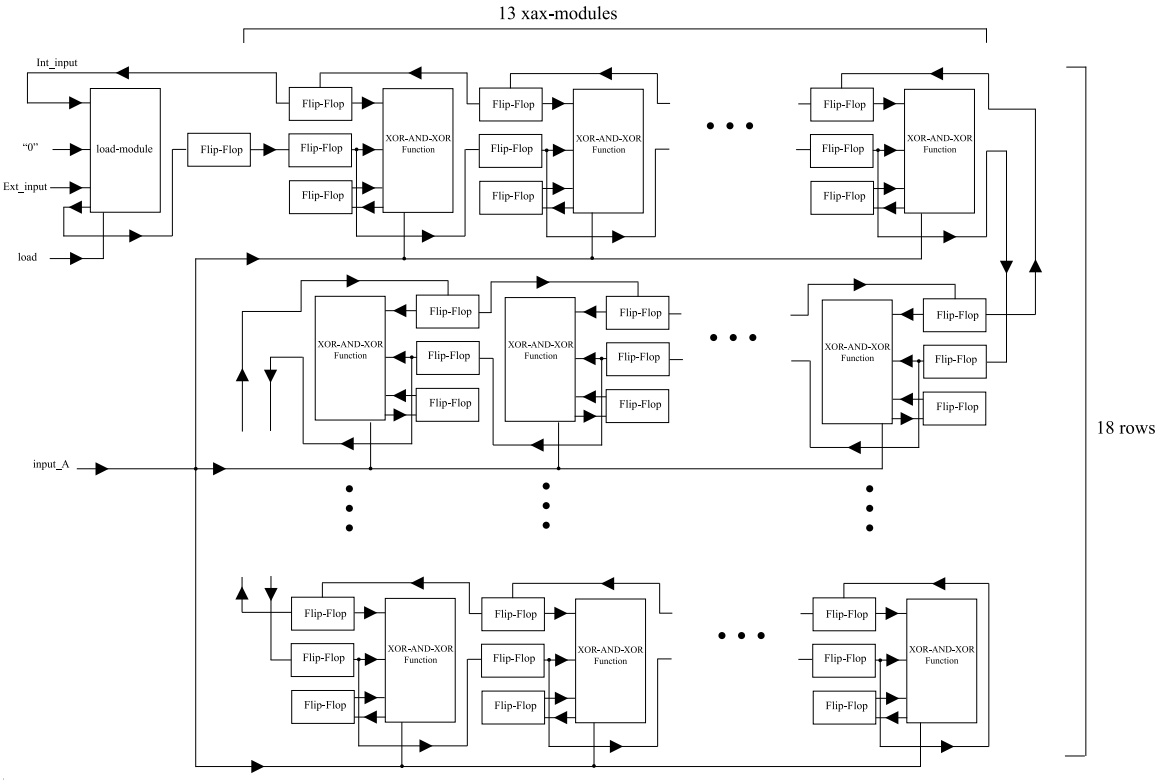


Figure 3.8: Block diagram of the 233-bit Multiplier

an even aspect ratio, which is typically desirable when floorplanning.

An extra block, referred to as the load-module, was added in Figure 3.8. It is used to load the coefficients of input b serially into the multiplier when the load signal is enabled. In order to achieve this, the a input of the load-module is connected to the load enable signal, and the c input to the external input, Ext_int. Input $b1$ of the load-module was connected to the output of the previous stage and, input $b2$ was shorted to ground. Since the xax-module implements the function $((b1 \oplus b2) \cdot a) \oplus c)$, the output of the load-module would be $((Int\_input \cdot load) \oplus Ext\_input)$. This can then be used to load the coefficients of operand B into the circular shift register. Table 3.4 tabulates the two combinations that can be used to load the data into the multiplier. When the load signal is "0", the output of the xax-module would be equal to the Ext_input, and when load is "1" and Ext_input is "1", the shift register acts as the circular shift register.

Table 3.4: load-module Input/Output Characteristics

Load	Ext_input	Int_input	Output
0	Ext_input	X	Ext_input
1	0	Int_input	Int_input

A tree-structure of similarly-sized buffers was used to generate the clock tree for the multiplier's clock signal. The same was done for the input a, as it is also a high fan out net that connects to every xax-module in the design. The full layout of the multiplier is shown in Figure 3.9. The large, empty-looking strip on the left consists of block-box library cells used to buffer the clock signal. The top-left block is the load module, while the rest of the design consists of 233 instantiations of the xax-module, one of which is highlighted in the Figure. The layout exactly follows the form shown in Figure 3.8.

The final layout of the multiplier including all parasitic capacitances was simulated in Cadence Analog Environment using Spectre. The circuit performed correctly up to a clock rate of $1.587\,GHz$. Simulation voltage waveforms for the clock frequency of $1.54\,GHz$ are shown in Fig. 3.10.

In this Figure, the first row is the buffered input a. Rows two, three, and four show the signals $b1$, $b2$, and c as they exit the xax-module's flip-flops and enter the XOR-AND-XOR function's inputs. Row five is the output of the xax-module (node R), while row six is the voltage at node Q in Fig. 3.5. Finally, row seven shows the multiplier's clock waveform as it exits the buffer-tree and enters the xax-modules. All 16 possible input combinations of a, $b1$, $b2$, and c, were tested and verified to give the correct output when determining the maximum operating frequency.

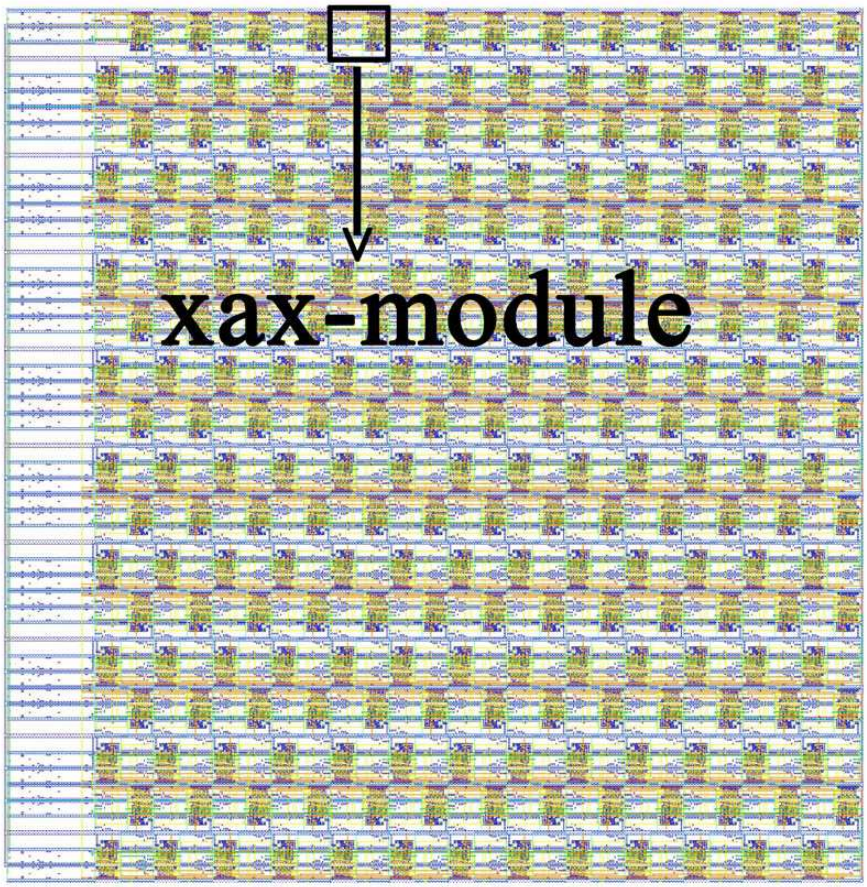


Figure 3.9: 233-bit Proposed Multiplier Layout

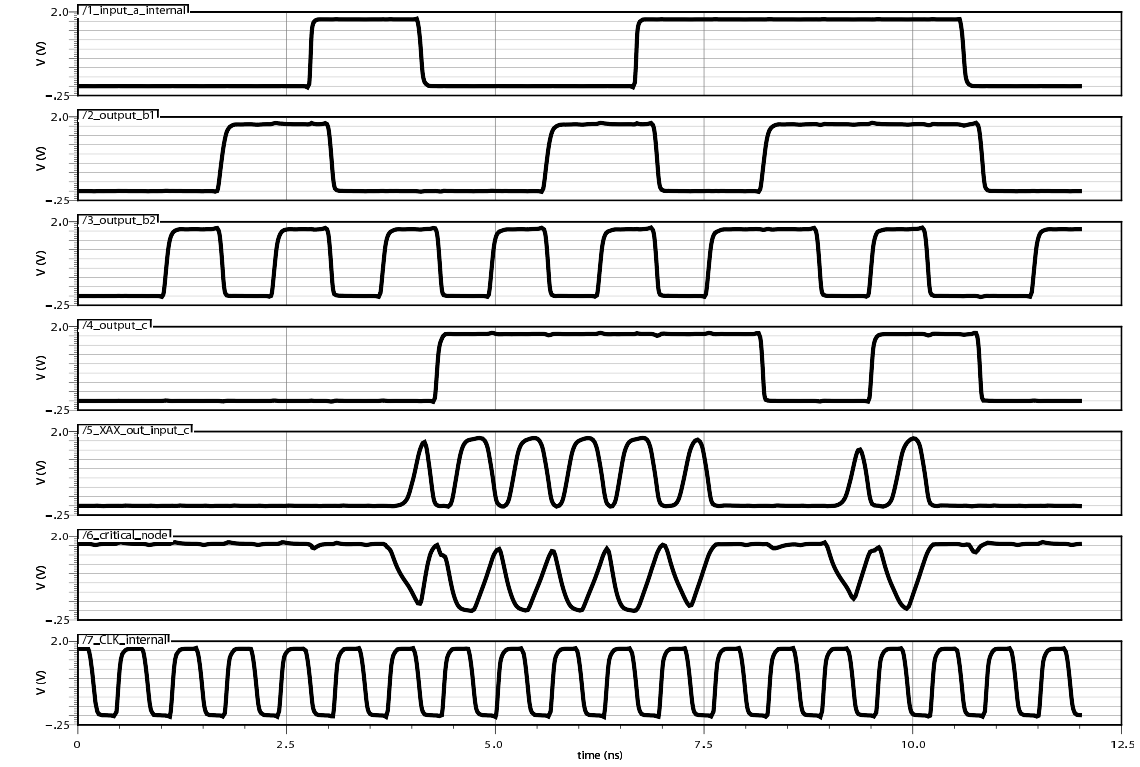


Figure 3.10: Post Place-and-Route Simulation Result of the Proposed 233-bit Multiplier, from top to bottom: input a, input $b1$, input $b2$, input c, node R, node Q, clock

3.4 Design of the 233-bit Multiplier Using Static CMOS

The static CMOS multiplier implements the same functionality as the domino logic design. Similar to the domino-logic design, the static CMOS version also incorporates a load-module (implemented in static CMOS) to serially load an external input into the multiplier.

The static CMOS design process by writing the parametrized C code to generate the VHDL code describing the multiplier in hardware. Using this method, different size multipliers are easily generated by changing parameters in the C code. The VHDL code was simulated using Cadence NCSim to confirm that the architectural code was functioning correctly. Afterwards, the VHDL code was synthesized to a gate-level netlist using Synopsys Design Compiler. Compilation parameters were always chosen to maximize the operating frequency of proposed design; the critical path delay at this stage was $1.11\,ns$.

Next, the generated gate-level netlist was simulated again using Cadence NCSim to confirm that the functionality did not change during the synthesis stage. Then the verified gate-level netlist was used for partitioning, placement, and routing using Cadence Encounter; the clock tree was also generated using Encounter's Clock Synthesizer. The worst negative slack calculated by Encounter after the place-and-route steps was reported to be $0.145\,ns$, bringing the total critical path delay to $1.255\,ns$.

The post place-and-route area of the multiplier was $216737.136\,\mu m^2$ which could be clocked up to $796\,MHz$. The total number of standard cells used was 1965, while achieving a maximum gate density of 80%. The final layout for the static CMOS multiplier is shown in Figure 3.11.

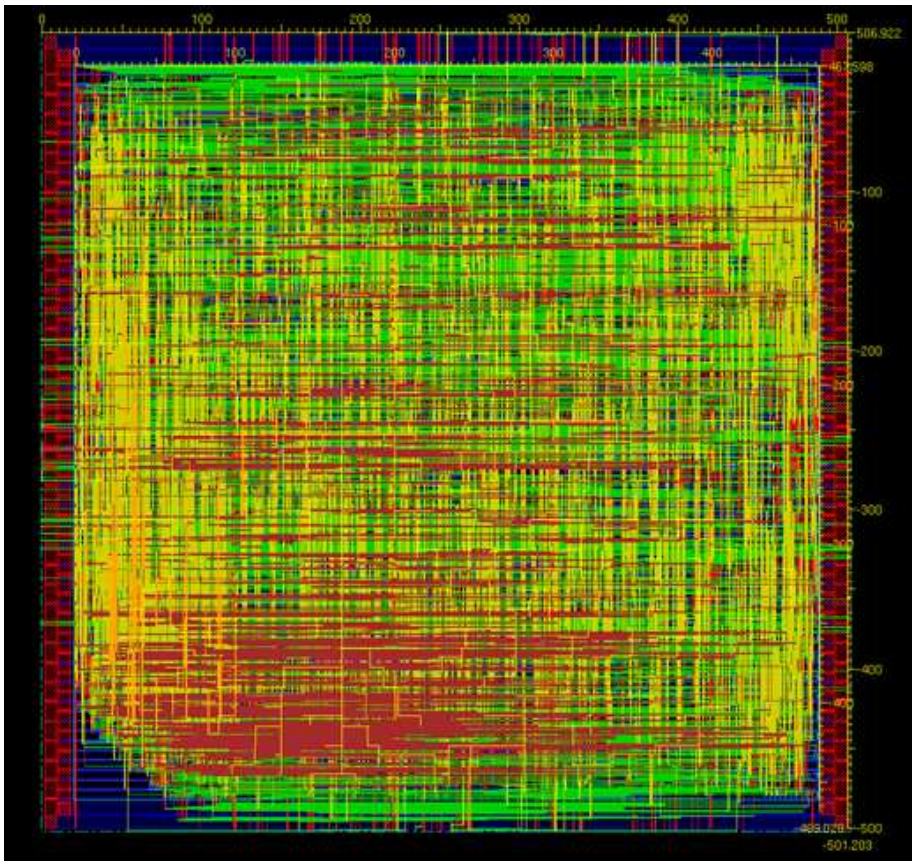


Figure 3.11: Static CMOS 233-bit SIPO RNB multiplier layout

3.5 A Comparison of Different VLSI Implementations

Comparison between different VLSI implementations are shown in Table 3.5. In this table clock frequency represents the clock speed of the multiplier, while Multiplication Delay is the total amount of time that takes the multiplier to finish one multiplication operation. Note that most of architectures in this table are hybrid and have different parallelism level, which means that a number of multipliers are running in parallel to reduce the multiplication delay. This, however, increases the area and power consumption used by the multiplier. Also used as a measure of performance is the clock speed of the multiplier. The first row represents the famous Massey-Omura multiplier implemented by Agnew et al. The second row presents the polynomial basis multi-

plier by Okada, which was designed as a general $GF(2^m)$ multiplier lacking particular optimizations for specific field sizes. The third row of the table represents a multiplier by Satoh et al. This multiplier could perform $GF(2^m)$ and $GF(p)$ multiplications for variable input sizes as long as enough memory exist. Next one is the polynomial basis multiplier proposed by Tang, the multiplier would take 36 clock cycles to finish one complete multiplication operation since it was made of 8 parallel bit-level multipliers connected together. The area requirement of the multiplier in CMOS 0.18 μm was equal to 189297 μm^2. The next row presents the polynomial basis multiplier proposed by Ansari; this architecture would take five clock cycles to finish the multiplication but since it is a full parallel multiplier the area requirement is significant (roughly $4\mu m^2$). It also uses a smaller filed size (163 instead of 233 for the proposed).

The next row of the table presents the static CMOS implementation from section 3.4, this design would take 233 clock cycles to finish one multiplication operation while the area utilization was 216737 μm^2. The maximum clock frequency of this design was 796 Mhz. The last row represents the proposed design using the xax-module, this design could be clocked at 1.587 Ghz , while it would take 233 clock cycles to finish one multiplication. Note that the proposed design is not made of any parallel smaller multipliers, so the area requirements is small and equal to 109644 μm^2. As shown, the clock rate increase is 99%, while the area reduction is 49% for the proposed design compared to static CMOS.

Table 3.5: Comparison between different VLSI implementations for finite field multipliers

Architecture	Field Size	Clock Freq. (MHz)	Multiplication Delay (ns)	Area (μm^2)	Area × Delay $\times 10^6$	Technology	Base
Agnew [45]	155	40	3900			$2\,\mu m$	ONB II
Okada [53]	163	66	545			$0.25\,\mu m$	Poly.
Satoh [54]	160	510.2	256.75			$0.13\,\mu m$	Poly.
Tang [55]	233	50	600	189,297	113.58	$0.18\,\mu m$	ONB II
Ansari [56]	163	125	40	1,272,102	50.88	$0.18\,\mu m$	Poly.
Static CMOS	233	796	293	216,737	63.50	$0.18\,\mu m$	RNB
Proposed	233	1,587	147	109,644	16.12	$0.18\,\mu m$	RNB

If area times delay is defined as a performance metric, it can be concluded that

the proposed design is almost four times more efficient than the static CMOS design.

3.6 Summary

In this chapter, a new VLSI implementation for a 233-bit finite field multiplier was presented. The proposed design employs a main building block designed in domino logic. The speed improvement was measured to be 99% in comparison to static CMOS implementation, while area reduction was 49%. The final design was successfully simulated up to a clock rate of $1.587\,GHz$. The proposed design's field size is currently recommended by the NIST standard in their Elliptic Curve Digital Signature Standard, rendering it a desirable building block in elliptic curve cryptosystem designs. With its small area utilization and high operating speed, the proposed multiplier could be integrated into a CPU to accelerate the performance of scalar point multiplication. The results of this work were published in [1].

CHAPTER 4

A Full-Custom, Improved SIPO Multiplier Using RNB

4.1 Introduction

The work presented in this chapter is focused on improving the serial-in parallel-out architecture proposed in [4], and implemented in the previous chapter. As shown in the Chapter 4, this multiplier presents the smallest critical path delay compared to similar designs, and it presents a highly regular architecture that is very well suited to a full-custom VLSI implementation. Furthermore, as shown in the previous chapter, the regularity of this architecture can be used to create a high-speed multiplier by designing optimized, custom-layout building blocks. In this chapter, a more efficient implementation is created by using different building blocks, and by making use of custom-designed flip-flops. The new implementation can perform multiplication 12% faster than the previous design, while it reduces the area utilization by 43%.

The rest of this chapter is organized as follows. In section 4.2, the design and implementation of the multiplier's main building block, the XA-module, is presented.

Implementation details of a 233-bit multiplier using XA-modules is given in section 4.3. Simulation results are presented in section 4.4, while a comparison between similar implementations is discussed in section 4.5. Finally, section 4.6 contains some concluding remarks.

4.2 Design and implementation of the XA-module

A high-speed, and low-area 233-bit multiplier design was presented in Chapter 3. The architecture is made of multiple copies of a building block called the 'xax-module', which, in turn, is made of three D flip-flops, an AND gate, and two XOR gates as shown in Figure 4.1. Different size multipliers can be realized by serially connecting xax-modules together.

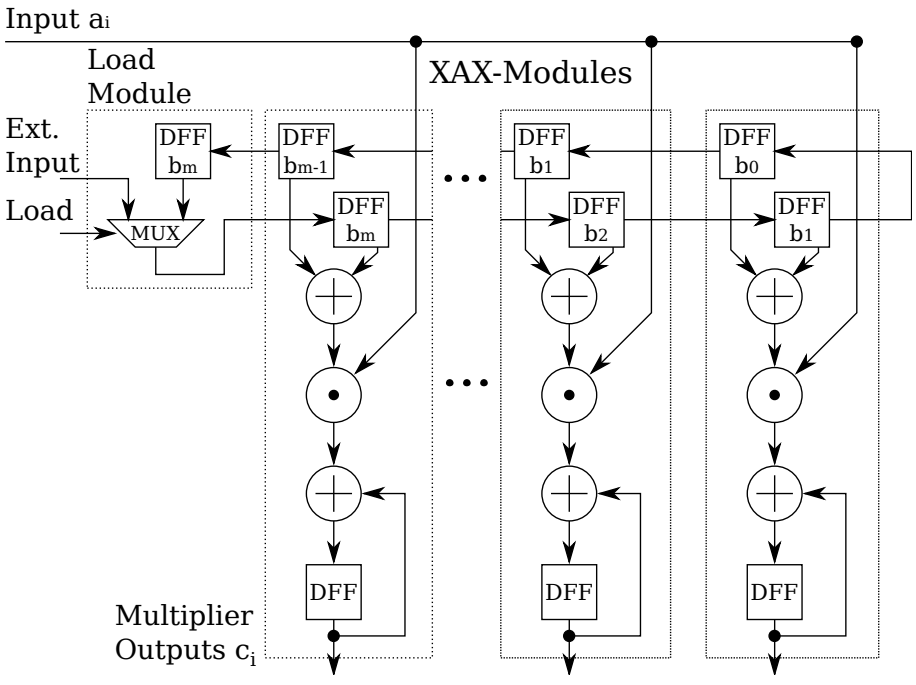
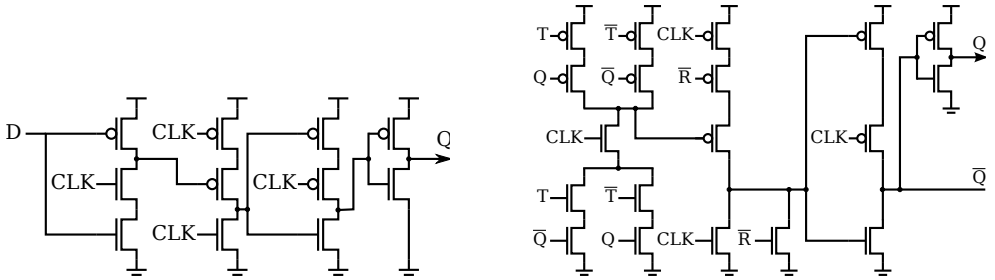


Figure 4.1: Multiplier Block Diagram

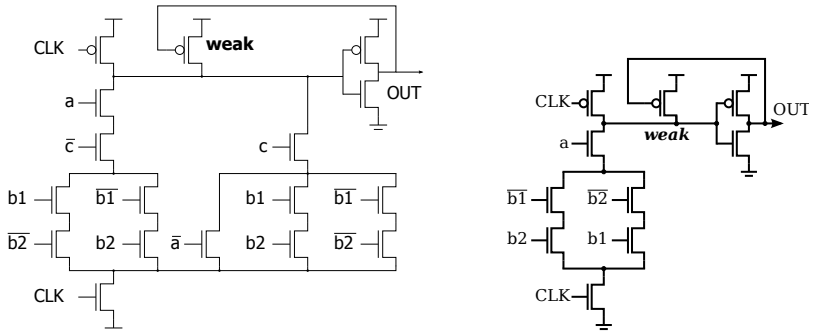
In this work, we present a more efficient design using a different implementation for the building block module. The main improvements are as follows. First, we have used custom designed flip-flops instead of standard library cell flip-flops. We have

used the TSPC flip-flops [5], whose architecture is shown in Figure 4.2a. Second, we have replaced one XOR gate in addition to a D flip-flop with a single T flip-flop. As shown in figures 4.3a and 4.3b, this results in a much simpler combinational circuit that will be shown to perform faster than the domino logic implementation presented in the previous chapter. Note that this idea has been recently proposed in [57]. The block diagram of the new building block (XA-module) along with the previous one (xax-module) is shown in Figure 4.4.



(a) TSPC D flip-flop (b) TSPC T flip-flop

Figure 4.2: Circuit schematics used for domino logic T and D flip-flops [5]



(a) XOR-AND-XOR function (b) XOR-AND function

Figure 4.3: Circuit schematics used for xor-and-xor function (from Chapter 3), and the xor-and function used in this chapter

It should be noted that all of the flip-flops used for this design are negative-edge triggered. This is done to maximize the amount of time available for combinational logic to evaluate, enabling higher maximum operating speeds. The T flip-flop schematic is shown in Figure 4.2b.

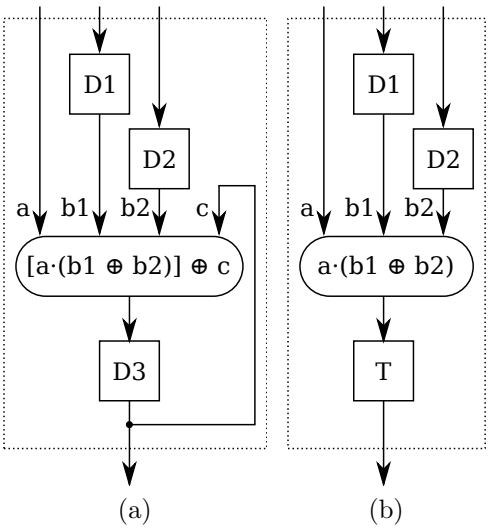


Figure 4.4: Block diagrams for (a) xax-module from Chapter 3, and (b) the XA-module proposed here. Replacing the D flip-flop D3 with a T flip-flop allows the function to be simplified.

After determining the appropriate transistor sizes in a schematic editor, the layout of the XA-module was created in a 6-metal CMOS 0.18 µm^2 process; it is shown in Figure 4.5, and its dimensions are 18.68 µm × 12.32 µm. Additional fine-tuning was performed post-layout in an effort to reduce parasitic capacitance and further increase speed. The XA-module was deliberately designed to be twice the height of one of the library cells, allowing it to share the same size power rails without any floorplanning difficulties.

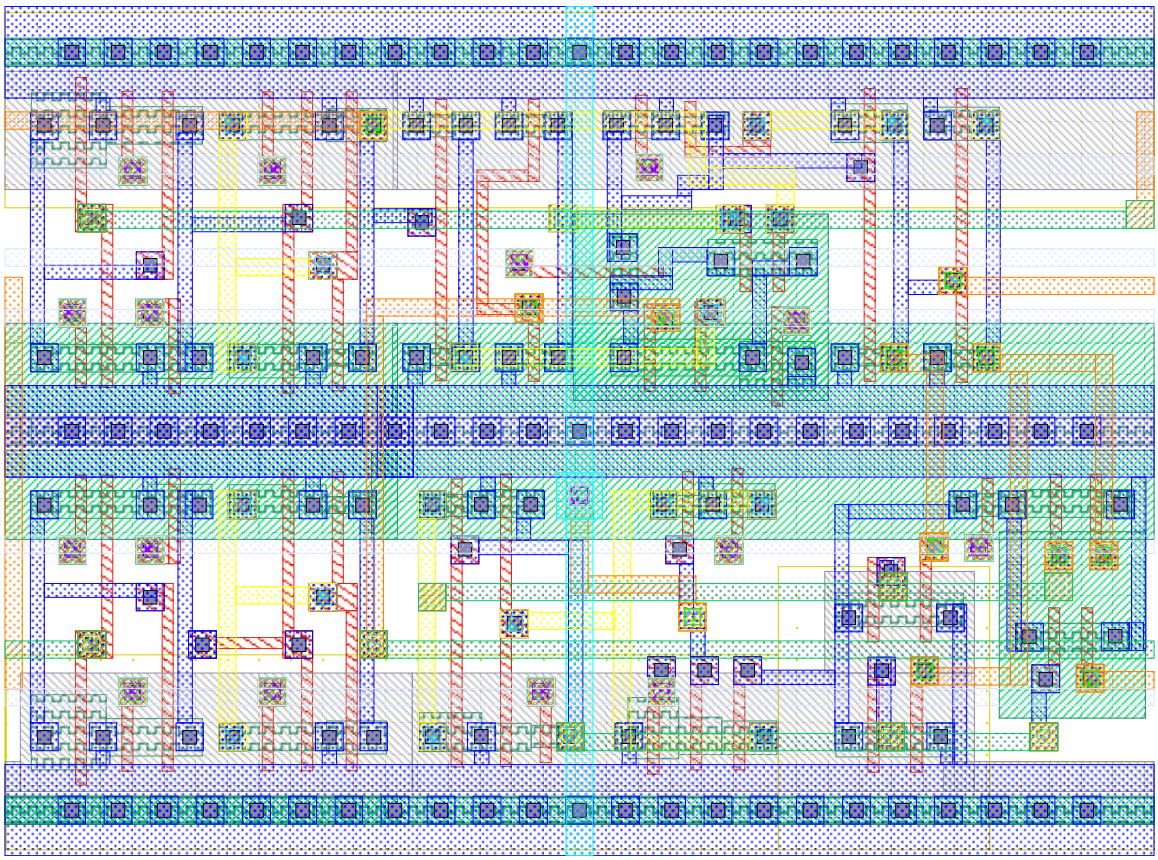


Figure 4.5: XA-module Layout

4.3 Design and implementation of the 233-bit multiplier using the XA-module

Using XA-modules as a building blocks, desgining multipliers of arbitrary size is easily achieved by connecting XA-modules together serially, as shown in Figure 4.1. We have designed a 233-bit multiplier. This size was chosen because it is practical for cryptographic applications, and recommended by NIST for elliptic curve cryptography [35].

In order to have a roughly square geometry (which is desirable when floorplanning), the chain of XA-modules was shaped into an array of 13 columns 18 rows. Additionally, two buffers are used on every row: one 16X drive-strength buffer to drive the clock signal, and an 8X buffer for the 'input-a' signal. Standard library cells

were used for this function, and appear as empty boxes in Figure 4.6.

One special module was created to serially load the coefficients of element 'B' into the XA-modules via an external input. Once complete, the load signal changes and the D flip-flops of all of the XA-modules behave as one large shift register, as shown in Figure 4.1. The proposed 233-bit multiplier layout dimensions are 277 µm × 224 µm for a total area utilization of 62048 µm^2.

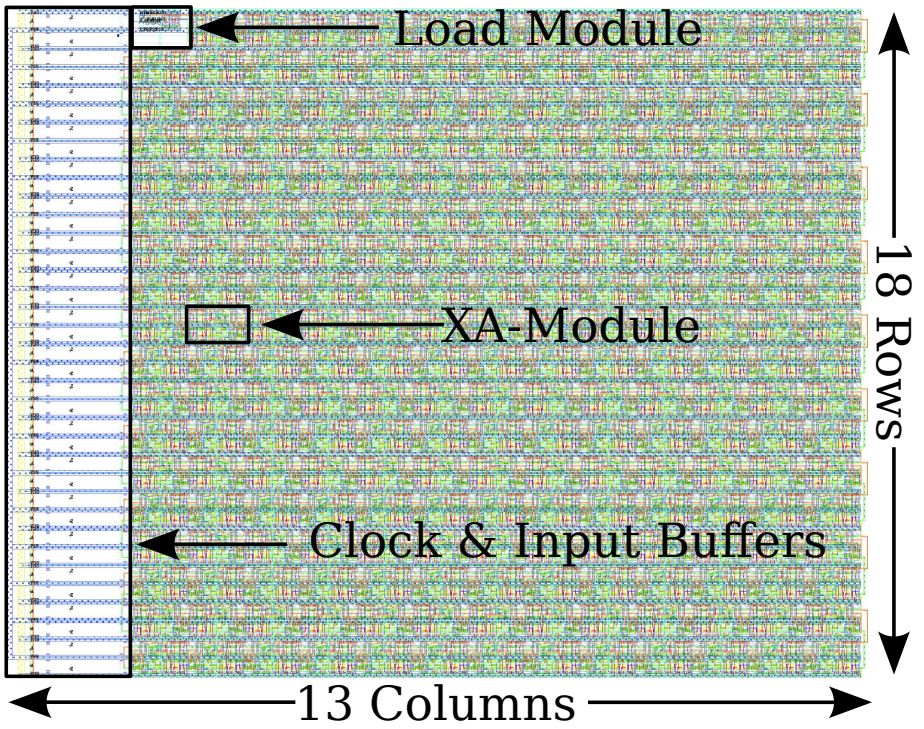


Figure 4.6: 233-bit Multiplier Layout

4.4 Simulation results

A series of waveforms displaying the correct operation of one of the multiplier's XA-modules at its maximum operating frequency of 1.79 GHz is shown in Figure 4.7. From top to bottom, the waveforms are: the system clock, input A, input B1, input B2, the result of the XOR-AND function, and the T flip-flop output. It can be easily verified that the circuit is functioning correctly. On the falling edge of every clock

cycle, the XOR-AND function is computed, determining $a \cdot (b_1 \oplus b_2)$. If this signal evaluates to logic 1, the T flip-flop is toggled. The simulation was performed in Cadence's Analog Environment using Spectre. Finally, it is important to note that the presented results include the parasitic capacitances extracted from the physical layout, and that all input signals were passed through appropriate buffers to ensure a realistic (limited) drive strength.

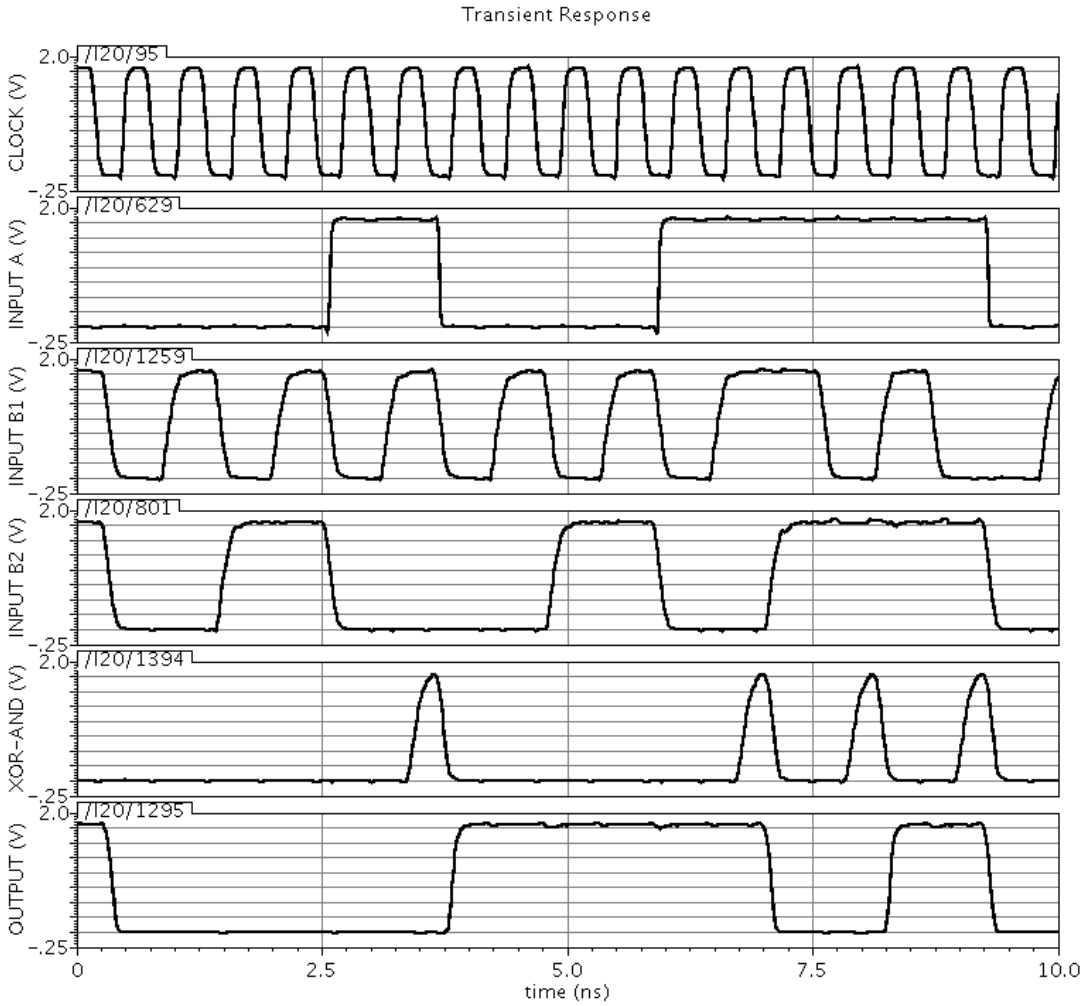


Figure 4.7: Simulation Voltage Waveforms

4.5 Comparison of similar implementations

Table 4.1 compares the proposed design to several published results. All entries in the table are for 233 finite field multipliers, with the exception of Ansari et al. [56] which uses a field size of 163. All designs are implemented in a CMOS 0.18 µm process. Note that the designs presented in the first two rows employ a polynomial basis, whereas the rest use ONB type II.

Table 4.1: Comparison of finite field multiplier implementations

Architecture	Base	Field Size	Max Clock Freq. (MHz)	Multiplication Delay (ns)	Power (mW/MHz)	Area (μm^2)	Area × Delay ($\times 10^6$)
Ansari [56]	Poly	163	125	40	-	1,272,102	50.884
Tang [55]	Poly	233	130	223	0.1843	189,297	42.213
Static[1]	ONB II	233	796	293	0.1848	216,737	63.504
Domino[1]	ONB II	233	1587	147	0.0851	109,644	16.118
Proposed	ONB II	233	1,790	130	0.0837	62,048	8.066

Some additional remarks are as follows: All designs presented in this table have been implemented using the same $0.18\mu m$ CMOS technology. Ansari's [56] design has the least delay overall, however it should be pointed out that it accomplishes this via a large amount of parallelization, which requires a vast silicon area. Tang's [55] architecture is a word-level multiplier, with a bus width of 8 bits, which is why it completes a multiplication operation in 223 ns despite having a comparatively low clock speed. Finally, the last three rows of the table present the results of the various serial-in parallel-out finite field multipliers, including the proposed design. The third row presents the static CMOS design, which was the result of synthesized HDL code, a logic compiler, and place-and-route tool. The fourth row presents a domino logic design which used the xax-modules shown in Figure 4.1.

As shown in this table, the proposed design is 12% faster and 43% smaller than the next fastest design (Domino[1]), which was presented in Chapter 3. A performance measure of area × delay is proposed in order to easily compare the architectures' overall performance, and is shown in the rightmost column of Table 4.1. It can be seen that the proposed design compares favorably to other multipliers, as the its area

$\times$ delay is 50% smaller compared to other designs listed in the table.

4.6 Summary

In this chapter, a new VLSI implementation of a 233-bit serial-in parallel-out finite field multiplier is presented. The field size of 233 bits is in the practical range for embedded security applications, and is recommended by NIST. The proposed design was shown to be 43% smaller, and operate 12% faster compared to the implementation presented in Chapter 3. This design is easily scaled to any practical size multiplier, by simply adding additional building blocks, and additionally, the design could be integrated into a CPU to implement binary field multiplication in hardware, enabling very high-throughput scalar point multiplication. The results of this work were published in [2].

CHAPTER 5

A Review of GPU Programming

5.1 Introduction

Section 5.2 of this chapter presents a brief history of GPU computing, followed by a detailed examination of the architectural differences between CPUs and GPUs in terms of cache, instruction set, execution units, and register file in section 5.3. The methods used by the GPU to parallelize processing are presented, along with some of its pitfalls in section 5.4. Finally, some concluding remarks are given in section 5.5.

5.2 A brief history of GPU computing

Using GPUs for general purpose computing is a new paradigm that has been receiving attention from academia since 2003, and official support from GPU manufacturers in 2007. The core concept is simple enough: employ a graphics processing unit (GPU) that is normally used to render graphics for 3-D computer games to carry out general

purpose computations in place of a CPU. A GPU is a single-program, multiple-data (SPMD) machine with user controlled cache, a large number of ALUs, a generously-sized register file, and its own dedicated off-chip RAM. Using a GPU in the place of a CPU is highly advantageous for applications which can be parallelized to take advantage of the GPU's resources.

In 2003, a group of researchers at Stanford University pioneered GPU computing when they developed a compiler project called "BrookGPU" that allowed programs to be developed for execution on GPUs [58]. In its nascent state, GPU computing required that the input data are formatted as computer graphics texture information; computations were performed by having the GPU operate on these data using existing graphics rendering routines that are part of driver packages such as Microsoft's DirectX or OpenGL. Luckily, a large number of the various algebraic transforms normally used to render a 3-dimensional scene employ matrix multiplication, which is an operation that is also frequently needed for applications in scientific computing. GPU manufacturers NVIDIA and AMD (formerly ATI) took notice of this phenomenon, and in 2007 both companies began offering products that allowed direct use of the GPU hardware for scientific computing, without having to encode and process the data as textures.

Since this time, the GPU computing market has exploded. Thousands of papers have been published detailing how GPUs can be used to improve program runtimes by orders of magnitude for a vast array of applications. Rather than focusing exclusively on improving the performance of 3-D computer games, AMD and NVidia are now improving the scientific computing ability of their products as well.

5.3 Differences between GPUs and CPUs

The GPU architecture used for the work presented in this dissertation is called the "NVIDIA Fermi", and the specific video card used is the Zotac GTX-480. AMD also manufactures GPUs which may be used for general purpose computing, however their toolsets, documentation, and community support are not as mature as NVIDIA's at this time. While the work presented here is optimized for the NVIDIA architecture, it is fully expected that many of this work's contributions should work with similar effect on AMD hardware.

There are a number of significant differences between GPUs and CPUs in terms of instruction set, architecture, and how transistors are allocated. This section highlights the major differences between the recently released Intel core-i7 3770 desktop CPU, and the NVIDIA GTX-480 GPU used for the work presented in this dissertation.

5.3.1 Physical specifications

Table 5.1 presents a number of physical specifications for both devices. The GPU die size is significantly greater than the CPUs, and it possesses over twice as many transistors. Power consumption is much higher, which is due to the increased transistor count and also because the Intel CPU uses a more recent 22 nm fabrication process. The CPU's clock frequency is over twice that of the GPU's; the GPU compensates for its slower clock with massive parallelization. The 3770 has 4 physical cores, while the GTX-480 has 15 streaming multiprocessors (SMs).

5.3.2 GPU and CPU instruction sets

The Intel core-i7 3770's cores employ a massive instruction set which includes over a thousand instructions belonging to several groups [59]:

- SSE - streaming SIMD extensions

Table 5.1: Comparison of price and physical specifications for the Intel 3770 and NVIDIA GTX 480

Specification	Intel core i7-3770	NVIDIA GTX 480
Price (USD)	$310	$250
Fabrication Process	22 nm	40 nm
Processors per device	4 cores	15 SMs
Transistors Count	1400 M	3200 M
Die Size	160 mm2	529 mm2
Thermal Design Power	77 W	250 W
CPU Speed	3.6 GHz	1.4 GHz

- MMX - another SIMD instruction set
- AES - special-purpose cryptography instructions
- AVX - vector instructions
- The base x86 instruction set, which has been growing steadily since its creation in the late 1970s' for the Intel 8086 microprocessor

The GTX-480 uses a total of 96 instructions; compared to the CPU, this significantly reduces the area utilization of the instruction decoder, and allows the GPU to spend its transistors on other components [6].

5.3.3 Serial processing features

Desktop and server CPUs, the programs targeting these architectures, and the entire compiler tool chain, are heavily optimized for serial execution. Accordingly, CPU architectures incorporate features to further improve serial program performance such as *branch prediction & speculative execution*, which allows queued instructions to proceed even if the result of a branch condition is not yet known. This is done by making an educated guess based on how the program branched in previous loop iterations, and it should be noted that if the branch predictor guesses wrong, the pipeline must be flushed, and instructions repeated with the (now known) correct

program branch. Depending on the benchmark, these features may improve serial program performance by 5-15% while consuming approximately 5% of the CPU core's area. GPUs do not possess any of these optimizations.

5.3.4 Cache

Desktop CPUs like the 3770 possess generous on-chip cache in an effort to improve instruction throughput by reducing the number of RAM memory accesses, which often form the system bottleneck. Table 5.2 highlights the various cache levels used in GPUs and CPUs. The CPU has two levels specific to individual cores, and a third, very large cache, which is shared among all cores. The GTX-480 only has two levels of cache; the first is per-SM, and the second is similar to the CPU's L3 cache in that it is shared across all processing units, however it is much smaller: 768kB instead of 8192kB. Overall, the GPU has roughly one fifth the cache of the CPU.

Table 5.2: Cache size comparison for the Intel 3770 and NVIDIA GTX 480

Cache Type	Intel core i7-3770	NVIDIA GTX 480
L1	64kB / core	80kB / SM
L2	256kB / core	768kB / GPU
L3	8192kB / CPU	-
Total per device	9472kB	1968kB

5.3.5 Register file

One of the areas where the GPU hardware designers spend the transistors saved from frugal cache sizes and a spartan instruction set is the register file. Table 5.3 gives a breakdown of the number of registers per processor, as well as a total per device. Note that the GPU has a single type of register which may be used for floating point and integer arithmetic, whereas the CPU uses different sets of registers for different tasks. The total size of the GPU register file dwarfs the CPUs, with a total count of 491,520 registers compared to 216.

Table 5.3: Register comparison for the Intel 3770 and NVIDIA GTX 480

Register type	Intel core i7-3770	NVIDIA GTX 480
General Purpose	16 64-bit / core	32768 32-bit / SM
Floating Point	8 80-bit / core	-
Memory Segment	6 16-bit / core	-
MMX	8 64-bit / core	-
XMM	16 128-bit / core	-
Total per device	216	491,520

Table 5.4: Comparison of execution units for Intel 3770 and NVIDIA GTX 480

	Intel core i7-3770	NVIDIA GTX 480
Per core or SM	1 MMX/SSE/FP ALU 1 MMX/SSE/FP Addition ALU 1 MMX/SSE/FP Mult. ALU	32 Integer / FP ALUs 4 Special function units
Total per device	12 Heterogeneous ALUs	480 ALUs, 60 SFUs

5.3.6 Execution units

The other major area where GPUs use more transistors than CPUs is in execution units. As shown in Table 5.4, each of the Intel 3770's cores have three different execution units that can execute different groups of instructions, with a total of 12 per CPU. The GTX-480 GPU has two types of execution units: a general purpose ALU which can perform integer and floating point arithmetic as well as logic operations, and a special function unit (SFU) that can carry out square root and trigonometric functions. Each SM has 32 general purpose ALUs and 4 SFUs for a total of 480 ALUs and 60 SFUs per device.

5.4 GPU computing concept

In order to use a GPU to execute a program, data is copied over the PCI-Express bus from the CPU (or *host*) RAM to the GPU RAM; GPU manufacturers supply appropriate APIs to handle this task with a simple function call. The CPU then instructs the GPU to initiate its program, which operates on the data stored in the

GPU RAM. Once the GPU program is complete, data is copied from the GPU RAM back to the host RAM using an API call.

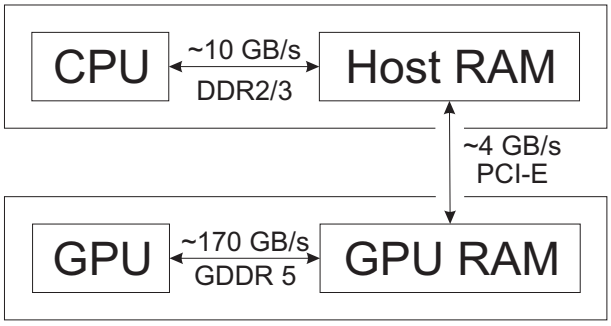


Figure 5.1: Data transfer between host machine and GPU

In order to efficiently employ this large number of execution units, a single program, multiple data (SPMD) paradigm is adopted. SPMD machines are very similar to single instruction, multiple data (SIMD) processors, the difference being that SPMD allows for divergent program branching, where SIMD does not.

5.4.1 Kernels, the compute grid, thread blocks, and threads

A program running on the GPU is referred to as a *kernel*. When the kernel is *launched* (executed on the GPU), it sets up a *grid*, which is an array of thread blocks, or simply *blocks*. Each block, in turn, contains an array of threads which are responsible for carrying out the actual work specified by the kernel. This is shown in Figure 5.2. A grid may possess up to a 2-dimensional array of blocks, and each block may have up to a 3-dimensional array of threads. Figure 5.2 has a 2×3 array of blocks, and each block has a 2×2 array of threads.

At this time, there are two language choices available for programming GPU kernels. The "Open Compute Language", or "OpenCL" was originally developed by Apple, and it now stands as an open, royalty-free standard governed by the Khronos Group. Programs written in OpenCL may be compiled for NVIDIA and AMD GPUs. The competing language is NVIDIA's proprietary "C for CUDA". Although C for

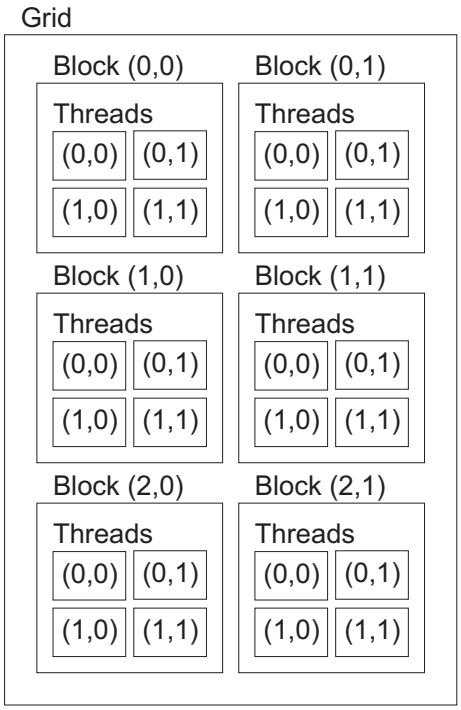


Figure 5.2: Hierarchy of the GPU compute grid, thread blocks, and threads [6]

CUDA is only available for NVIDIA GPUs, it has a more robust toolchain of compilers, assemblers, disassemblers and debuggers compared to OpenCL. Additionally, certain low-level features may only be used by programs written in C for CUDA. Due to the more mature toolset and access to low-level features, the GPU programs presented in this work have been written in C for CUDA.

Parallelization is achieved by having each thread execute the same program on multiple data elements. A practical example of this is unrolling a **for** loop whose arguments do not exhibit data-dependence: each iteration of the **for** loop may be computed in parallel by different ALUs. As shown in listing 5.1 is a pair of standard **for** loops which serially compute the result $c = a \cdot b$ where a, b, and c are 6×4 two-dimensional arrays of floating point elements:

Listing 5.1: Example C-code for multiplying together elements of two arrays

```
1 //Function to compute c = a * b for 6 by 4 2-dimensional
     arrays
2 int mult(float *a, float *b, float *c)
3 {
```

```
4   for(int i=0; i<6; i++)
5     for(int j=0; j<4; j++)
6       c[i][j] = a[i][j] * b[i][j];
7
8   return 0;
9 }
```

When using a GPU this may be written in C for CUDA using $6 \times 4 = 24$ individual threads by launching a thread array with an x-dimension of 6, and a y-dimension of 4. The GPU kernel performing the multiplication is shown in listing 5.2. First, *linear_index* must be computed using the built in variables *threadIdx.x* and *threadIdx.y*, which are the thread's unique X and Y coordinates, respectively. This allows each thread with coordinates *(threadIdx.x, threadIdx.y)* to access unique array elements in a, b, and c. The final line in the code listing computes the result of array c in parallel.

Listing 5.2: Example C for CUDA kernel code

```
1 //GPU kernel to compute c = a * b  in parallel
2 //Uses a 6 by 4 array of threads
3 __global__ void mult(float *a, float *b, float *c)
4 {
5   int thread_index = threadIdx.x + 2*threadIdx.y;
6   int linear address = thread_index;
7
8   c[linear_address] = a[linear_address] * b[linear_address
      ];
9 }
```

While the previous example uses a single block of threads with dimensions 6×4, it is possible to use multiple blocks with fewer threads per block to achieve the same result. Once multiple blocks are used, the same number of threads as before are employed, although their indexing must be modified slightly. Shown in listing 5.3 is an example that is functionally equivalent to the previous one: this time a 3×2 array of blocks is used, and each block employs a 2×2 array of threads, which is the same arrangement shown in Figure 5.2.

Listing 5.3: Example C for CUDA kernel code using multiple blocks

```
1 //GPU kernel to compute c = a * b  in parallel
2 //Uses a 3 by 2 grid of blocks
3 //Each block has a 2 by 2 array of threads
```

```
4  __global__ void mult(float *a, float *b, float *c)
5  {
6    int thread_index = threadIdx.x + 2*threadIdx.y;
7    int block_index = 4*blockIdx.x + 8*threadIdx.y;
8    int linear_address = thread_index + block_index;
9
10   c[linear_address] = a[linear_address] * b[linear_address
       ];
11 }
```

This code listing first computes the thread index as before, while it also calculates *block_index*. These are added together in order to determine the unique linear memory address so that each thread operates on unique elements in the a, b, and c arrays.

GPU computing may use up to 3-dimensional thread arrays, and up-to two-dimensional arrays of blocks may be used.

5.4.2 GPU memory spaces

As shown in Figure 5.3, the Fermi architecture possesses a number of different memory spaces. Individual threads have a private register space, as well as a user-controlled cache that is shared between other threads belonging to the same block. Threads in the same block may cooperate with each other by writing to (and reading from) the shared cache.

Threads also have access to a RAM memory, which can also facilitate communication between different threads. Depending on the specific GPU card, global memory can be anywhere from a few hundred megabytes to several gigabytes; the GTX-480 card used for this work has 1576 megabytes of high-bandwidth GDDR5 RAM. Unlike the cache, any and all threads may share information with each other through the GPU RAM, however barrier synchronization operations are required to ensure memory coherency; without synchronization, it may be possible to read an outdated value from RAM, as the order of execution for threads belonging to different blocks is undefined.

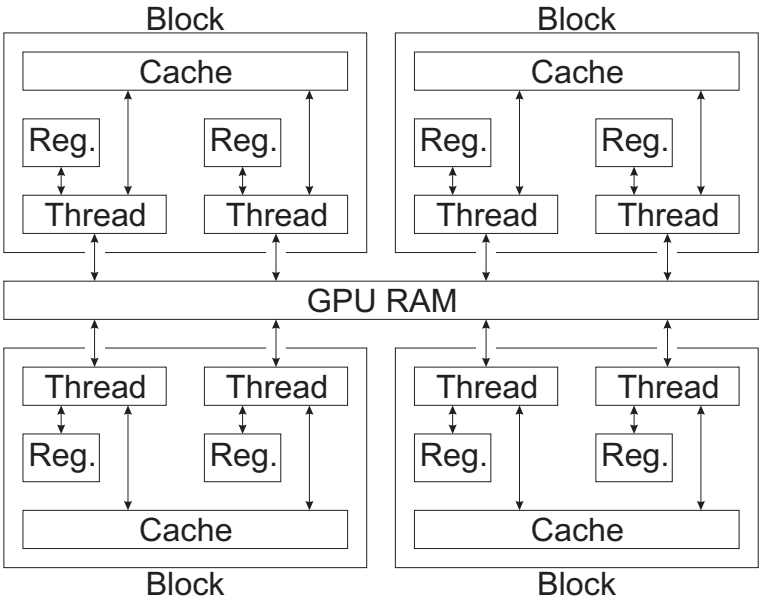


Figure 5.3: The Fermi architecture memory hierarchy [6]

In terms of latency, registers are of course the fastest and require a single clock cycle to access. Exact cache latencies are not officially disclosed by NVIDIA, however it is estimated that the cost is between 10 and 100 clock cycles. Finally, it should be noted that RAM accesses are the most expensive. Despite the high-bandwidth GDDR5 RAM used on the GTX-480 card (which boasts a maximum throughput of 170 GB/s), it is almost always the bottleneck of a GPU program, and care must be taken to minimize the host/device memory traffic. GPU RAM access latency is quite high, and while exact figures have not been released, NVIDIA states that they typically require 400-1200 clock cycles.

A complete block diagram of the SM is shown in Figure 5.4, including register file, cache, and the load/store units that fetch and store operands. Also shown are the hardware thread schedulers, which are explained in section 5.4.3.

5.4.3 Warps, concurrency, and resource allocation

The ALUs are deeply pipelined, requiring between 18 and 26 stages; this requires a large amount of instructions in flight to saturate the SM's compute resources. In

practice, this is achieved by queueing a *very* large number of threads on the device in batches of 32 threads called a *warp*: Up to 1536 threads may be actively processed by each SM, that is, up to 1536 threads may have registers and cache allocated. More than 1536 may be queued, however processing on the additional threads is stalled until there are available openings on the GPU.

A large number of active threads per SM allows the high-latency RAM accesses to be effectively hidden. As long as the thread scheduler can find a warp of threads whose input operands are available, they are set in flight, which allows more time for other warps to fetch their operands from RAM or cache.

5.4.4 Program branching

As previously stated, the difference between SIMD and SPMD machines is that SPMD architectures allow for branching. As described in section 5.4.3, threads are issued in batches of 32 called warps. Should an **if**-statement be encountered in a program, and a single thread in a warp branches differently than the rest, the remaining 31 thread lanes are masked, forcing them to wait for the divergent thread to complete its work before their execution can resume as a group. If, on the other hand, *all* threads take the same execution path, there is no penalty. In practice, careless use of nested **if**-statements quickly serializes a GPU program, severely affecting performance.

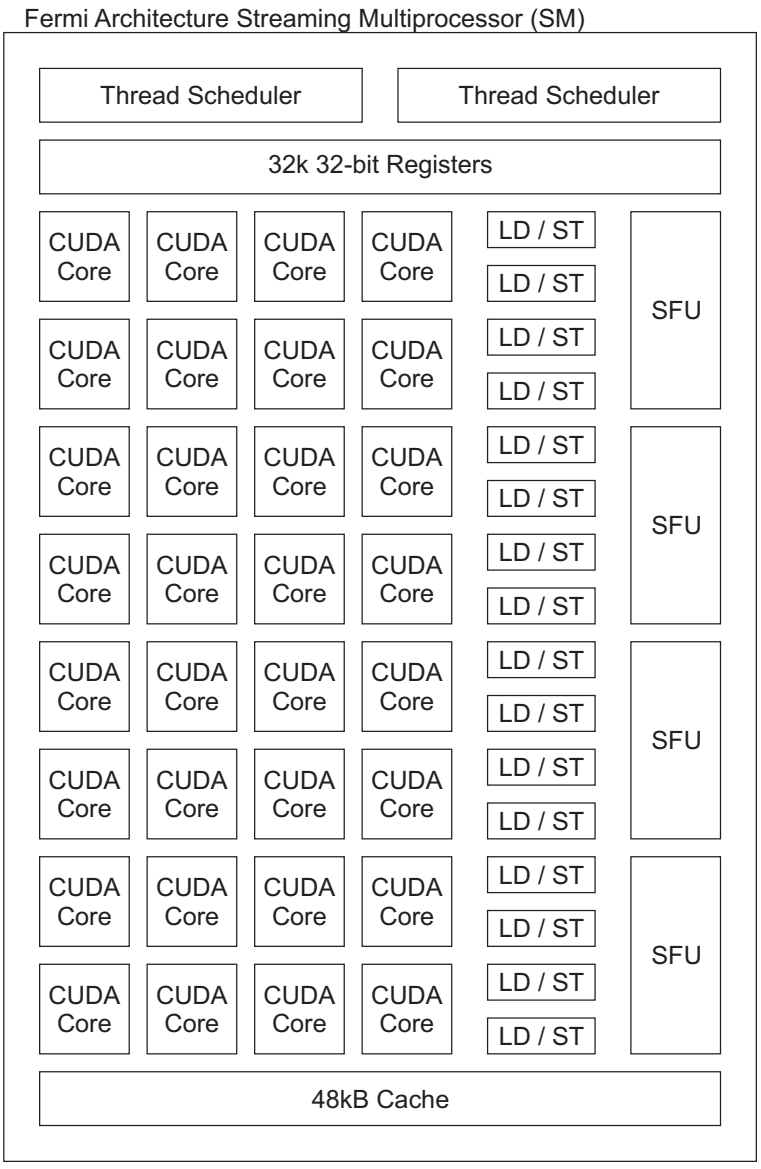


Figure 5.4: Fermi Streaming Multiprocessor (SM) architecture block diagram [6]

5.5 Summary

This chapter presented a review of GPU computing. The architectural differences between GPUs and CPUs were discussed. The organizational structure of GPU computing in terms of threads, blocks and grids was presented, and some simple example programs demonstrating how GPUs parallelize program execution, and process it in groups of threads called warps, were examined. Finally, the GTX-480 Fermi architecture used for the work presented in this dissertation was presented in depth.

CHAPTER 6

Type II Optimal Normal Basis Multiplication for GPU

6.1 Introduction

In this chapter, an ONB multiplication algorithm that takes advantage of the GPU's massive parallelism is proposed. In normal basis, each of the $\lceil m/32 \rceil$ 32-bit words which form the result of an m-bit wide multiplication $C = A \cdot B$ may be computed independently without any carry operations. This property allows for easy bit-slicing, where each 32-bit result word can be computed in parallel by multiple ALUs: a good fit for GPUs.

NB, ONB, and GNB multiplication is much more commonly implemented in hardware than in software due to its heavy reliance on multi-word circular shift and logic operations. Such operations are, of course, inexpensive to carry out in hardware using circular shift registers and simple logic gates. When executed by a CPU, logic operations may require as many clock cycles as floating point or integer multiplication. Multi-machine-word circular shifts are especially expensive due to the large

number of memory accesses required, and in practice, comb-type polynomial basis multipliers have greater operation throughput compared to the best software ONB implementations [60].

In place of pre-computing and storing shifted values of the multiplication's input operands as in most ONB software implementations, the operands are stored in the user-controlled cache, and shifted operands are reconstructed every time they are needed. Although this may seem more costly than pre-computation, due to the cache's low latency compared to the GPU RAM, and the ability to exactly control cache memory accesses, this is not the case.

Compared to other GPU-based binary field multiplication algorithms, the proposed is significantly faster, however it falls behind CPU implementations using polynomial basis, indicating that further research in this area is needed.

The rest of this chapter is organized as follows. Section 6.2 presents related work, and section 6.3 details the proposed algorithm. Specific implementation details are explained in 6.4, results are compared in section 6.5, and some concluding remarks are given in 6.7.

6.2 Related work

There are a number of software algorithms for implementing ONB and GNB multiplication in the literature. Rosing [27] proposed an ONB multiplication algorithm that pre-computes and stores all m circular shifts of the A and B operands, and computes the partial sum in Equation 6.1 using word operations.

$$c_k = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} a_{i+k} b_{j+k} \lambda_{ij0} \tag{6.1}$$

Ning and Yin [61] improve the pre-computation strategy used by Rosing [27], reducing memory use (and number of instructions required for pre-computation) from

complexity $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$. Once the precomputations are complete, Ning and Yin use the same algorithm as Rosing to accumulate the partial sum and produce the result of the multiplication.

Reyhani-Masoleh and Hasan [62] proposed a series of algorithms which could be used for GNB, as well as ONB. Their work also proposes a pre-computation strategy that uses less memory compared to [61], however [60] reports that Ning and Yin's approach is faster in practice. Dahab et al. also propose some fast GNB multiplication algorithms [60]; theirs use a a pre-computation strategy similar to Ning and Yin [61], while their algorithm focuses on reducing the number of expensive circular shift instructions. Their work also shows that Ning & Yin's algorithm [61] is the fastest for Type-II ONB multiplication [60].

6.3 Proposed algorithm

The proposed type-II ONB multiplication algorithm for GPU is based on Rosing's work [27], which noted that for $i = 0$, expanding $\sum_{j=0}^{m-1}$ in section 6.2 has a single non-zero entry in λ_{ij0}, and for all other $i \neq 1$, expanding $\sum_{j=0}^{m-1}$ has exactly two non-zero values in λ_{ij0}; this property is guaranteed for all type-II optimal normal bases. Rosing also proposed the use of two lookup tables, t_1 and t_2 to hold the non-zero values of λ_{ij0} for each i. The λ_{ij0} table, and the associated lookup tables t_1 and t_2, for the ONB multiplication example presented in Chapter 2 on page 17 are shown in tables 6.1a and 6.1b.

The use of lookup tables allows Equation 6.2 to be re-written as Equation 6.3. Note that the $i = 0$ term has been expanded from the sum over i: the term $i = 0$ differs from the other others in that it corresponds to the only entry in λ_{ij0} that does not have two non-zero terms.

i \ j	0	1	2	3	4
0	0	1	0	0	0
1	1	0	0	1	0
2	0	0	0	1	1
3	0	1	1	0	0
4	0	0	1	0	1

(a) λ_{ij0} table

i	t_1	t_2
0	1	-
1	0	3
2	3	4
3	1	2
4	2	4

(b) t_1 and t_2

Table 6.1: λ_{ij0}, t_1, and t_2 tables for the optimal normal basis multiplication Table 2.3

$$c_k = \sum_{i=0}^{m-1} \left(\sum_{j=0}^{m-1} a_{i+k} b_{j+k} \lambda_{ij0} \right) \tag{6.2}$$

$$= (a_k b_{t1[0]+k}) + \sum_{i=1}^{m-1} [a_{k+i} \cdot (b_{t_1[i]+k} + b_{t_2[i]+k})] \tag{6.3}$$

Rosing's approach processes the bits of c_k simultaneously using word operations. For a machine word size of 32 bits, the result $C = A \cdot B$ is computed using $W = \lceil \frac{m}{32} \rceil$ iterations, each of which computes a batch of 32 bits of the result. Pseudocode for Rosing's algorithm is shown in algorithm 6.1. Note that ONB elements are represented as arrays of unsigned integers which are denoted using upper-case characters A, B, and C. Each array is W words long, and the 32-bit words that form the arrays are denoted as $a[i]$, for $0 \leq i < W$.

6.3.1 Compute grid

In this work, the proposed GPU algorithm fully parallelizes the partial-accumulation of C using a bit-slicing approach across W threads. This takes advantage of the fact that the inner loop which calculates the partial products $c[j]$ on line 6 of algorithm 6.1 does not exhibit any data dependency on the other $c[j]$.

The proposed approach processes each multiplication operation using W threads

Algorithm 6.1: ONB Multiplication algorithm [27]

Input: Finite field elements A and B, lookup tables t_1 and t_2
Output: Result $C = A \cdot B$
// $W = \lceil \frac{m}{32} \rceil$, the number of 32 bit words representing A, B, and C
// a_s denotes an s-bit left circular shift of a

```
for j ← 0 to T − 1 do
    temp ← b_{t1[0]+j·32}
    c[j] ← a_{j·32} AND temp
end
for i ← 1to m − 1 do
    for j ← 0 to W − 1 do
        temp ← b_{t1[i]+j·32}
        temp ← temp XOR b_{t2[i]+j·32}
        temp ← temp AND a_{i+j·32}
        c[j] ← c[j] XOR temp
    end
end
return C
```

at a time instead of one, however, this is not nearly enough work to saturate the GPU's resources and obtain computational efficiency as well as high operation throughput. To remedy this, every block of threads also computes N multiplications in parallel.

The thread block arrangement used for the proposed GPU algorithm is shown in Figure 6.1. Along the thread block's X-dimension are the W threads computing the result, while along the Y-dimension are the N multiplication results the block is computing in parallel. As stated earlier, the X-dimension is simply $W = \lceil \frac{m}{32} \rceil$, and the Y-dimension ($N$) is chosen to maximize the compute efficiency of the GPU. A one-dimensional array of blocks is used.

In order to choose the best N, the number of threads per SM $\text{Threads}_{\text{SM}}$ must be maximized subject to the hardware constraints constraints that maximum number of blocks per sm N_{blocks} is 8, and $\text{Threads}_{\text{SM}} \leq 1536$, as shown in Equation 6.4.

$$\text{Threads}_{\text{SM}} = \left(\lceil \frac{m}{32} \rceil \times N \right) \times N_{\text{blocks}}, \qquad 1 \leq N_{\text{blocks}} \leq 8, \ \text{Threads}_{\text{SM}} \leq 1536 \quad (6.4)$$

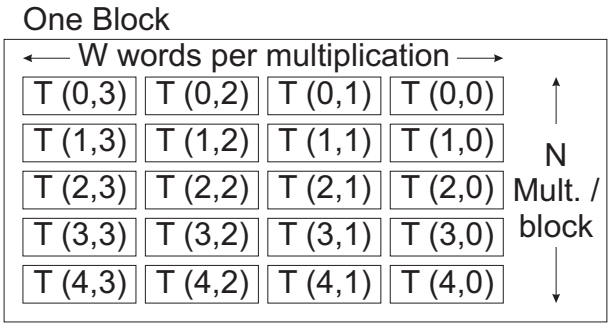


Figure 6.1: Thread-block arrangement for proposed ONB multiplication algorithm: in this example, each multiplication produces a $W = 4$-word wide result, and $N = 5$ multiplications in total are being carried out by this block

6.3.2 Memory layout

When storing operands to RAM, it is important to consider how memory access patterns can affect performance. The GPU RAM can only obtain peak transfer bandwidth if adjacent or *coalesced* memory accesses are performed. The GTX-480's RAM, for example, requires contiguous 128-byte (32-word) memory accesses to achieve peak performance.

As such, the proposed algorithm stores operands in GPU RAM using a structure of arrays approach as shown in Figure 6.2: all A-operands are stored contiguously, as are the B and C operands. This ensures coalesced memory accesses allowing the highest possible memory bandwidth.

6.3.3 Multi-word parallel circular shifting

The proposed algorithm requires a large number of multi-machine-word circular left-shift operations. As noted earlier, in algorithm 6.1, a_s denotes left shifting the m-bit wide a-operand by s bits.

While all software implementations of ONB in the literature use some type of precomputation strategy to reduce the number of multi-word circular shifts required, the proposed GPU algorithm simply stores double-wide "wrapped" operands to the cache, and reconstructs the shifted values as required. Due to the availability of

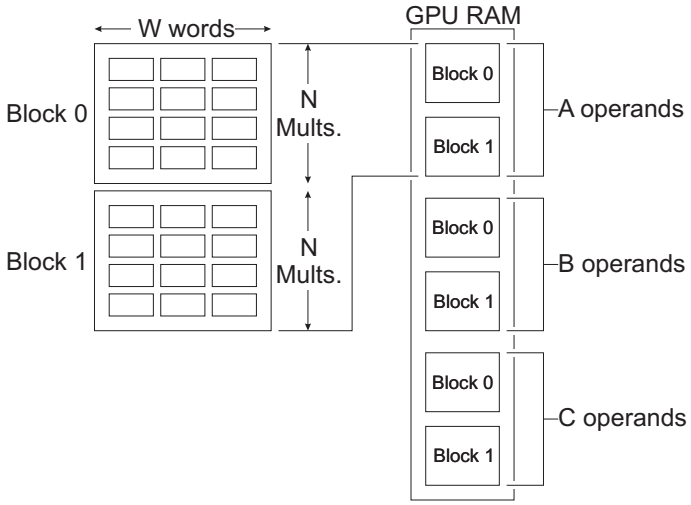


Figure 6.2: Memory for the proposed algorithm is arranged in a structure of arrays format to ensure memory accesses are coalesced

a user-controlled low-latency cache and an abundance of ALUs, it is less expensive in terms of clock cycles to recompute the shifted operands with each iteration of i compared to fetching precomputed values from the GPU RAM.

Constructing the double-wide arrays

The double-wide wrapped operand arrays are constructed from the original operands as shown in Equation 6.5, where $l = \lceil \frac{m}{32} \rceil \times 32 - m$, the number of unused bits in the operand arrays; additionally, "$<< s$" and "$>> s$" denote left and right shift operations by s places. An example of a double-wide, wrapped operand is shown in Figure 6.3.

$$a_w[i] = \begin{cases} a[i] & \text{if } 0 \leq i < W-1 \\ a[W-1] \text{ OR} \Big[a[0] << (w-l) \Big] & \text{if } i = W-1 \\ \Big[a_w[i-W] >> l \Big] \text{ OR} & \\ \Big[a_w[i+1-W] << (w-l) \Big] & \text{if } W-1 < i < 2W-1 \end{cases} \tag{6.5}$$

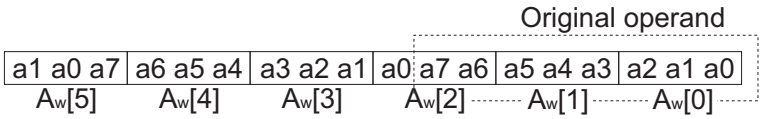


Figure 6.3: Example of double-wide "wrapped" operand, with $W = 3$, word size $= 3$, $l = 1$

Constructing shifted operands

With the double-wide arrays loaded into the user-controlled cache, multi-word circular left shifts are computed in parallel for all W threads along the X-direction of the thread array. The code listing which performs this task is shown in 6.1. In order to parallelize the circular shift, each thread must fetch array elements from the double-wide array. Given an amount to shift the operands by "s", which is provided by the lookup tables $t1$ and $t2$ previously described, bit and word-level offsets are first calculated as shown in lines 8 and 9. Each of the W threads adds this offset to its respective "threadIdx.x" index, modulo W. Finally, this value is used to load the proper double-wide array element from A_w, and it is shifted to the right using the "shift" offset previously calculated, and stored to "RS_temp". A similar series of steps are carried out in order to determine "LS_temp", which is added to "RS_temp" to form a word of the shifted operand.

Note that "shift", "word", "word+1", and "32-shift" are all known quantities at compile time, and by using a technique such as loop unrolling, it is possible to pre-compute them all, which leads to a much more efficient parallel circular shift operation.

For clarity, the code listing omits the block and multiplication number offsets which allow parallelization along the Y-direction of the thread array, and across multiple blocks, respectively. These additional offsets are simply added to the array address in lines 10 and 11.

Listing 6.1: Multiprecision circular shift operations using W threads

```
1 //Inputs: double-wide wrapped operand array 'Aw'
```

```
//      number of bits to shift by 's'
//      thread's x-coordinate, threadIdx.x
//Output: 'ashifted', the shifted version of the 'Aw' that
    corresponds to threadIdx.x

int RS_temp, LS_temp, a_shifted;

int shift = s % 32;
int word = (s - shift) / 32;
RS_temp = Aw[(word + threadIdx.x) % W] >> shift;
LS_temp = Aw[ (word+1 + threadIdx.x) % W] << (32 - shift);

a_shifted = RS_temp + LS_temp;
```

6.4 Implementation details

The proposed algorithm was implemented in C for CUDA, using some special programming techniques to ensure maximum performance. Loop unrolling was used to take advantage of the fact that a number of the values in the circular shift algorithm can be precomputed at compile time. While this has the effect of expanding the program size, it should be recalled that instruction fetches are cached, and instruction fetch cost is amortized across thousands of threads.

Loop unrolling proved difficult using the NVIDIA "nvcc" compiler, however: it simply did not honour requests to unroll more than 10 loop iterations. In order to work past this, a C program was developed to manually unroll the loop and print out the resulting C for CUDA code, which was then inserted into the main program.

While this allowed the loop to be fully unrolled, the nvcc compiler performed extremely poorly in assigning variables to registers. Further investigation of the resulting assembly code revealed that the nvcc compiler had failed mapping its intermediate single-static-assignment form to individual registers. Luckily, further research revealed that the use of the "volatile" keyword could assist the compiler is mapping

from single-static assignment to individual registers. This allowed loop unrolling to succeed, which resulted in over twice the operation throughput; the performance of the loop-unrolled program is presented in section 6.5.

6.5 Testing and validation

The proposed work was tested on field sizes $m = 173$, 233, 281, 410, and 575. These field sizes were chosen to allow the proposed algorithm to be compared to polynomial basis multipliers using the popular NIST fields. Aside from the field $\mathbb{F}_{2^233}$, the NIST fields do not use type-II ONB basis, which is why the next closest field sizes where a type-II ONB exists were used [35].

Performance testing consisted of randomly generating several million sets of input operands and processing them with the GPU. Timing was measured using CUDA timing API calls, and very little variation in runtime (less than 1%) was observed between program runs using different sets of random data. The results were validated by checking them against results generated by a publicly available ONB multiplication program from Rosing [27] which used the same input operands.

6.6 Results and comparison

Presented in the second column of Table 6.2 are the results from Cohen and Parhi [63], which is the only other binary field multiplication algorithm for the GPU in the literature. In their approach they used polynomial basis, and the GPU processes a single multiplication at a time in parallel across W threads. Note that their algorithm under-utilizes the GPU's resources for smaller field sizes; there is virtually no difference in operation throughput between a 233 and 409 bit multiplication. The GPU used for their work is an NVIDIA GTX 285, which is one generation behind the GTX-480 used in the proposed implementation.

Table 6.2: Finite field multiplication average operation throughput in 10^3 multiplications per second

Field Size	Cohen[63] and Parhi	Taverne et al. [26]	Proposed
163/173	20		27,027
233	19	27,717	19,608
281/283	19		10,989
409/410	18	10,426	5,236
571/575	16		2,667

In the third column from the left is a CPU implementation carried out by Taverne et al. [26]. They developed a heavily optimized assembly implementation of binary field multiplication using Intel PCLMULQDQ instruction for carry-less multiplication for the 3.325 GHz 22 nm, Intel Westmere core-i5 660 CPU. Taverne et al. also use polynomial basis for their approach.

To the best of the author's knowledge, Taverne et al. currently hold the record for highest binary field multiplication operation throughput for any single device.

Finally, the proposed algorithm's operation throughput is shown in the rightmost column. To the best of the author's knowledge, the proposed algorithm is currently the fastest GPU-based binary field multiplication algorithm, and by a significant margin. Compared to a highly optimized CPU implementation using assembly instructions, it is, however, between 1.4 and 2.0 times slower.

6.7 Summary

Overall, the operation throughput performance for the presented GPU implementation of a type-II optimal normal basis multiplier compares favorably with the state of the art of other GPU binary multiplication implementations. The optimal normal basis' carry-less multiplication properties do lend themselves well to a bit-sliced GPU implementation, however this approach is not able to compete with highly optimized CPU programs that employ Intel's carry-less multiplication instruction.

Further research in this area may prove fruitful; the more recent NVidia Kepler GPU architecture nearly doubles the peak theoretical logic and shift instruction throughput compared to previous generations, and this may prove a serious contender to the state of the art CPU implementation. This work could be extended to use GNB, so that any of the NIST fields can be used, further extending its utility. Bypassing the nvcc compiler and re-writing the unrolled main loop in assembly may also allow better use of GPU resources.

CHAPTER 7

High-Throughput NIST Prime Field Multiplication for GPU

7.1 Introduction

The previous chapter implemented a type-II ONB multiplication algorithm for the GPU. Although this algorithm could take advantage of the GPU's parallel computation capabilities, it was unable to surpass the state of the art desktop CPU implementation. Moving forward, a closer examination of the GPU's strengths steered research towards elliptic curve cryptography over prime fields, which relies heavily on integer multiplication.

As shown in Table 7.1, the GPU's floating point multiplication operation throughput is equivalent to most logical operation throughput, and it is actually *double* the throughput of a shift operation. Integer multiplication is also quite efficient, with a throughput of 16 operations per clock cycle, per SM.

Due to the GPU's high integer arithmetic performance, it is an excellent candidate for prime field multiplication, which is simply multiplication modulo a prime number.

Table 7.1: Instruction throughput in operations per clock cycle per SM for the Fermi GTX-480

Type	Operation	Operations per clock per SM
Integer	Addition	32
	Multiplication	16
Floating Point	Addition	32
	Multiplication	32
Logic	AND	32
	OR	32
	XOR	32
	Shift	16

This chapter presents a high performance prime field multiplication algorithm in the GPU's low-level virtual machine language, PTX. The proposed algorithm carries out finite field multiplication over the NIST prime fields of size 192, 224, 256 and 384 bits, which are employed in a number of standards [35], as described in Chapter 2.

This chapter begins in section 7.2 with a review of the state of the art of GPU-based prime field multiplication. A detailed presentation of the proposed high operation throughput NIST field multiplication algorithm is provided in section 7.3, and implementation details are provided in section 7.4. Results and comparison to the state of the art are given in section 7.5, while concluding remarks are provided in section 7.7.

7.2 History of prime field multiplication for the GPU, and the state of the art

Research in the area of GPU implementation of prime field arithmetic is a new and very active area. The first works attempting GPU-based prime field multiplication were published in 2007; Fleissner [64] developed a parallel Montgomery multiplication for the GPU and Moss et al. [65] used the residue number system (RNS [66]) to

parallelize multiplication across multiple threads. Note that at this time, GPUs did not support general purpose computing, and a tool chain for GPU programming was not available, requiring the data to be manipulated as texture data –an inefficient process.

In 2008, Szerwinski and Güneysu were first to use a proper toolchain to develop prime field multiplication [67]. Their implementation used the residue number system (RNS) to parallelize a multi-precision multiplication across multiple threads. In 2009, Giorgi et al. [68] experimented with the use of different memory spaces on the GPU in order to carry out prime field multiplication with the finely-integrated operand scanning (FIOS) method of the Montgomery multiplication algorithm; they determined that, as expected, multiplication is faster when all the operands can reside in registers. Also in 2009, Harrison and Waldron [69] use Montgomery multiplication with a serial approach: every thread computes a single multiplication, which increases the amount of time required per thread, although bulk operation throughput is greatly enhanced as all inter-thread communication is eliminated along with expensive synchronization operations.

Another pair of publications were released in 2011. Antao et al. [70] improve on the RNS approach, and apply their work to elliptic curve and RSA cryptography. In [71], Neves and Arauji perform a comparison of different Montgomery multiplication implementations on the GPU, and determine that the FIOS arrangement works best.

Finally, in 2012, Henry and Goldberg develop a coarsely-integrated operand scanning (CIOS) implementation of the Montgomery multiplication algorithm which takes advantage of in-line PTX code for multi-precision addition and subtraction [72].

7.3 Proposed algorithm

The goal of the proposed algorithm is to compute NIST prime field multiplication on the GPU with the highest possible operation throughput. This work is guided by the following principles:

1. Global (RAM) memory accesses are the most expensive operations; these must be limited
2. Idled threads must be kept to a minimum: program branching as a result of conditional statements, reduction networks, and thread synchronization events should be reduced or eliminated
3. The GPU is not efficient unless it carries out a large volume of computations
4. The very high ratio of ALUs and registers to cache memory is atypical when compared to CPUs; algorithms which focus on reducing the number of multiplication instructions while increasing memory accesses, idled threads, or addition operations are now much more expensive

The proposed algorithm has two main stages: (schoolbook) multiplication, followed by reduction. Due to this work's interest in the NIST prime fields, reduction is easily computed by exploiting the special form of the NIST prime numbers, as will be shown in a subsequent section.

7.3.1 Proposed thread layout

In the proposed algorithm, every individual thread processes a single modular multiplication operation start-to-finish, which was first proposed by [69]. While this increases the amount of time taken per thread, the elimination of any inter-thread communication allows each thread to process a result without interruption; as such, each block instantiates a one-dimensional array of threads. While this thread layout

will allow the maximum number of threads per SM to be used (1536), care must be taken to minimize the number of registers and amount of cache used per thread: if there are not enough hardware resources to simultaneously process 1536 threads, the compiler will force smaller, less optimal, thread blocks to be used. In order to maximize the number of simultaneous threads, the multiplication stage must make careful use of registers and cache.

7.3.2 Multiplication stage

A block diagram of the multiplication stage of the proposed algorithm is shown in Figure 7.1, and its associating pseudo-code is shown in algorithm 7.1. Before the algorithm begins execution, the multiplicand's words are loaded into cache memory (A_{cache}). Once this is complete, the algorithm begins its main loop shown on line 2, which loads the words of the multiplier (B) from GPU RAM into register %b_tmp. The GTX-480 does not possess an instruction that produces the 64-bit result of a 32-bit $\times$ 32-bit multiplication, however it does have instructions to separately generate the high-order and low-order 32-bits of the result.

These instructions are used across two passes, which are shown in the **for** loops on lines 3 and 12, which compute the high and low order result of the 32-bit $\times$ 32-bit multiplication, respectively; these two **for** loops correspond to the "lo" and "hi" blocks in Figure 7.1.

RAM accesses are minimized by storing the A-operand in cache, and register use is minimized by carefully recycling registers after each iteration of the outer loop on line 2. This is done by storing the lower words of the result to RAM when they are no longer needed (line 8, and recycling its register for the current highest word. The register recycling process carried out on lines 6 and 15 of algorithm 7.1, where the register that the result is saved to is chosen as $\%t_{i \bmod W}$ and $\%t_{i+1 \bmod W}$, respectively. In effect, this "slides" a window of registers from the right to the left

Algorithm 7.1: Proposed algorithm for the multiplication stage

input : W-word wide operand arrays A and B
output: Upper result of multiplication in registers $\%t_0$ to $\%t_{W-1}$; lower half of result in $C[0]$ to $C[W-1]$

```
// Note:  %x denotes a register
// Assume A operands have already been loaded to cache (A_cache)
```

1 **for** $i \leftarrow 0$ **to** $W-1$ **do**
2 $\quad \%b_{tmp} \leftarrow B[i]_{\text{RAM}}$
3 $\quad$ **for** $j \leftarrow 0$ **to** $W-1$ **do**
4 $\quad\quad \%a_{tmp} \leftarrow A[j]_{\text{cache}}$
5 $\quad\quad \%r \leftarrow$ mul.lo$(\%a_{\text{tmp}}, \%b_{\text{tmp}})$
6 $\quad\quad \%t_{i \bmod W} \leftarrow$ add_with_carry$(\%r, \%t_x)$
7 $\quad\quad$ **if** $j == 0$ **then**
8 $\quad\quad\quad C[i]_{\text{RAM}} \leftarrow \%t_{i \bmod W}$
9 $\quad\quad$ **end**
10 $\quad$ **end**
11 $\quad$ **if** $j \neq W-1$ **then**
12 $\quad\quad$ **for** $j \leftarrow 0$ **to** $W-1$ **do**
13 $\quad\quad\quad \%a_{tmp} \leftarrow A[j]_{\text{cache}}$
14 $\quad\quad\quad \%r \leftarrow$ mul.hi$(\%a_{\text{tmp}}, \%b_{\text{tmp}})$
15 $\quad\quad\quad \%t_{i+1 \bmod W} \leftarrow$ add_with_carry$(\%r, \%t_x)$
16 $\quad\quad$ **end**
17 $\quad$ **end**
18 **end**

as the multiplication algorithm proceeds through its iterations, as shown in the block diagram.

When the multiplication algorithm is complete, the result's upper W words will be stored in the $\%t$ registers, and the lower-order words will be saved to RAM in the C array. The multiplication stage has been very carefully designed to use the least amount of cache and registers possible. The total number of registers needed is $W+3$, and the number of cache storage locations is only W words. By efficiently using such a small number of registers, a very large number of multiplications are carried out simultaneously across many threads and many blocks, saturating the GPU's ALU resources. Specifically, with many threads working on the GPU simultaneously, cache and RAM accesses are more effectively hidden with the help of the GPU's deep

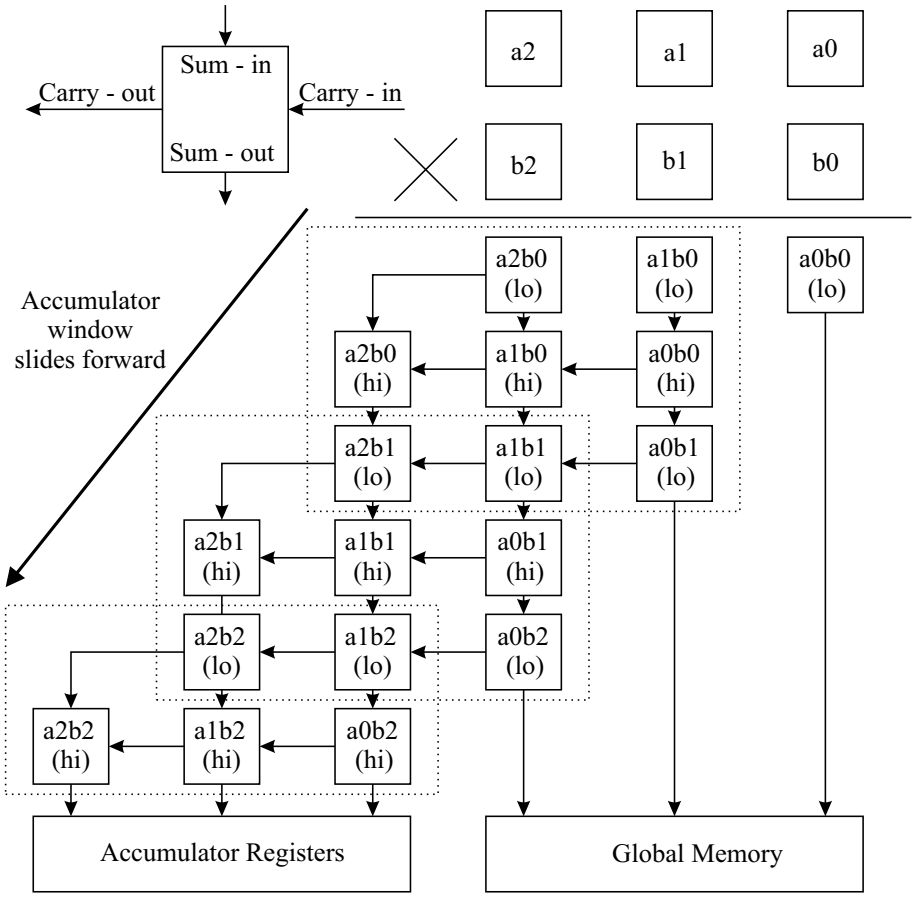


Figure 7.1: The proposed multiplication algorithm. Each thread serially processes this algorithm; generating the upper and lower bits of the partial products, accumulating them, and storing the results to memory as required. As results are stored to memory, the lower-bit accumulator registers are used for higher-bit partial product storage, as denoted by the sliding window.

pipelines.

7.3.3 Memory layout and access patterns

Each thread loads its operands serially, and saves interim results to accumulator registers and cache memory as it runs. RAM and cache access patterns were carefully designed to ensure contiguous accesses; as shown in Figure 7.2, operands are stored such that consecutive threads accessing the same limb of a W-word operand access consecutive locations. This is as opposed to having limbs from a single thread occupy consecutive memory locations, which will give rise to memory bank conflicts and

serialized memory accesses [6].

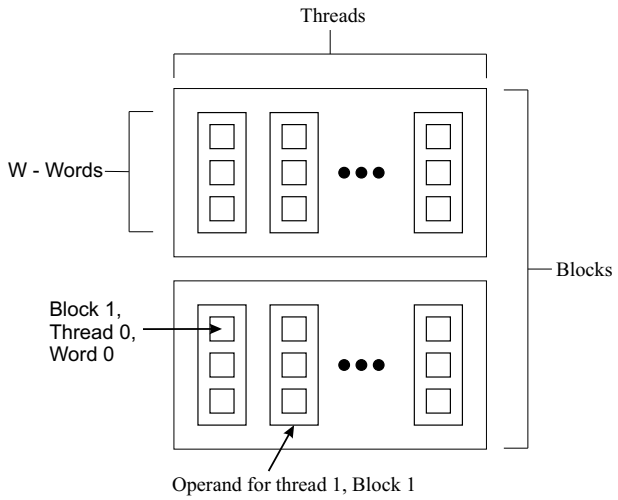


Figure 7.2: Memory configuration: consecutive threads loading the same limb of the multi-word operands access consecutive memory locations.

7.3.4 Reduction stage

Once the 2W-wide result of the multiplication is available in the accumulator registers and GPU RAM, the reduction operation begins. The typical method for NIST prime reduction outlined in [34] is used. This approach takes advantage of the fact that NIST primes are expressed as the sum of a small number of powers of two to quickly reduce the double-wide result without the need for any multiplication operations. For example, the reduction algorithm for the NIST prime $p_{192} = 2^{192} - 2^{64} - 1$ is shown in algorithm 7.2; note that this is a reproduction from [34].

Algorithm 7.2: NIST reduction for p_{192} [35]

input : An integer $c = (c_5, c_4, c_3, c_2, c_1, c_0)$ in base 2^{64} with $0 \leq c < p_{192}^2$
output: $c \bmod p_{192}$

1 Define 192-bit integers:
2 $s_1 = (c_2, c_1, c_0), s_2 = (0, c_3, c_3), s_3 = (c_4, c_4, 0), s_4 = (c_5, c_5, c_5)$
3 Return $(s_1 + s_2 + s_3 + s_4 \bmod p_{192})$

All of the NIST reduction algorithms only use addition, subtraction, and shifting, and can be computed in relatively few steps. The reduction algorithm for each of

the NIST primes is different, however, which requires them to be written individually and by-hand.

7.4 Implementation details

It was imperative that the proposed NIST prime field multiplication algorithm be implemented using NVIDIA's PTX low-level virtual machine language for several reasons. First, the C for CUDA API does not provide access to a number of the GPU's intrinsic instructions required for the proposed algorithm, such as add-with-carry, and multiply.hi / multiply.lo. Although it would be possible to simply in-line these instructions as needed, this approach does not give a strong measure of control over the device's register use. Furthermore, as learned in Chapter 6, the nvcc compiler does not intelligently map registers from its single-static-assignment intermediate code to machine code when unrolling a large amount of code, which is the case here.

The result of these constraints required that the proposed algorithm be written completely in the PTX language. In order to assist in this process, a tool in the C language was developed to automatically generate the the required code for the multiplication part of the proposed algorithm, while the reduction operation was written by hand for the different field sizes.

7.5 Verification and testing

The proposed algorithm was confirmed to be functioning correctly by using the popular 'GMP' (The GNU Multiple Precision Arithmetic Library) [73] software package to generate sets of approximately 10 million random input operands, which were copied to and processed by the GPU. Once complete, the data was copied back from the GPU to the host, and results were compared against a set of 'golden' results generated by passing the same operands through the GMP modular multiplication function. Every

multiplication result was fully verified against the results from the GMP package.

NVIDIA's 'Compute Visual Profiler' tool was used to determine the kernel's runtime; this value was then divided by the total number of multiplications carried out during that interval to arrive at an average time per multiplication.

7.6 Results and Comparison

Shown in Table 7.2 are the different instruction types, and the number of times each thread executed them to perform a single multiplication operation. These data were obtained by using the NVIDIA CUDA object dump tool to extract the machine instructions executed by the GPU. As shown in the table, the number of integer addition and multiplication instructions increases significantly as the field size increases, while the number of global memory accesses are kept to a minimum. Cache memory accesses increase as well; this is certainly desirable compared to a greater number of RAM transactions.

Table 7.2: Operation counts for the proposed algorithm

Field Size	Add. and Sub.	Mult.	Cache Mem.		RAM Mem.	
			Loads	Stores	Loads	Stores
192	88	61	76	22	18	12
224	135	85	99	22	21	14
256	225	113	172	68	24	16
384	425	265	358	106	36	24

Finite field multiplication operation throughput results for several algorithms in the literature are shown in Table 7.3. Values shown are measured operation throughput in millions of multiplications per second. In the second column from the left are results from the Intel "Integrated Performance Primitives" functions, running on a 3.0 GHz Intel q9650 CPU; these values are used as a performance baseline. Note that the simple test program which performed these measurements was created for this work.

In the third column from the left the FIOS Montgomery multiplication algorithm from Giorgi et al. , which attempts to fit all operands in registers. Facilitating this approach is the fact that the older 9800 GX2 GPU used for their work allowed a large number of registers per thread: a maximum of 128 vs. 63 for the GTX-480 used for the proposed algorithm. It is important to remark that the Montgomery multiplication algorithm employed by these authors will allow any prime field to be used, whereas the proposed is limited to fields over the NIST polynomials, where reduction is especially easy to compute.

Code for Henry and Goldberg's work was made publicly available, and it was possible to run their work on the GTX-480 GPU used for the proposed algorithm, in order to obtain a direct and fair comparison. Results of their work are shown in the second column from the right in Table 7.3. Their work is quite recent, and was independently carried out at the same time as the work is this dissertation.

Interestingly, both the proposed algorithm and Henry & Goldberg's utilize custom PTX code for the multiplication function in order to bypass the nvcc compiler's limitations. Differing from the work of Giorgi et al., Henry and Goldberg chose to implement a CIOS Montgomery multiplication algorithm, which, again, enables this design to be used for any prime field multiplication. One limitation of this work is that is shared with the proposed is that the field prime must be fixed for all threads.

Compared to the next best GPU-based finite field multiplication algorithm, the proposed work boasts 1.4 to 1.6 times greater operation throughput, while it is 58.6 to 93.0 times faster than the CPU implementation running Intel's algorithm. To the best of the author's knowledge, the proposed algorithm has the highest NIST field operation throughput in the literature.

Table 7.3: Comparison of operation throughput for different algorithms and field sizes in 10^6mults./s

Field Size Size	Intel IPP [74]	Giorgi[68]	Henry[72]	Proposed	Improvement vs. Henry
192	19	30	1000	1471	1.47×
224	13	18	781	1235	1.58×
256	14	12	625	847	1.36×
384	8	4	301	478	1.59×

7.7 Summary

A new algorithm for computing modular multiplication over the NIST prime fields on a GPU was presented. The algorithm was implemented entirely in PTX assembly, and validated against the results generated by a popular multi-precision software package. Compared to the next-fastest GPU-based algorithm, the approached proposed here boasts between 1.36 and 1.59 times higher operation throughput, depending on the field size. Compared to the Intel IPP running on a CPU, the proposed algorithm has 58.6 to 93.0 times higher operation throughput. To the best of the author's knowledge, the proposed algorithm boasts the highest NIST prime field multiplication operation throughput in the literature.

CHAPTER 8

A Complete Prime Field Arithmetic Library for the GPU

8.1 Introduction

In this chapter, a complete finite field arithmetic library for the GPU is developed. The proposed library includes addition, subtraction, multiplication, multiplication by a constant, squaring, and inversion.

The NIST-prime field multiplication algorithm presented in Chapter 7 is replaced with a Montgomery multiplication algorithm that allows any prime field to be used (rather than the special NIST primes), while improving performance. The proposed Montgomery multiplication algorithm outperforms the state-of-the-art CPU and GPU implementations in the literature, and its performance is comparable to the fastest reported FPGA design. If operation throughput per dollar is considered as a metric, the proposed design has the highest performance compared to any other device.

The resulting finite field arithmetic library is suitable for carrying out elliptic curve scalar point multiplication.

The rest of this chapter is organized as follows: Section 8.2 discusses improvements to the multiplication stage, and section 8.3 discusses reduction techniques. In section 8.4, a GPU-based Montgomery multiplication algorithm is proposed and analyzed. Following that, section 8.5 presents the addition, subtraction, and squaring algorithms used in the proposed finite field library, and section 8.6 proposes a GPU-based inversion algorithm. Finally, section 8.7 presents the operation throughput performance results of the proposed library and compares to the state of the art, while section 8.8 summarizes the chapter.

8.2 Improving multiplication

The major goal of this section is to improve the prime field multiplication operation throughput. Using the NIST-field multiplication algorithm presented in Chapter 7 as a starting point, it is possible to analyze the number of arithmetic instructions as well as RAM and cache transactions per multiplication in order to determine the performance bottleneck; these are shown in Table 8.1.

Table 8.1: Compute time, ALU Instruction count, RAM, and cache transactions per multiplication, the NIST prime field multiplication algorithm

Field Size	Average (ns/mult.)	RAM bytes/mult.	Cache bytes/mult.	ALU Instr./mult.
192	0.68	120	392	210
224	0.81	140	484	305
256	1.18	160	960	451
384	2.09	240	1856	955

In order to determine the bottleneck, the obtained instruction throughput for memory accesses and computations must be compared to the maximum theoretical values. The maximum RAM bandwidth for the GTX-480 has been published by NVIDIA, and it is 177.4×10^9 bytes/s. Note that 177.4×10^9 differs from 177.4 GB, which is 177.4×2^{30} bytes. Considering a single addition instruction as a unit

of computation, the GTX-480 can perform $1400 \times 10^6 \frac{\text{cycles}}{s} \times 32 \frac{\text{operations}}{\text{SM}} \times 15 \frac{\text{SMs}}{\text{GPU}} = 672 \times 10^9$ operations per second. Finally, the L1 cache bandwidth can be calculated as $4 \frac{\text{bytes}}{\text{bank}} \times 7 \cdot 10^8 \frac{\text{cycles}}{s} \times 32 \frac{\text{banks}}{SM} \times 15 \frac{SM}{GPU} = 1.344 \times 10^{12}$ bytes per second [6]. With these quantities known, the fraction of theoretical RAM, Cache, and computation throughput used by the NIST prime field multiplication algorithm have been determined, as shown in Table 8.2.

Table 8.2: RAM, cache, and operation throughput for NIST field multiplication

Field Size	Operation Throughput			Percent of Maximum (%)		
	RAM 10^9 bytes/s	Cache 10^9 bytes/s	Compute 10^9 Instr./s	RAM	Cache	ALU
192	177	576	309	99.4	42.9	45.6
224	173	598	377	97.4	44.6	56.0
256	136	814	382	76.4	60.5	56.9
384	115	888	457	64.7	66.0	68.0

As indicated in the third column from the right in Table 8.2, the performance of smaller field sizes is completely bound by RAM bandwidth, and it is impossible to further improve performance without reducing the number of RAM transactions per operation.

For larger field sizes, the limiting factor is the burst memory-access pattern employed by the algorithm: although the average RAM bandwidth is not completely saturated, the majority of the memory accesses occur all at once, causing stalls. Thus, for all field sizes, improving multiplication will require reducing the number of RAM accesses, improving the temporal access pattern, or both. The most straight-forward solution is to use fewer active threads per SM, which would allow more registers and cache per thread, reducing RAM accesses. Taken too far, however, there will not be enough threads to fill the SM pipeline, causing stalls. It is now clear that there is an ideal number of threads per SM vs. resources per thread, finding it will yield the most efficient multiplication algorithm

8.2.1 Threads per SM vs. resources per thread

The multiplication stage of the NIST algorithm was designed to maximize the number of threads per SM, and had to be re-written to allow the threads per SM, registers per thread, and cache per thread to be adjustable, compile-time parameters. Following this, an experiment was conducted where the number of threads per SM was varied from 32 to 1536 in increments of 32 (one warp), and the multiplication algorithm recompiled each time to make optimal use of the available register and cache resources per thread. For comparison, a second multiplication algorithm was created without using any cache at all, in order to study the effect of the user-controlled cache. This experiment repeated for operand sizes of 112, 128, 160, 192, 224, 256, 320, 384, 512, and 521 bits, to correspond with the field sizes of interest in elliptic curve cryptography.

Shown in Figure 8.1 is an example that the effect of varying threads per SM vs. resources per thread has on the 384-bit multiplication algorithm's operation throughput. Before the analysis proceeds, an important remark is that in the topmost plot, it appears that the memory use is exceeding 100% for the interval between 1312 and 1536. After some experimentation, it was determined that this was caused by the RAM's L2 cache, which could in certain cases boost the effective RAM throughput by up to 10%.

Proceeding from left-to-right, it is shown that when a very small number of threads per SM is used (32-64), the SM pipeline is not full and stalls occur, which limits operation throughput.

As the number of threads is increased gradually, operation throughput, RAM transactions, and ALU operations all increase dramatically, achieving peak performance between as few as 96 to 384 threads per SM. Note that at 384 threads per SM, there is virtually no difference in performance between the multiplication algorithm which uses cache and the one that does not. This is because there are still enough

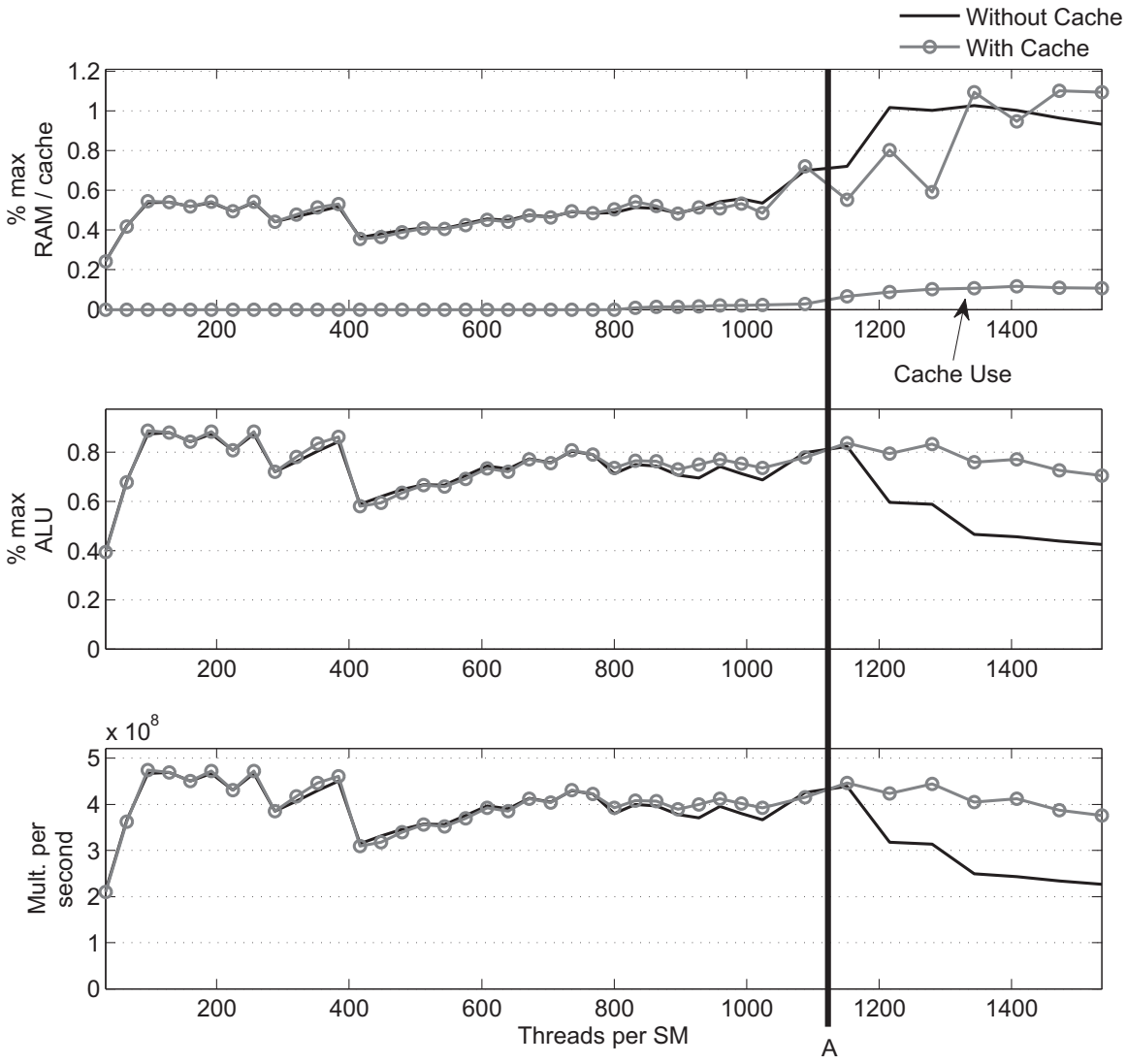


Figure 8.1: Memory transaction, ALU utilization, and multiplication operation throughput vs. threads per SM for 384-bit multiplication

registers available to house the operand and accumulator words, and cache is not actually being used. This remains the case until 1152 threads (marked point A on the Figure), where the multiplication algorithm without cache must resort to storing its operands in RAM, and the algorithm with cache uses that instead, as shown in the top-most plot of Figure 8.1. Multiplication operation throughput for the cache-less algorithm drops dramatically, while the cache-algorithm remains steady.

As demonstrated, integrated cache into the multiplication algorithm allows a greater number of threads per SM to be used, however peak performance may actually

occur with far fewer threads per SM.

Finally, it should be noted that the point A where cache begins to be used in Figure 8.1 is different as the multiplication algorithm's operand size changes. In the case of larger field sizes, A may occur earlier on, with fewer threads. For smaller operands, cache may not ever need to be used, because the operands easily fit in registers. In either case, this raises the question: what role does cache play in terms of peak operation throughput across all thread configurations, and is it worth the additional programming difficulty?

8.2.2 Cache vs. cache-less multiplication algorithms

In order to determine the utility of integrating cache with the multiplication algorithm, the peak multiplication operation throughput for different operand sizes was measured and tabulated for cache and cache-less multiplication. Shown in Table 8.3 are the complete results of the experiment: the multiplication algorithms' maximum operation throughput for each operand size were measured. The rightmost column of Table 8.3 shows the percentage improvement of using the cached vs. cache-less algorithms; in all cases the improvement is marginal. Given the added difficulty of integrating cache management into the multiplication algorithm, a 2% to 5% increase is not a significant enough of a performance gain to warrant this approach.

8.2.3 Asymptotically fast multiplication

The multiplication algorithm employed thus far has been the standard schoolbook method, where the number of multiplication instructions scaling with complexity $\mathcal{O}(m^2)$. Other multiplication methods exist with sub-$\mathcal{O}(m^2)$ complexity, such as Karatsuba-Ofman multiplication [75] with complexity $\mathcal{O}(m^{\log_2 3})$, and Schönhage-Strassen multiplication [76] with complexity $\mathcal{O}(m \log(m) \log(\log(m)))$. These algorithms are referred to as asymptotically fast multiplication, because they possess

Table 8.3: Multiplication cost in ns, with and without cache management

Operand Size	Multiplication	Multiplication w/ cache	Improvement (%)
112	0.52	0.50	3.8
128	0.54	0.52	3.7
160	0.66	0.63	4.5
192	0.77	0.75	2.6
224	0.90	0.87	3.3
256	1.03	1.01	1.9
320	1.44	1.40	2.8
384	2.10	2.05	2.4
512	4.19	4.11	1.9
521	4.88	4.65	4.7

significant overhead cost, and only become useful for very large operand sizes.

Karatsuba multiplication has the least overhead, and its suitability for GPU implementation is examined first; it is carried out as follows:
For $m = 2l$, $A = a_1 2^l + a_0$, and $B = b_1 2^l + b_0$ the product $A \cdot B$ can be written as:

$$A \cdot B = (a_1 2^l + a_0)(b_1 2^l + b_0) \tag{8.1}$$

$$= a_1 2^l b_1 2^l + a_0 b_1 2^l + a_1 b_0 2^l + a_0 b_0 \tag{8.2}$$

$$= a_1 b_1 2^{2l} + (a_0 b_1 + a_1 b_0) 2^l + a_0 b_0 \tag{8.3}$$

$$= a_1 b_1 2^{2l} + [(a_0 + a_1)(b_0 + b_1) - a_1 b_1 - a_0 b_0] 2^l + a_0 b_0 \tag{8.4}$$

Using this divide and conquer strategy replaces an m-wide multiplication with three $\frac{m}{2}$-bit multiplications ($a_1 b_1$, $(a_0 + a_1)(b_0 + b_1)$, and $a_0 b_0$), two $\frac{m}{2}$-bit additions, and a total of three m-bit wide addition and two m-bit wide subtractions. This algorithm can be applied recursively to each of $a_0, a_1, b_0, b_1, (a_0 + a_1)$, and $(b_0 + b_1)$, however this will require additional storage for intermediate results.

It is of interest to determine if the advantages afforded by Karatsuba multiplication outweigh its overhead on the GTX-480 GPU for operand sizes that are of interest in elliptic curve cryptography. Beginning with one of the largest field size of interest, 512

bits, the Karatsuba multiplication cost can be estimated by examining the 256-bit addition, subtraction, and multiplication costs, as shown in Table 8.4.

Table 8.4: Multi-word addition, subtraction, and multiplication costs in ns

Field Size	Addition	Subtraction	Multiplication
112	0.45	0.44	0.52
128	0.44	0.44	0.54
160	0.52	0.52	0.66
192	0.60	0.60	0.77
224	0.69	0.68	0.90
256	0.77	0.77	1.03
320	0.93	0.92	1.44
384	1.09	1.06	2.10
512	1.41	1.41	4.19
521	1.50	1.51	4.88

The minimum cost of a Karatsuba step (which assumes no additional overhead stemming from memory storage and address pointer calculation) is three 256-bit multiplications, two 256-bit additions, three 512-bit additions, and three 512-bit subtractions: $3 \cdot 1.03 + 2 \cdot 0.77 + 3 \cdot 1.41 + 2 \cdot 1.41 = 12.45ns$ which is significantly greater than a single 512-bit multiplication, which costs $4.88ns$.

Given this result, it can be concluded that Karatsuba multiplication, and asymptotically fast algorithms with even greater overhead, are not well suited for the operand sizes needed while using this GPU architecture.

8.3 Improving reduction

As shown in Chapter 7, the NIST fields allow very efficient reduction, requiring only multi-precision addition and subtraction operations. This imposes some significant limitations: it is unlikely that a randomly chosen prime number will allow for such an efficient reduction, and indeed certain standards such as the Brainpool curves do not possess this property [40]. Additionally, hand-coding is required for each of the NIST fields, which may increase the cost of maintaining and testing code. This

section presents work which enables arbitrary field primes to be used for the proposed GPU-based prime field multiplication algorithm.

8.3.1 Montgomery reduction

Montgomery reduction [77] is the finite field reduction algorithm published in 1985 by Montgomery which is employed almost ubiquitously in both hardware and software implementations; the significance of Montgomery reduction in public key cryptography is likened to the importance of the fast Fourier transform in digital signal processing.

Review of Montgomery reduction

In essence, Montgomery reduction is an efficient way of computing $TR^{-1} \bmod p$. The key step is shown on line 6 of algorithm 8.1 [78], where division by b^W is carried out. With b set to the machine word size, 2^{32} in this case, division is performed as a 32-bit right-shift operation, which is *much* less expensive than a trial division.

Algorithm 8.1: The Montgomery reduction algorithm [77]

input : integers $p = (p_{W-1}, \dots p_1, p_0)_b$, with $\gcd(p, b) = 1$, $R = b^W$, $p' = -p^{-1} \bmod b$, and $T = (t_{2W-1}, \dots, t_1, t_0)_b < pR$
output: $TR^{-1} \bmod p$

1 $A \leftarrow T$, with $A = (a_{2W-1}, \dots, a_1, a_0)$
2 **for** $i \leftarrow 0$ **to** $W - 1$ **do**
3 $\quad u_i \leftarrow a_i p' \bmod b$
4 $\quad A \leftarrow A + u_i p b^i$
5 **end**
6 $A \leftarrow A/b^W$
7 **if** $A \geq p$ **then** $A \leftarrow A - p$
8 **return** A

It is possible to compute $c = ab \bmod p$ for using Montgomery reduction as follows: First, convert operands a and b to Montgomery form, $\bar{a} = aR \bmod p$ and $\bar{b} = bR \bmod p$, and T can be computed as $T = \bar{a} \times \bar{b} = abR^2$. This can be followed with a

Montgomery reduction step, producing: $TR^{-1} \bmod p = abR^2R^{-1} \bmod p = \bar{a}\bar{b}R \bmod p$ With an efficient way to determine $TR^{-1} \bmod p$, it is possible to $TR^{-1} \bmod p$, which is equivalent to $\bar{c} = \bar{a}\bar{b} = cR \bmod p$. Finally, $\bar{c}$ must be converted back to non-Montgomery form by finding $c = \bar{c}R^{-1} \bmod p$.

Conversion to Montgomery form can be performed by using Montgomery reduction with $T = aR^2$, and conversion to original form can be performed using Montgomery reduction with $T = \bar{a}$.

8.3.2 Other reduction techniques

Barrett reduction [79], and modified Barrett reduction [80] are other techniques used for modular reduction. Although their use was considered, Barrett reduction requires more multiplication steps than Montgomery [81], and the modified Barrett incorporates Karatsuba multiplication into the algorithm, which has already been shown to be more expensive than schoolbook multiplication in section 8.2.3.

8.4 Montgomery multiplication

The Montgomery reduction algorithm can be integrated directly with multiplication, to allow for $\bar{a}\bar{b}R^{-1} \bmod p$ to be directly computed for two operands $\bar{a}$ and $\bar{b}$.

A variety of implementations exist [82], which employ different strategies for performing the multiplication; for example, it is possible to separately compute the multiplication, and then perform the reduction (serparted operand scanning, or SOS), or alternatively reduction can follow after every iteration of the multiplication's inner loop, a technique known as coarsely integrated operand scanning (CIOS).

For the proposed GPU-based multiplication algorithm, it would be advantageous to use the fewest registers possible, rather than require more RAM accesses which were shown to form the bottleneck of the NIST finite field multiplication algorithm

presented in Chapter 7. Of the techniques surveyed by Koç et al., the CIOS method requires the least amount of temporary storage with only $W + 2$ memory locations.

8.4.1 Proposed implementation for the Montgomery multiplication algorithm

The proposed Montgomery multiplication algorithm inherits the memory and compute grid layouts from the NIST-fields algorithm proposed in Chapter 7. Also similar to the NIST algorithm, the algorithm Montgomery multiplication algorithm is implemented in NVIDIA's PTX low-level virtual machine language. Rather than code by hand, the PTX code is generated using parameterized C code, which allows the 'active threads vs. resource per thread' space to be easily explored as described in section 8.2.1.

The proposed GPU-implementation of the Montgomery multiplication algorithm closely follows the CIOS algorithm presented by Koç et al. [82], whose pseudocode is reproduced below:

Listing 8.1: CIOS implementation of the Montgomery multiplication algorithm

```
1
2  for i=0 to W-1
3  {
4    C = 0
5
6    //Multiplication loop
7    for j=0 to W-1
8    {
9      (C,S) = t[j] + a[j]*b[i] + C
10     t[j] = S
11   }
12   (C,S) = t[W] + C
13   t[W] = S
14   t[W+1] = C
15
16   C = 0
17   m = t[0]*p'[0] mod W
18   (C,S) = t[0] + m*p[0]
```

```
19
20    //Reduction loop
21    for j=1 to W-1
22    {
23      (C,S) = t[j] + m*p[j] + C
24      t[j-1] = S
25    }
26    (C,S) = t[W] + C
27    t[W-1] = S
28    t[W] = t[W+1] + C
29  }
```

A number of implementation-specific optimizations have been carried out to improve the proposed algorithm's performance. First, all loops have been completely unrolled. While this increases the code size somewhat, it eliminates a large amount of computation. For example, rather than store the field prime in RAM it embedded in the program in the form of immediate operands, eliminating the retrieval of $p'[0]$, $p[0]$, and $p[j]$ on lines 17, 18, and 23 in code listing 8.1. Another major optimization is that where possible, separate multiplication and addition operations have been replaced with a combined multiply-accumulate instructions.

Results and comparison to NIST multiplication

Shown in Table 8.5 is a comparison of the NIST multiplication algorithm from Chapter 7, and the Montgomery multiplication algorithm proposed here.

For the Montgomery multiplication algorithm, a number of runs were performed, varying the number of threads per SM from 32 to 1536 in increments of 32, as shown in section 8.2.1 The cuobjdump binary dump tool was used to obtain an instruction count, and determine the memory and ALU utilization, and the NVIDIA CUDA API's function calls were used to obtain timing information for each run. Note that as with the NIST multiplication data, only the multiplication operation is timed, and does not include host-to-GPU RAM transfers. This is acceptable because the intended application, elliptic curve scalar point multiplication, this initial host-to-GPU transfer

cost will be amortized over a very large number of finite field multiplications. Each run consisted of multiplying together several million results, and validating them against the GMP modular multiplication functions [73]. Each of the data in Table 8.5 are from the run with the highest measured operation throughput.

Table 8.5: Comparison of proposed GPU-based NIST and Montgomery multiplication algorithms

Field Size	% Max. Memory Use			% Max. ALU		10^6 mult. / s	
	NIST Cache	NIST RAM	Mont. RAM	NIST	Mont.	NIST	Mont.
192	42.9	99.4	77.0	45.6	80.5	1471	1563
224	56.0	97.4	69.4	56.0	87.6	1235	1282
256	56.9	76.4	62.8	56.9	87.8	847	1031
384	68.0	64.7	44.9	68.0	96.0	478	510

Compared side-by-side in Table 8.5 are the memory and ALU use, as well as the multipliers' bulk operation throughput rates in 10^6 multiplications per second.

The situation is an improvement rather than a trade-off: the unused ALU capacity from the NIST algorithm has been applied to the Montgomery algorithm's more complicated reduction stage, which allows finite field multiplication with *any* prime, rather than those in the special NIST forms. This conclusion is further supported by the operation throughput of both multiplication algorithms, which actually favors the proposed Montgomery algorithm.

Moving forward, the remaining finite field operations used in this dissertation are based on the GPU-based CIOS Montgomery multiplication algorithm proposed here.

8.5 Finite field addition, subtraction, and squaring

With a strong finite field multiplication algorithm selected, the remaining finite field operations must be developed in order to perform elliptic curve scalar point multiplication.

8.5.1 Addition and subtraction

Finite field addition and subtraction are implemented using simple addition and carry chains. As with the proposed multiplication algorithm, each thread performs a single operation. Straightforward add/sub-with-carry instructions are used, the field prime is subtracted (or added) to the result as needed. While this introduces thread divergence, it is negligible. Results of the addition and subtraction algorithms are shown later in section 8.7.

8.5.2 Squaring

The proposed finite field squaring algorithm for the GPU is based on the Montgomery multiplication algorithm described in section 8.4. It has been modified such that operands a and b are the same, reducing the number of memory accesses that are required from a minimum of $3W$ words to $2W$. Results of the proposed squaring algorithm are shown in section 8.7.

8.6 Modular inversion

Finite field inversion over prime fields is an operation so computationally expensive that it is usually avoided if possible. For example, a common heuristic for estimating the cost of finite field inversion is to multiply the cost of finite field multiplication by 80 or even 100.

This section will show that the extended Euclidean algorithm that is commonly used to carry out inversion for prime fields is particularly ill-suited for GPU implementation. In its place, a normally more costly approach based on Fermat's little theorem is employed. Although the proposed inversion algorithm is still quite costly compared to multiplication, it is competitive with state-of-the-art CPU implementations.

8.6.1 The binary inversion algorithm

The binary inversion algorithm (8.2), is based on the extended Euclidean algorithm, which is for finding the greatest common divisor of two integers. Note that this algorithm employs an abundance of branch operations: the algorithm's main **while** loop on line 3 depends on the values u and v to control its continued execution. Meanwhile, there are two nested **while** loops contained inside the main on lines 4 and 9, and each of these contain **if** statements, inducing further branching.

While none of these branch constructs present an issue for serial processors, this is a major stumbling block for the GPU implementation: since threads are processed in batches (warps) of 32, if a single thread diverges the remaining ones are stalled. As shown in algorithm 8.2, the branching and **while** loops all depend on the operands, and if 32 are processed at a time, there will almost certainly be a large number of divergent threads, which is further compounded by the multiple levels of branching present in this algorithm.

A GPU implementation was attempted, and results were extremely poor; *the cost of a single inversion operation was over 10,000 times greater than a multiplication.*

With such a high expense, it would be far more economical to perform inversion on a CPU than the GPU. Luckily, another inversion technique exists that, while still expensive, is far more amenable to GPU implementations.

8.6.2 Proposed GPU finite field inversion algorithm

A theorem known as "Fermat's little theorem" is shown in Equation 8.5, for p prime, and $a \in [1, p-1]$:

$$a^p \equiv a \bmod p \tag{8.5}$$

Algorithm 8.2: The binary inversion algorithm for fields $\mathbb{F}_p$ [34]

input : Prime p and $a \in [1, p-1]$
output: $a^{-1} \bmod p$

```
1  u ← a, v ← p
2  x1 ← 1, x2 ← 0
3  while u ≠ 1 and v ≠ 1 do
4  |   while u is even do
5  |   |   u ← u/2
6  |   |   if x1 is even then x1 ← x1/2
7  |   |   else  x1 ← (x1 + p)/2
8  |   end
9  |   while v is even do
10 |   |   v ← v/2
11 |   |   if x2 is even then x2 ← x2/2
12 |   |   else  x2 ← (x2 + p)/2
13 |   end
14 |   if u ≥ v then u ← u − v, x1 ← x1 − x2
15 |   else v ← v − u, x2 ← x2 − x1
16 end
17 if u = 1 then return x1 mod p
18 else  return x2 mod p
```

Thus,

$$a^{p-1} \equiv 1 \bmod p \tag{8.6}$$

and

$$a^{p-2} \equiv a^{-1} \bmod p. \tag{8.7}$$

Using Equation 8.7, it is possible to compute a^{p-2} by using a chain of multiplication and squaring operations as shown in algorithm 8.3.

For GPU implementations, the most important aspect of inverting a field element with the Fermat method is that all threads perform the same execution path. The only conditional statement in algorithm 8.3 is on line 3, and it depends on the field prime, which is common across all threads.

Algorithm 8.3: Proposed inversion algorithm based on Fermat's little theorem

input : m-bit wide field prime p, operand $a \in [0, p-1]$
output: $a^{-1} \bmod p$

```
1 q ← p − 2; x ← a
2 for i ← 0 to m − 1 do
3 |   if q_i = 1 then
4 |   |   x ← x × a
5 |   end
6 |   x ← x^2
7 end
8 return x
```

Inversion cost

The proposed approach uses a naïve algorithm with a total cost of $H(p-2)M + mS$, where $H(p-2)$ is the Hamming weight of the prime $p-2$, m is the number of bits in p, and M & S are the costs of multiplication and squaring, respectively. With inversion cost now dependent on the specific field prime, it is important to remark that field primes with a large Hamming weight may have at most twice the cost of a field prime with a small Hamming weight. For example with the proposed algorithm, the most expensive field primes are Mersenne primes, which are primes of the form $2^m - 1$ for some m, such as the case for the NIST prime $p_{521} = 2^{521} - 1$.

It may be possible to improve the inversion algorithm by finding an efficient addition chain that minimizes the number of multiplications required in algorithm 8.3 by using some intermediate storage, however such improvements are beyond the scope of this dissertation, and are left as future work.

Implementation details

Some implementation-level optimizations are possible for this approach. For the proposed inversion algorithm, multiplication and squaring were integrated into a single, large, function. While increasing the number of registers needed to run the algorithm, a large amount of function-call overhead was eliminated. Eliminating function

calls to squaring and multiplication subroutines also reduced the number of memory transactions that would have been required; in the CUDA PTX language the register space is in the function scope, and passing variables to functions can only be done through the RAM memory.

The proposed inversion algorithm does *not* unroll the main **for** loop, as this would grow the resulting binary to several megabytes in size, while placing a great burden on the nvcc compiler which was designed for much lighter-weight programs.

Finally, the value $q = p - 2$ is embedded into the program at compile time by using a lookup table. This is done for two reasons: first, this reduces the number of registers needed, as q is now stored in program instructions. Second, the lookup table is small, and the program instructions are cached, ensuring fast accesses compared to storing the table in RAM.

Results for the proposed inversion algorithm are shown in Table 8.6 in the next section.

8.7 Results and comparison

Shown in Table 8.6 are the results of the proposed finite field arithmetic library for GPU. They are shown in ns / operation. Results were collected using the same procedures outlined in 8.2.1 and 8.4.1.

The least expensive operations for small field sizes are squaring and multiplication by a constant. At first this may seem counter-intuitive: addition and subtraction are almost always considered less costly than squaring. The reason squaring and multiplication by a constant are less expensive is due to the fact that these are unary operands, and and require only $\frac{2}{3}$ of the memory transactions compared to addition and subtraction. This effect dominates the operation costs for small field sizes. As the operand size increases, addition and subtraction are approximately half the cost

Table 8.6: Finite field arithmetic library operation costs in ns

Field Size	Add.	Sub.	Mult.	Sqr.	Mult. by Constant	Inv. (Max)	Inv. (Min)
112	0.41	0.41	0.42	0.35	0.34	24.55	47.11
128	0.41	0.41	0.42	0.34	0.34	28.20	54.20
160	0.49	0.48	0.52	0.47	0.45	53.56	103.76
192	0.57	0.56	0.64	0.61	0.59	89.88	175.04
224	0.63	0.64	0.78	0.78	0.75	143.51	278.82
256	0.71	0.71	0.97	0.96	0.92	211.00	408.96
320	0.86	0.85	1.42	1.41	1.36	403.41	794.24
384	1.02	1.02	1.96	1.95	1.89	697.91	1360.01
512	1.32	1.35	3.40	3.35	3.24	1624.99	3202.51
521	1.41	1.42	3.77	3.75	3.64	3634.61	7070.88

of multiplication and squaring.

Shown in the two rightmost columns are the costs for inversion with very high and very low Hamming weight primes; as expected, heavy-weight primes have roughly twice the cost of the low-weight ones. In either case, the cost of inversion is very high: 72 to roughly 2000 times the cost of multiplication depending on the field size and the prime's Hamming weight.

Unfortunately, aside from multiplication, there are a lack of published results in the literature for other finite field operations. As such, the comparison of the proposed work to the state-of-the-art will be limited to the GPU-based Montgomery multiplication algorithm; the results of are shown in tables 8.7 and 8.8.

Table 8.7 compares the proposed multiplication algorithm to various CPU and GPU implementations. In the second column from the left are the results from Giorgi et al., which were thoroughly discussed in Chapter 7, section 7.6. In the third column from the left is a software implementation running an a 2.5 GHz Intel Xeon E5420 CPU from Chow et al. The fourth column from the left presents results from Neves and Araujo, which was also discussed in Chapter 7. In their work, they implemented the FIOS, CIOS, and FIPS versions of the Montgomery multiplication algorithm on the NVIDIA GTX-260 GPU, and found that the FIOS version worked best in their

case, which is the result presented here.

Also previously described are the results from Henry and Goldberg, which are the next best implementation for CPU and GPU. Their work was performed independently, and at the same time as the proposed, and they have developed a very similar system which also uses PTX code, Montgomery CIOS multiplication, and loop unrolling. The proposed algorithm is able to outperform their work by 1.24 to 1.72x depending on the field size; this is due to the additional optimizations described in sections 8.2.1 and 8.4.1, such as performing an exhaustive search to determine the optimal number of threads per SM vs. registers per thread, and embedding the field prime into the GPU kernel.

Table 8.7: Comparison of proposed GPU-based finite field multiplication algorithm to the state of the art GPU and CPU implementations, results in 10^6 multiplications per second

Field Size	Giorgi [68]	Chow [83]	Neves [71]	Henry [72]	Proposed	Improvement vs. Henry
112				1,923	2,381	1.24 ×
128		34		1,923	2,381	1.24 ×
160	45			1,351	1,923	1.42 ×
192	30			1,000	1,563	1.56 ×
224	18			781	1,282	1.64 ×
256	12	18		625	1,031	1.65 ×
320				420	704	1.68 ×
384	4			301	510	1.69 ×
512		12	21	175	294	1.68 ×
521				155	265	1.72 ×

Table 8.8: Multiplication operation throughput and multiplication operation throughput per dollar comparison of proposed GPU-based finite field multiplication algorithm to the state of the art FPGA implementation

Field Size	10^3 Mults. per second			10^3 Mults. per second per dollar		
	Chow [83]	Proposed	Improvement	Chow [83]	Proposed	Improvement
128	6,400,000	2,380,952	0.37 ×	655	9,524	14.55 ×
256	1,900,000	1,030,928	0.54 ×	194	4,124	21.21 ×
512	400,000	294,118	0.74 ×	41	1,176	28.75 ×

In Table 8.8, the proposed work is compared to the state-of-the-art FPGA implementation. In this work, Chow et al. have designed a Karatsuba multiplier for a Xilinx Virtex 6 XC6VSX475T-2 FPGA running at 400 MHz. Worth noting here, is that the FPGA device used for this work is significantly more expensive compared to the GTX-480 GPU used in the proposed: $9,000 vs. $250. As such, the table compares operation throughput, as well as operation throughput per dollar. In the first case, the proposed design has 0.37x to 0.74x the operation throughput of the FPGA design, while if operation throughput per dollar is considered, the proposed design performs 14.55x to 28.75x better.

8.8 Summary

This chapter presented the results of an in-depth search of various methods and techniques to further improve GPU-based finite field multiplication for elliptic curve cryptography. Various asymptotically fast multiplication techniques, reduction algorithms, explicit cache management techniques, and the optimization of GPU resource utilization were explored. Through mathematical analyses and experimentation, it was shown that:

- Schoolbook multiplication is best suited for the NVIDIA Fermi architecture and field sizes of practical interest for elliptic curve cryptography
- While NIST reduction is fast, Montgomery multiplication can match and surpass its performance on a GPU for field sizes of interest for elliptic curve cryptography
- Explicit cache management only offers very modest performance gains at the field sizes of interest for elliptic curve cryptography
- Optimizing the number of active threads per SM, and the number of registers

per thread is crucial, offering perhaps the largest performance gains for minimal implementation difficulty

- Finite field inversion is a very expensive operation for the GPU; further research in this area may be beneficial

A complete finite field arithmetic library for the GPU was developed, and the resulting work is, to the best of the author's knowledge, faster than any other CPU or GPU implementation in the literature: the proposed algorithm's multiplication operation throughput is 1.24x to 1.72x greater than the next fastest GPU implementation. Compared to the fastest FPGA implementation, the work is 0.37 to 0.74x the speed, however if the device cost is considered, the proposed GPU solution performs much better.

The proposed finite field arithmetic library is fully capable of implementing elliptic curve scalar point multiplication, enabling GPUs to accelerate elliptic curve cryptography operations.

CHAPTER 9

Elliptic Curve Scalar Point Multiplication for the GPU

9.1 Introduction

The finite field library developed in the previous chapter is now used to create a scalar point multiplication algorithm for the GPU.

In section 9.2 Elliptic curve point operations are carefully chosen, section 9.3 reviews scalar point multiplication, and in section 9.4 an algorithm suitable for the GPU's SPMD architecture is proposed. The algorithm is tested, and a number of performance metrics are determined, including peak operation throughput (section 9.5, minimum batch size to obtain the peak operation throughput (section 9.6), and additional latency compared to a CPU implementation (section 9.7). In section 9.8, the operation throughput of a CPU implementation using Intel's IPP functions is measured and compared to the proposed, along with the best CPU, GPU and FPGA designs reported to date in the open literature, and section 9.9 summarizes the chapter.

9.2 Elliptic curve point addition and doubling

The point doubling and addition formulae presented in Chapter 2 require one inversion per point addition / double. Recall from Chapter 8, section 8.7 that finite field inversion for the GPU is prohibitively expensive, and that every effort must be taken to avoid it.

This can be accomplished with the use of projective coordinate systems which do not require inversion for elliptic curve point addition and subtraction. It is possible to transform from the affine coordinate system which was used in Chapter 2 to a projective coordinate system through a change in variables [34]; for example, affine points can be converted to use Jacobian coordinates by converting (x, y) to $(X/Z^2, Y/Z^3, Z)$.

A large number of point addition and doubling formulae using different coordinate systems have been proposed, each requiring different amounts of underlying finite field operations. Shown in tables 9.1 and 9.2 are the point doubling and addition costs for each of these different coordinate systems. By using the table of the finite field arithmetic operation costs from section 8.7 in Chapter 8, elliptic curve point doubling and addition costs can be estimated, as shown in tables 9.3 and 9.4.

Table 9.1: Elliptic curve point doubling operation costs for different coordinate systems & formulae

ID	Name	Coordinates	Mult.	Const. Mult.	Sqr.	Add./Sub.	Inv.
dbl-A	Affine	$A \leftarrow 2A$	1	1	1	4	1
dbl-B	Bernstein 2001[84]	$J \leftarrow 2J$	3	4	5	8	0
dbl-C	Hankerson [34]	$J \leftarrow 2J$	4	3	4	5	0
dbl-D	Hasegawa [85]	$J \leftarrow 2J$	4	11	4	7	0
dbl-E	Bernstein 2007 [86]	$J \leftarrow 2J$	1	5	8	10	0
dbl-F	Cohen [87]	$J \leftarrow 2J$	3	6	6	4	0
dbl-G	Hasegawa [85]	$J \leftarrow 2J$	3	5	6	6	0
dbl-H	Chudnovsky [88]	$J \leftarrow 2J$	4	5	6	5	0
dbl-I	Chudnovsky [88]	$J \leftarrow 2J$	3	6	7	4	0
dbl-J	Cohen [87]	$J \leftarrow 2J$	3	6	7	4	0
dbl-K	Bernstein 2007 [86]	$P \leftarrow 2P$	7	5	3	5	0
dbl-L	Cohen [87]	$P \leftarrow 2P$	6	7	5	4	0
dbl-M	Cohen [87]	$P \leftarrow 2P$	7	7	6	4	0

As shown in the highlighted column in tables 9.3 and 9.4 are the lowest-cost

Table 9.2: Elliptic curve point addition operation costs for different coordinate systems & formulae

ID	Name	Coordinates	Mult.	Const. Mult.	Sqr.	Add./Sub.	Inv.
add-A	Affine	$A \leftarrow A + A$	1	0	1	6	1
add-B	Bernstein 2007 [86]	$J \leftarrow J + A$	7	4	4	9	0
add-C	Hankerson [34]	$J \leftarrow J + A$	8	1	3	6	0
add-D	Cohen [87]	$P \leftarrow P + A$	9	1	2	6	0
add-E	Cohen [87]	$P \leftarrow P + P$	12	1	2	6	0
add-F	Bernstein 2007 [86]	$P \leftarrow P + P$	11	6	6	10	0
add-G	Chudnovsky [88]	$P \leftarrow P + P$	11	2	5	7	0
add-H	Cohen [87]	$P \leftarrow P + P$	19	1	6	6	0
add-I	Bernstein 2007 [86]	$J \leftarrow J + J$	11	4	5	9	0
add-J	Cohen [87]	$J \leftarrow J + J$	12	1	4	6	0
add-K	Bernstein 2001 [84]	$J \leftarrow J + J$	12	1	4	6	0

Table 9.3: Elliptic curve point doubling costs in ns for different coordinate systems & formulae

Field Size	dbl-A	dbl-B	dbl-C	dbl-D	dbl-E	dbl-F	dbl-G	dbl-H	dbl-I	dbl-J	dbl-K	dbl-L	dbl-M
112	50.9	7.6	6.1	9.8	9.0	7.0	7.5	6.75	7.5	7.4	7.8	8.3	9.1
128	58.0	7.6	6.1	9.7	8.9	7.0	7.5	6.71	7.5	7.3	7.7	8.2	9.0
160	108.8	9.6	7.7	12.5	11.4	9.0	9.6	8.59	9.6	9.5	9.8	10.6	11.6
192	181.5	11.9	9.6	15.6	14.1	11.4	11.9	10.75	12.0	12.0	12.2	13.3	14.6
224	287.8	14.3	11.6	19.1	17.0	14.0	14.5	13.11	14.7	14.8	14.8	16.4	17.9
256	421.2	17.0	14.0	23.1	20.2	17.0	17.5	15.82	17.8	18.0	17.9	20.0	21.9
320	808.2	23.6	19.7	32.7	28.0	24.3	24.6	22.36	25.2	25.7	25.4	28.6	31.4
384	1387.8	31.3	26.4	44.0	37.0	33.0	33.1	30.12	34.0	34.9	34.2	38.9	42.8
512	3241.5	50.4	43.2	72.7	59.3	55.0	54.3	49.64	56.4	58.3	56.9	65.3	72.0
521	7186.8	55.8	47.9	80.8	65.7	61.3	60.4	62.72	62.7	64.9	63.1	72.7	80.1

Table 9.4: Elliptic curve point addition costs in ns for different coordinate systems & formulae

Field Size	add-A	add-B	add-C	add-D	add-E	add-F	add-G	add-H	add-I	add-J	add-K
112	51.3	9.4	7.2	7.3	8.5	12.9	9.9	12.8	11.4	9.2	9.2
128	58.5	9.4	7.2	7.3	8.5	12.8	9.9	12.8	11.4	9.2	9.2
160	109.4	11.7	8.9	9.0	10.6	16.1	12.3	16.0	14.3	11.5	11.5
192	182.1	14.4	10.9	11.0	12.9	19.9	15.2	19.7	17.6	14.1	14.1
224	288.2	17.3	13.1	13.1	15.4	24.1	18.3	23.9	21.1	16.9	16.9
256	421.6	20.7	15.7	15.8	18.7	29.1	22.2	29.2	25.5	20.5	20.5
320	808.5	28.8	22.0	22.1	26.3	40.8	31.3	41.7	35.8	29.1	29.1
384	1387.9	38.3	29.4	29.5	35.4	54.8	42.1	56.7	48.0	39.2	39.2
512	3240.8	62.0	48.2	48.4	58.6	90.1	69.5	95.3	78.9	65.0	65.0
521	7185.9	68.6	53.3	53.4	64.7	99.9	77.0	105.7	87.4	72.0	72.0

coordinate system and formulae for elliptic curve point operations across all field sizes, and both are from Hankerson et al. [34].

Doubling uses the Jacobian coordinates described earlier, and point addition uses a

mixed Jacobian-affine coordinate system, where one of the operands is kept in affine form. The proposed GPU implementation uses these point addition and doubling formulae presented on page 91 and 92 of [34], and are reproduced here in algorithms 9.2 and 9.1.

Algorithm 9.1: Selected point doubling formula [34]

input : $P = (X_1, Y_1, Z_1)$ in Jacobian coordinates
output: $2P = (X_3, Y_3, Z_3)$ in Jacobian coordinates

1 **if** $P = \infty$ **then return** ∞
2 $T_1 \leftarrow Z_1^2$
3 $T_2 \leftarrow X_1 - T_1$
4 $T_1 \leftarrow X_1 + T_1$
5 $T_2 \leftarrow T_2 \cdot T_1$
6 $T_2 \leftarrow 3T_2$
7 $Y_3 \leftarrow 2Y_1$
8 $Z_3 \leftarrow Y_3 \cdot Z_1$
9 $Y_3 \leftarrow Y_3^2$
10 $T_3 \leftarrow Y_3 \cdot X_1$
11 $Y_3 \leftarrow Y_3^2$
12 $Y_3 \leftarrow Y_3/2$
13 $X_3 \leftarrow T_2^2$
14 $T_1 \leftarrow 2T_3$
15 $X_3 \leftarrow X_3 - T_1$
16 $T_1 \leftarrow T_3 - X_3$
17 $T_1 \leftarrow T_1 \cdot T_2$
18 $Y_3 \leftarrow T_1 - Y_3$
19 **return** *(X3, Y3, Z3)*

Algorithm 9.2: Selected point addition formula [34]

input : $P = (X_1, Y_1, Z_1)$ in Jacobian coordinates
$Q = (x_2, y_2)$ in affine coordinates
output: $P + Q = (X_3, Y_3, Z_3)$ in Jacobian coordinates

if $Q = \infty$ **then return** (X_1, Y_1, Z_1)
if $P = \infty$ **then return** $(x_2, y_2, 1)$
$T_1 \leftarrow Z_1^2$
$T_2 \leftarrow T_1 \cdot Z_1$
$T_1 \leftarrow T_1 \cdot x_2$
$T_2 \leftarrow T_2 \cdot y_2$
$T_1 \leftarrow T_1 - X_1$
$T_2 \leftarrow T_2 - Y_1$
if $T_1 = 0$ **then**
  if $T_2 = 0$ **then** use Algorithm 9.1
  else return ∞
else
  return ∞
end
$Z_3 \leftarrow Z_1 \cdot T_1$
$T_3 \leftarrow T_1^2$
$T_4 \leftarrow T_3 \cdot T_1$
$T_3 \leftarrow T_3 \cdot X_1$
$T_1 \leftarrow 2T_3$
$X_3 \leftarrow T_2^2$
$X_3 \leftarrow X_3 - T_1$
$X_3 \leftarrow X_3 - T_4$
$T_3 \leftarrow T_3 - X_3$
$T_3 \leftarrow T_3 \cdot T_2$
$T_4 \leftarrow T_4 \cdot Y_1$
$Y_3 \leftarrow T_3 - T_4$
return *(X3, Y3, Z3)*

9.3 Scalar point multiplication

Recall from Chapter 2 that scalar point multiplication, for a scalar k, and points Q & P on an elliptic curve $\mathbf{E}$ with group order n is defined as:

$$Q = kP, \text{ for } Q, P \in \mathbf{E} \text{ and } k \in [0, n-1] \tag{9.1}$$

$$= P + P + \ \ldots \ + P \text{ (k times)} \tag{9.2}$$

The most straightforward way to compute Q is a simple double-and-add algorithm [89], as shown in Figure 9.3. All scalar point multiplication algorithms require elliptic point formulae for addition and doubling, or addition and halving. The latter operation, elliptic curve point halving, is used for scalar point multiplication over binary fields, and is outside the scope of this work.

Algorithm 9.3: Right-to-left double and add scalar point multiplication [34]

input : $k = (k_{W-1}, \ldots, k_1, k_0), P \in \mathbf{E}$
output: kP

```
1 Q ← ∞
2 for i ← 0 to W − 1 do
3 |   if k_i = 1 then
4 |   |   Q ← Q + P
5 |   end
6 |   P ← 2P
7 end
8 return Q
```

Assuming the average Hamming weight of the m-bit wide scalar k is $\frac{m}{2}$, the average cost of the double-and-add algorithm is $\frac{m}{2}A + mD$, where A is the cost of a point addition operation, and D is the cost of point doubling.

The majority of techniques in the literature which aim to reduce the cost of scalar point multiplication focus on reducing the average number of point addition operations required. The most common (and successful) approach is to re-code the

scalar k to a non-adjacent form (NAF), which is very similar to Booth encoding in practice: the scalar k is converted from a binary form to a signed-digit representation which reduces the average number of non-zero digits in k [34].

Unfortunately, none of these techniques may be successfully applied to the proposed GPU implementation: the underlying finite field arithmetic library presented in Chapter 8 uses one thread per operation, and threads are processed in warps of 32. If the scalar k values are assigned to NAF representation, it is unlikely that all 32 threads belonging to the same warp have the same NAF of k. The result is that the cost of scalar point multiplication on the GPU is changed from processing one scalar multiplication per thread, it is not possible to $\frac{m}{2}A + mD$ to $m(A + D)$. This is one of the fundamental problems of SPMD/SIMD architectures, which in this case is overcome by the very fast underlying finite field arithmetic operations.

9.4 Proposed GPU-based scalar point multiplication algorithm implementation

The proposed scalar point multiplication algorithm is a straightforward implementation of algorithm 9.3, which in turn uses algorithms 9.1 and 9.2 for point doubling and addition. As with the finite field library, the scalar point algorithm has been implemented using NVIDIA's PTX code. Point doubling and addition are implemented using a series of function calls to the finite field library functions presented in Chapter 8, which use the same memory and thread configurations as before. The proposed implementation uses 4 W-wide temporary storage locations for T_1, T_2, T_3, and T_4, 5 locations to store operands for X_1, Y_1, and Z_1, x_2 and y_2, and finally one additional location to store the scalar multiple k.

9.5 Scalar point multiplication operation throughput and comparison to CPU implementation

The proposed scalar point multiplication algorithm was tested using the same procedures outlined in section 8.7 in Chapter 8, and operation throughput was measured. All timing results include transfer time from the host machine to the GPU and vice-versa.

Tests were run on three families of curves; the SEC curves [37], which are a superset of the NIST curves, the Brainpool curves [40], which have randomly chosen parameters a and b, and the twisted Brainpool curves, which use $a = -3$, and randomly chosen b. Curves with $a \neq 3$ have a slightly more expensive point-doubling formula, which the proposed elliptic curve scalar point multiplication algorithm uses in such cases.

For comparison, the Intel IPP [74] library functions were also implemented and measured. Although there are other results reported in the literature, it was not possible to find any for the Brainpool curves, or the smaller (and now deprecated) field sizes in the SEC curves. Comparisons to results in the literature are made where possible in section 9.8 The Intel results are from a Dell XPS 8500, which uses all 8 logical cores of a 3.6 GHz Intel core-i7 3770 that was described in Chapter 5. The Intel functions are heavily optimized for the SEC curves, while they possess no special optimizations for either set of Brainpool curves.

As shown in tables 9.5, 9.6, and 9.7, the operation throughput of the proposed GPU-based scalar point multiplication algorithm is between 5.4x and 31.1x greater than the Intel CPU and IPP function implementation. The field with smallest improvement is the SEC/NIST 521 bit field. This result is not unexpected: the proposed implementation enjoys the highest performance gains compared to the CPU implementation over smaller field sizes, and the 521-bit NIST field uses a Mersenne

prime, which allows for particularly efficient reduction. Finally, results for both sets of Brainpool curves are roughly equivalent with respect to one another, with the twisted Brainpool curves possessing slightly higher operation throughput, as expected.

Table 9.5: Maximum operation throughput for scalar point multiplication over the SEC curves

Field Size	Max CPU Throughput (kP/s)	Max GPU Throughput (kP/s)	Improvement
112	43,85	1,382,576	31.1
128	77,101	1,208,078	15.7
160	25,157	669,742	26.6
192	42,230	424,424	10.1
224	30,321	277,068	9.1
256	22,619	191,779	8.5
384	9,506	56,927	6.0
521	3,822	20,743	5.4

Table 9.6: Maximum operation throughput for scalar point multiplication over the Brainpool curves

Field Size	Max CPU Throughput (kP/s)	Max GPU Throughput (kP/s)	Improvement
160	24,284	647,370	26.7
192	21,035	398,674	19.0
224	14,130	255,257	18.1
256	12,425	173,269	13.9
320	7,581	89,475	11.8
384	5,325	52,413	9.8
512	2,904	22,009	7.6

Table 9.7: Maximum operation throughput for scalar point multiplication over the twisted Brainpool curves

Field Size	Max CPU Throughput (kP/s)	Max GPU Throughput (kP/s)	Improvement
160	25,820	678,381	26.3
192	22,272	422,114	19.0
224	15,223	273,800	18.0
256	13,323	186,346	14.0
320	8,139	96,261	11.8
384	5,763	56,452	9.8
512	3,178	23,850	7.5

9.6 Operation batch size vs. operation throughput

In this section, the operation throughput vs. batch size is examined for each curve family. The goal is to determine the number of scalar point multiplications that must be performed by the GPU in order to obtain the operation throughput results shown in the previous section.

Tests were carried out using the same approach as before, however this time the total number of threads to be processed by the entire GPU was varied from 1 to 32768. Results are shown in figures 9.1, 9.2 (which is the same as the previous Figure, except it does not show curves 112 and 128), 9.3, and 9.4. Note that all figures use a logarithmic Y-axis. In the case of all curves tested here, full operation throughput was obtained with a batch size of approximately 12,000 to 15,000 scalar multiplications.

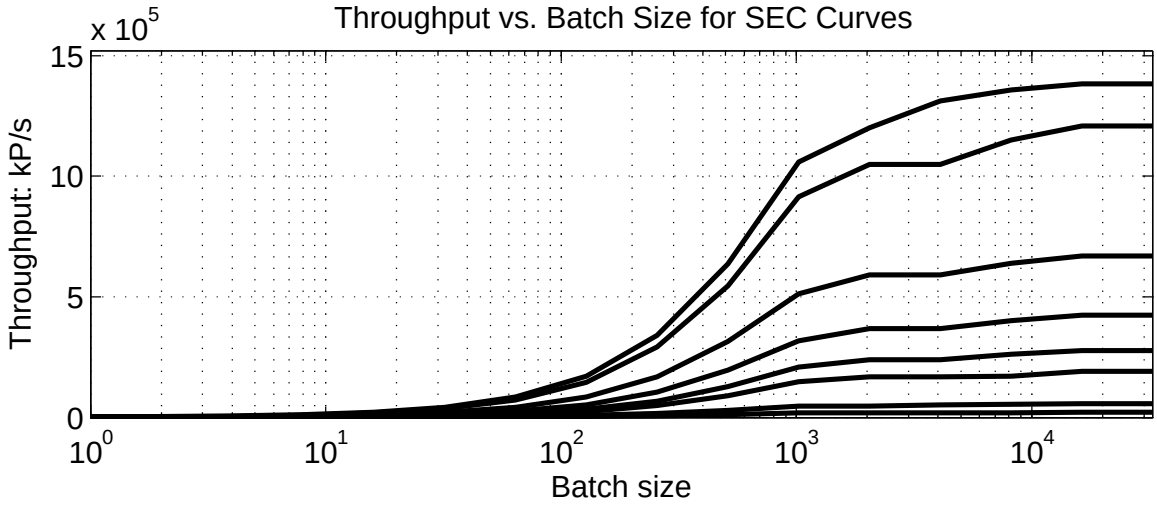


Figure 9.1: Operation throughput vs. batch size for SEC curves; from top to bottom: 112, 128, 160, 192, 224, 256, 384, 521

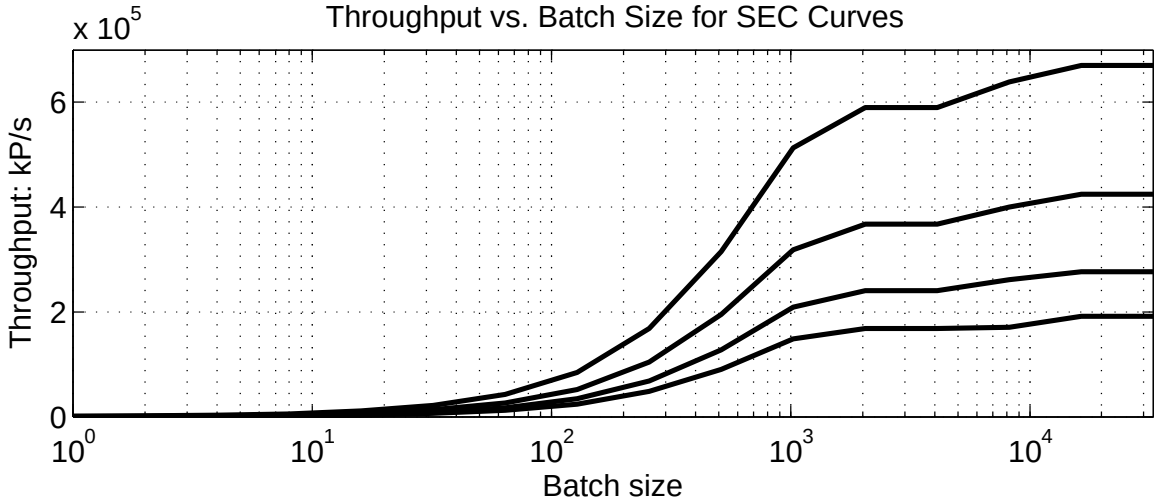


Figure 9.2: Operation throughput vs. batch size for SEC curves excluding 112 and 128; from top to bottom: 160, 192, 224, 256, 384, 521

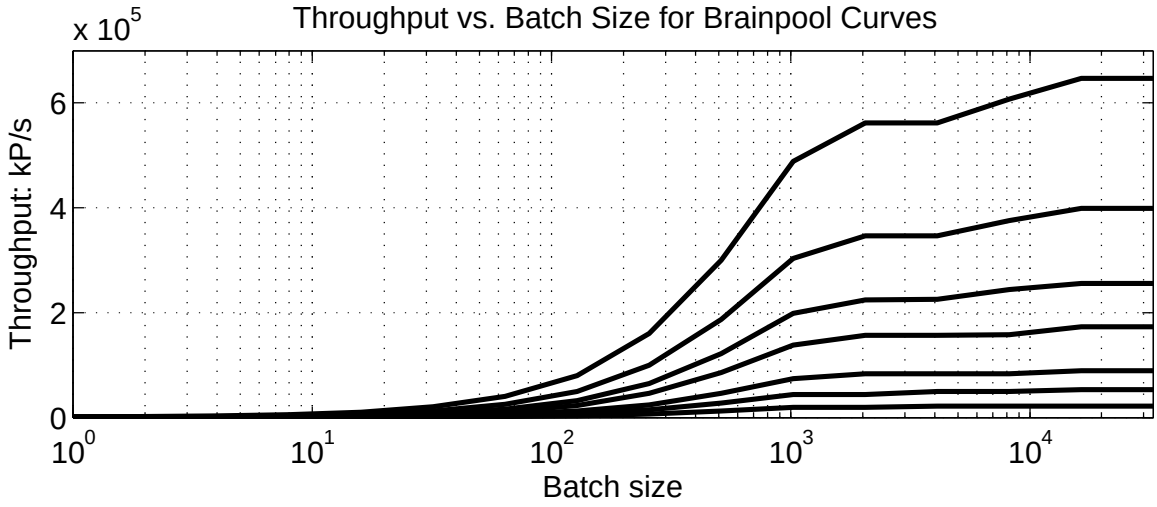


Figure 9.3: Operation throughput vs. batch size for Brainpool curves; from top to bottom: 160, 192, 224, 256, 320, 384, 512

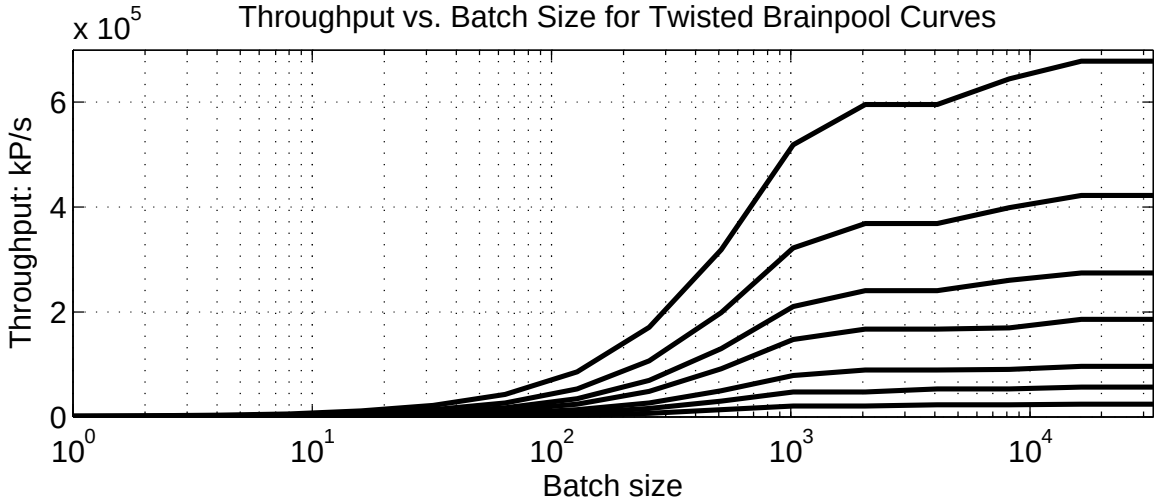


Figure 9.4: Operation throughput vs. batch size for twisted Brainpool curves; from top to bottom: 160, 192, 224, 256, 320, 384, 512

9.7 Scalar point multiplication latency

The proposed scalar point multiplication algorithm was also compared in terms of additional latency cost vs. the Intel IPP CPU implementation. A significant amount of latency is unavoidable, due to the fact that the operands must travel from the CPU, to the CPU RAM, across the PCI-Express bus to the GPU RAM, through the GPU and back to the CPU RAM. The goal here is to determine additional latency in milliseconds that is caused by using the GPU to perform the operation. Shown in Figure 9.5, it can be seen that for smaller field sizes, the additional latency is negligible, while it becomes quite significant for field sizes greater than 320 bits, peaking at just over 600 milliseconds for the 512-bit Brainpool curve.

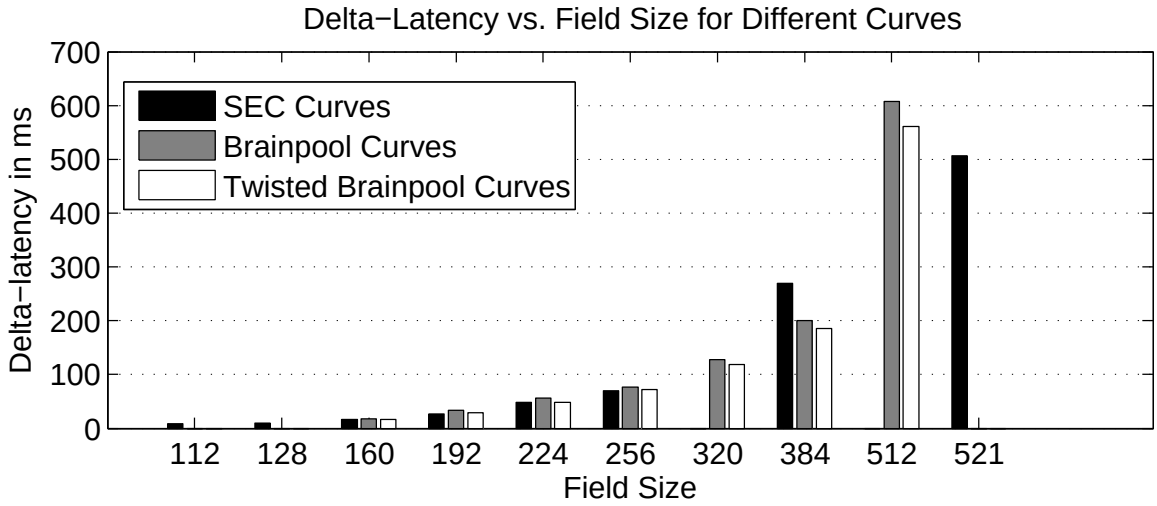


Figure 9.5: Scalar point multiplication GPU vs. CPU Δ-latency

9.8 Comparison to results in the literature

Shown in Table 9.8 is a comparison of the proposed scalar point multiplication algorithm to the results found in the open literature. Beginning from the left, Giorgi et al. use a GPU implementation based on their finite field multiplication algorithm which was discussed in chapters 7 and 8.

The next column presents work by Antao et al. who developed an RNS-based finite field multiplication algorithm with lazy reduction. Their work is implemented on an NVIDIA GTX 285 GPU, which is one generation behind the GTX-480 used for the proposed.

In the third column from the right is the work of Szerwinsky and Güneysu, which is one of the first GPU implementations of scalar point multiplication.

In the second column from the right is an FPGA-based design from Güneysu and Paar, which uses a $2000 Virtex 4 FPGA to perform scalar point multiplication. Their results are the next fastest ones in the literature; the proposed algorithm is 6.0x faster than their 224-bit design, and 7.8x faster than the 256-bit one. Notably, the results published by Güneysu and Paar are the only ones in the literature that outperform the Intel IPP library on the core-i7 CPU.

Table 9.8: Scalar point multiplication operation throughput of the proposed GPU-based algorithm to the literature

Field Size	Giorgi [68]	Antao [70]	Szerwinsky [67]	Güneysu [90]	Proposed
160	5,586				669,742
192	3,289				424,424
224	1,972	9,827	1,412	37,736	227,068
256	1,620			24,691	191,779
384	216				56,927

9.9 Summary

In this chapter, a GPU-based scalar point multiplication algorithm was proposed and profiled. The algorithm uses Jacobian coordinates for elliptic curve point doubling, and Jacobian-affine coordinates for point addition. The scalar point multiplication algorithm itself uses a simple double-and-add approach that is suitable for the GPU's SPMD architecture.

Scalar point multiplication was tested on three families of curves over $\mathbb{F}_p$: the SEC

curves, Brainpool curves, and the twisted Brainpool curves. Operation throughput was measured and compared to a highly optimized CPU implementation running on a recently released core-i7 CPU; the proposed work boasted operation throughput that was 5x to 31x faster than the 8-logical-core CPU, with best performance gains at smaller field sizes.

A batch-size analysis was performed, and it was determined that groups of approximately 12,000 to 16,000 threads must be executed on the GPU at a time in order to realize its full operation throughput potential.

Additional latency caused by the use of the GPU was determined; for small field sizes it is negligible, however for field sizes above 320 bits it may grow considerably, which may be an issue for time-sensitive applications, and should be subject to further research.

Finally, the proposed work was compared to the best designs in the literature; the proposed design is 6x to 7.7x faster than the next fastest reported design.

CHAPTER 10

Conclusions

10.1 Summary of contributions

In this dissertation, two implementations of a serial-in, parallel-out finite 233-bit field multiplier using reordered normal basis were presented. Both multipliers used serially connected copies of a block. Although the proposed implementation used a field size of 233 bits, in a multiplier over any field where there exists a Type-II ONB could be created using the same building block and design approach.

The first architecture used a combination of custom domino logic and standard CMOS library cells. The speed improvement was measured to be 99% in comparison to static CMOS implementation, while area reduction was 49%. The final design was successfully simulated up to a clock rate of 1.587 GHz.

The improved design employed a full-custom solution which allowed finite field multiplication to be carried out 12% faster than the semi-custom domino logic design, while reducing area utilization by 43%.

The proposed architectures are suitable for integration with a CPU in order to provide a special finite field multiplication instruction to accelerate elliptic curve scalar point multiplication.

A GPU-based type-II ONB multiplication algorithm was proposed, which, to the author's knowledge, is the fastest GPU-based finite field multiplier for binary extension fields. The proposed algorithm uses multiple GPU threads to carry out each multiplication operation, and several multiplications are carried out simultaneously in order to saturate the GPU's resources. While the algorithm's operation throughput does not surpass the state-of-the-art CPU implementation at this time, it may prove useful as GPU architectures evolve to possess better logic-operation performance.

A new algorithm for computing modular multiplication over the NIST prime fields on a GPU was presented. The algorithm was implemented entirely in PTX assembly, and validated against the results generated by a popular multi-precision software package. Compared to the next-fastest GPU-based algorithm, the proposed is 1.36 to 1.59 times higher operation throughput, depending on the field size. Compared to the Intel IPP finite field multiplication functions running on a CPU, the proposed algorithm has 58.6 to 93.0 times higher operation throughput. To the best of the author's knowledge, the proposed algorithm boasts the highest NIST prime field multiplication operation throughput in the literature.

A Montgomery multiplication algorithm for the GPU was developed, which allows finite field multiplication with the use of any field prime. The proposed algorithm was carefully optimized to have the best threads per SM vs. resources per thread combination, and it was shown that this optimization has a more profound effect than integrating cache into the multiplication stage. Asymptotically fast multiplication methods were explored, and it was determined that for the field sizes that are of interest in elliptic curve cryptography, schoolbook multiplication performs best. The proposed algorithm's operation throughput is 1.24x to 1.72x greater than the next

fastest GPU design design in the literature, and considering operation throughput per dollar as a metric, the proposed design is 14.55x to 28.75x better than the state-of-the-art FPGA implementation.

A finite field inversion algorithm for the GPU based on Fermat's little theorem was developed. Compared to the more commonly used binary inversion algorithm, which performs very poorly on the GPU due to heavy, operand-dependent program branching, the proposed Fermat inversion algorithm provided acceptable results. To the best of the author's knowledge, the proposed inversion algorithm is the first attempt to carry out this operation on a GPU.

A complete finite field arithmetic library was implemented entirely in PTX assembly, and then used to perform scalar point multiplication.

A complete finite field arithmetic library for the GPU was developed, and the resulting work is, to the best of the author's knowledge, faster than any other CPU or GPU implementation in the literature: the proposed algorithm's operation throughput is 6x to 7.7x greater than the next fastest FPGA design, and 5x to 31x greater than a powerful desktop CPU.

The performance of the GPU-based scalar point multiplication algorithm was further analyzed in terms of batch size required to obtain peak operation throughput, and the latency cost added by the GPU.

Overall, GPUs were shown to be suitable devices for accelerating elliptic curve scalar point multiplication, and could be used in a high-demand server environment to perform such operations.

10.2 Future work

In terms of future work, there are a number of areas worthy of further exploration.

ASIC architectures

While the multiplier architectures presented in this work are the fastest in the literature, much more sophisticated fabrication technologies have since been made available to the University of Windsor. It would be interesting to determine the performance of the implemented architectures in a 28nm or 22nm process.

If the opportunity were to become available, another avenue worth exploring is the integration of the proposed architecture with a CPU core; it may be possible to obtain an IP core from a company such as ARM in order to experiment with special instructions for accelerating cryptography.

Finite field inversion for the GPU

The most expensive operation, by far, is finite field inversion for the GPU. A number of techniques could be used to improve this operation, such as finding an efficient addition chain to compute a^{p-2}, rather than using the naïve approach proposed here.

Scalar point multiplication for binary fields

GPU-based scalar point multiplication over binary extension fields is a topic worth revisiting. NVIDIA's newly-released Kepler GPUs boast much higher logic and shift operation throughput, which will significantly improve the performance of multiplication over binary extension fields. Gaussian normal basis, reordered normal basis, and polynomial basis multiplication for the GPU should be explored more fully. Even if binary field multiplication performance is lacking compared to the state of the art CPU implementations, it may be possible to achieve much higher overall scalar point multiplication performance. This could be done by storing all operands in GPU registers and cache, which is much more feasible when multiple threads can pool their resources together on the same multiplication, as opposed to prime field multiplication which works best with one thread per operation.

Integrating the GPU accelerator with security APIs

The work presented in this dissertation established that GPUs may be used to accelerate elliptic curve scalar point multiplication over prime fields. Another major area of future work would be the integration of the proposed GPU system with a commonly used security API such as OpenSSL.

A number of challenges are likely to be encountered; multiplication operations must be queued and processed in batches, real-time or event-driven communication with the GPU will be required, and a method of testing a heavy-load system will require significant consideration.

References

[1] A. Namin, K. Leboeuf, R. Muscedere, H. Wu, and M. Ahmadi, "High-speed hardware implementation of a serial-in parallel-out finite field multiplier using reordered normal basis," *Circuits, Devices Systems, IET*, vol. 4, no. 2, pp. 168 –179, march 2010.

[2] K. Leboeuf, A. Namin, H. Wu, R. Muscedere, and M. Ahmadi, "Efficient VLSI implementation of a finite field multiplier using reordered normal basis," in *Circuits and Systems (MWSCAS), 2010 53rd IEEE International Midwest Symposium on*, aug. 2010, pp. 1218 –1221.

[3] K. Leboeuf, R. Muscedere, and M. Ahmadi, "High performance prime field multiplication for GPU," in *Circuits and Systems (ISCAS), 2012 IEEE International Symposium on*, may 2012, pp. 93 –96.

[4] H. Wu, M. Hasan, I. Blake, and S. Gao, "Finite field multiplier using redundant representation," *Computers, IEEE Transactions on*, vol. 51, no. 11, pp. 1306–1316, 2002.

[5] J. Yuan and C. Svensson, "New single-clock CMOS latches and flipflops with improved speed and power savings," *Solid-State Circuits, IEEE Journal of*, vol. 32, no. 1, pp. 62–69, 1997.

[6] NVIDIA Corporation, "NVIDIA CUDA C programming guide," 2012.

[7] E. Butler and I. Gallagher, "Hey web 2.0: Start protecting user privacy instead of pretending to," in *Information Security Conference: ToorCon 12 Sandiego*, 2010. [Online]. Available: http://codebutler.com/firesheep

[8] D. Vikan, "Why firesheep works and how to counter it," Tech. Rep., 2011.

[9] T. Kleinjung, K. Aoki, J. Franke, A. Lenstra, E. Thomé, J. Bos, P. Gaudry, A. Kruppa, P. Montgomery, D. Osvik *et al.*, "Factorization of a 768-bit rsa modulus," *Advances in Cryptology–CRYPTO 2010*, pp. 333–350, 2010.

[10] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of computation*, vol. 48, no. 177, pp. 203–209, 1987.

[11] V. Miller, "Use of elliptic curves in cryptography," in *Advances in Cryptology-CRYPTO 85 Proceedings*. Springer, 1986, pp. 417–426.

[12] A. Lenstra and E. Verheul, "Selecting cryptographic key sizes," *Journal of cryptology*, vol. 14, no. 4, pp. 255–293, 2001.

[13] N. Smart, "ECRYPT II yearly report on algorithms and keysizes," *Framework*, no. March, 2010.

[14] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid, "Recommendation for key management–part 1: General (revision 3)," *NIST special publication*, vol. 800, p. 57, 2011.

[15] *Référentiel Général de Sécurité*, ANSSI Mécanismes cryptographiques Règles et recommandations concernant le choix et le dimensionnement des mécanismes cryptographiques, January 2010.

[16] NSA, "Fact sheet: Suite B cryptography," 2011.

[17] ——, "The case for elliptic curve cryptography," 2009.

[18] A. Koblitz, N. Koblitz, and A. Menezes, "Elliptic curve cryptography: The serpentine course of a paradigm shift," *Journal of Number Theory*, vol. 131, no. 5, pp. 781–814, 2011.

[19] S. Gueron, "Intel advanced encryption standard (aes) instructions set," *Intel White Paper, Rev*, vol. 3, 2010.

[20] I. Herstein, *Topics in Algebra*. John Wiley & Sons, 1975.

[21] W. Stallings, *Cryptography and network security*. Prentice hall, 2003, vol. 2.

[22] R. Lidl and H. Niederreiter, *Introduction to finite fields and their applications*. Cambridge University Press, 1994.

[23] E. Galois, R. Bourgne, and J. Azra, *Écrits et mémoires mathématiques d'Évariste Galois*. Gauthier-Villars Paris, 1962.

[24] C. Koc and B. Sunar, "Low-complexity bit-parallel canonical and normal basis multipliers for a class of finite fields," *Computers, IEEE Transactions on*, vol. 47, no. 3, pp. 353 –356, mar 1998.

[25] T. Itoh and S. Tsujii, "Structure of parallel multipliers for a class of fields $GF(2^m)$," *Information and computation*, vol. 83, no. 1, pp. 21–40, 1989.

[26] J. Taverne, A. Faz-Hernández, D. Aranha, F. Rodríguez-Henríquez, D. Hankerson, and J. López, "Software implementation of binary elliptic curves: impact of the carry-less multiplier on scalar multiplication," *Cryptographic Hardware and Embedded Systems–CHES 2011*, pp. 108–123, 2011.

[27] M. Rosing, "Implementing elliptic curve cryptography," 1999.

[28] R. Mullin, I. Onyszchuk, S. Vanstone, and R. Wilson, "Optimal normal bases in $GF(p^n)$," *Discrete Applied Mathematics*, vol. 22, no. 2, pp. 149–161, 1989.

[29] D. Ash, I. Blake, and S. Vanstone, "Low complexity normal bases," *Discrete Applied Mathematics*, vol. 25, no. 3, pp. 191–210, 1989.

[30] *X9.62 Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)*, American National Standards Institute Std., 1998.

[31] S. Gao and S. Vanstone, "On orders of optimal normal basis generators," *Mathematics of Computation*, vol. 64, no. 211, pp. 1227–1234, 1995.

[32] N. Koblitz, *Introduction to elliptic curves and modular forms*. Springer, 1993, vol. 97.

[33] ——, *A course in number theory and cryptography*. Springer, 1994, vol. 114.

[34] D. Hankerson, S. Vanstone, and A. Menezes, *Guide to elliptic curve cryptography*. Springer-Verlag New York Inc, 2004.

[35] *Digital Signature Standards 186-3*, National Institute of Standards and Technology FIPS Publication, 2009.

[36] *X9.63 Public Key Cryptography for the Financial Services Industry: Key Agreement and Key Transport Using Elliptic Curve Cryptography*, American National Standards Institute Std., 1998.

[37] *SEC 1: Elliptic Curve Cryptography*, Certicom Research Standards for Efficient Cryptography, 2009.

[38] R. Gallant, R. Lambert, and S. Vanstone, "Faster point multiplication on elliptic curves with efficient endomorphisms," in *Advances in Cryptology CRYPTO 2001*. Springer, 2001, pp. 190–200.

[39] *IEEE 1363-2000*, IEEE Standard Specifications for Public-Key Cryptography, January 2000.

[40] J. Merkle and M. Lochter, *Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation*, Std., 2010.

[41] J. Omura and J. Massey, "Computational method and apparatus for finite field arithmetic," U.S. Patent 4,587,627, 1986.

[42] L. Gao and G. Sobelman, "Improved vlsi designs for multiplication and inversion in $GF(2^m)$ over normal bases," in *ASIC/SOC Conference, 2000. Proceedings. 13th Annual IEEE International.* IEEE, 2000, pp. 97–101.

[43] W. Geiselmann and D. Gollmann, "Self-dual bases in," *Designs, Codes and Cryptography*, vol. 3, no. 4, pp. 333–345, 1993.

[44] G. Feng, "A VLSI architecture for fast inversion in $GF(2^m)$," *Computers, IEEE Transactions on*, vol. 38, no. 10, pp. 1383–1386, 1989.

[45] G. Agnew, R. Mullin, I. Onyszchuk, and S. Vanstone, "An implementation for a fast public-key cryptosystem," *Journal of CRYPTOLOGY*, vol. 3, no. 2, pp. 63–79, 1991.

[46] A. Reyhani-Masoleh and M. Hasan, "Efficient digit-serial normal basis multipliers over binary extension fields," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 3, no. 3, pp. 575–592, 2004.

[47] ——, "Low complexity word-level sequential normal basis multipliers," *Computers, IEEE Transactions on*, vol. 54, no. 2, pp. 98–110, 2005.

[48] S. Kwon, K. Gaj, C. H. Kim, and C. P. Hong, "Efficient linear array for multiplication in $GF(2^m)$ using a normal basis for elliptic curve cryptography," in *Cryptographic Hardware and Embedded Systems - CHES 2004*, ser. Lecture Notes in Computer Science, vol. 3156. Springer Berlin / Heidelberg, 2004, pp. 76–91.

[49] D. Yang, C. Kim, Y. Park, Y. Kim, and J. Lim, "Modified sequential normal basis multipliers for type II optimal normal bases," in *Computational Science and Its Applications, ICCSA 2005*, ser. Lecture Notes in Computer Science, vol. 3481. Springer Berlin / Heidelberg, 2005, pp. 100–101.

[50] J. Uyemura, *CMOS logic circuit design.* Springer, 1999.

[51] P. Srivastava, A. Pua, and L. Welch, "Issues in the design of domino logic circuits," in *VLSI, 1998. Proceedings of the 8th Great Lakes Symposium on.* IEEE, 1998, pp. 108–112.

[52] available through Canadian Microelectronics Corporation, "Virtual silicon technologhy," Standard Cell Library, $0.18\mu m$ TSMC CMOS process, 1999.

[53] *Implementation of Elliptic Curve Cryptographic Coprocessor over $GF(2^m)$ on an FPGA*, ser. Lecture Notes in Computer Science, vol. 1965. Springer Berlin Heidelberg, 2000.

[54] A. Satoh and K. Takano, "A scalable dual-field elliptic curve cryptographic processor," *Computers, IEEE Transactions on*, vol. 52, no. 4, pp. 449 – 460, april 2003.

[55] W. Tang, H. Wu, and M. Ahmadi, "Vlsi implementation of bit-parallel word-serial multiplier in $GF(2^{233})$," in *IEEE-NEWCAS Conference, 2005. The 3rd International*. IEEE, 2005, pp. 399–402.

[56] B. Ansari, "Efficient implementation of elliptic curve cryptography," Master's thesis, University of Windsor, Windsor, Ontario, 2004.

[57] P. Meher, "On efficient implementation of accumulation in finite field over $GF(2^m)$ and its applications," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 17, no. 4, pp. 541–550, 2009.

[58] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: stream computing on graphics hardware," in *ACM Transactions on Graphics (TOG)*, vol. 23, no. 3. ACM, 2004, pp. 777–786.

[59] I. Intel, "and IA-32 architectures software developer's manual," 2012.

[60] R. Dahab, D. Hankerson, F. Hu, M. Long, J. López, and A. Menezes, "Software multiplication using gaussian normal bases," *IEEE Transactions on Computers*, vol. 55, no. 8, pp. 974–984, 2006.

[61] P. Ning and Y. Yin, "Efficient software implementation for finite field multiplication in normal basis," *Information and Communications Security*, pp. 177–188, 2001.

[62] A. Reyhani-Masoleh and M. Hasan, "Fast normal basis multiplication using general purpose processors," *Computers, IEEE Transactions on*, vol. 52, no. 11, pp. 1379–1390, 2003.

[63] A. Cohen and K. Parhi, "Gpu accelerated elliptic curve cryptography in $GF(2^m)$," in *Circuits and Systems (MWSCAS), 2010 53rd IEEE International Midwest Symposium on*. IEEE, 2010, pp. 57–60.

[64] S. Fleissner, "GPU-accelerated montgomery exponentiation," in *Computational Science–ICCS 2007*. Springer, 2007, pp. 213–220.

[65] A. Moss, D. Page, and N. Smart, "Toward acceleration of RSA using 3d graphics hardware," in *Proceedings of the 11th IMA international conference on Cryptography and coding*. Springer-Verlag, 2007, pp. 364–383.

[66] M. Soderstrand, W. Jenkins, G. Jullien, and F. Taylor, *Residue number system arithmetic: modern applications in digital signal processing*. IEEE press, 1986.

[67] R. Szerwinski and T. Güneysu, "Exploiting the power of GPUs for asymmetric cryptography," in *Cryptographic Hardware and Embedded Systems–CHES 2008*. Springer, 2008, pp. 79–99.

[68] P. Giorgi, T. Izard, and A. Tisserand, "Comparison of modular arithmetic algorithms on gpus," in *ParCo'09: International Conference on Parallel Computing*, 2009.

[69] O. Harrison and J. Waldron, "Efficient acceleration of asymmetric cryptography on graphics hardware," in *Progress in Cryptology–AFRICACRYPT 2009*. Springer, 2009, pp. 350–367.

[70] S. Antão, J. Bajard, and L. Sousa, "RNS-based elliptic curve point multiplication for massive parallel architectures," *The Computer Journal*, pp. 629–647, 2011.

[71] S. Neves and F. Araujo, "On the performance of GPU public-key cryptography," in *Application-Specific Systems, Architectures and Processors (ASAP), 2011 IEEE International Conference on*. IEEE, 2011, pp. 133–140.

[72] R. Henry and I. Goldberg, "Solving discrete logarithms in smooth-order groups with CUDA," Cheriton School of Computer Science, University of Waterloo, Tech. Rep., 2012. [Online]. Available: http://cacr.uwaterloo.ca/techreports/2012/cacr2012-02.pdf

[73] T. G. et al., "GNU multiple precision arithmetic library," http://gmplib.org.

[74] Intel, ""Intel Integrated Perforamnce Primitives IPP 7.0", note = "http://http://software.intel.com/en-us/articles/intel-ipp/"."

[75] A. Karatsuba and Y. Ofman, "Multiplication of multidigit numbers on automata," in *Soviet physics doklady*, vol. 7, 1963, p. 595.

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

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

[78] A. Menezes, P. Van Oorschot, and S. Vanstone, *Handbook of applied cryptography*. CRC, 1997.

[79] P. Barrett, "Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor," in *Advances in Cryptology CRYPTO86*. Springer, 1987, pp. 311–323.

[80] W. Hasenplaugh, G. Gaubatz, and V. Gopal, "Fast modular reduction," in *Computer Arithmetic, 2007. ARITH '07. 18th IEEE Symposium on*, june 2007, pp. 225 –229.

[81] A. Bosselaers, R. Govaerts, and J. Vandewalle, "Comparison of three modular reduction functions," in *Advances in Cryptology CRYPTO93*. Springer, 1994, pp. 175–186.

[82] C. Kaya Koc, T. Acar, and B. Kaliski Jr, "Analyzing and comparing Montgomery multiplication algorithms," *Micro, IEEE*, vol. 16, no. 3, pp. 26–33, 1996.

[83] G. Chow, K. Eguro, W. Luk, and P. Leong, "A karatsuba-based montgomery multiplier," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*. IEEE, 2010, pp. 434–437.

[84] D. Bernstein, "A software implementation of nist p-224," in *Presentation at the 5th Workshop on Elliptic Curve Cryptography (ECC 2001)*, 2001.

[85] T. Hasegawa, J. Nakajima, and M. Matsui, "A practical implementation of elliptic curve cryptosystems over $gf(p)$ on a 16-bit microcomputer," in *Proceedings of the First International Workshop on Practice and Theory in Public Key Cryptography: Public Key Cryptography*, ser. PKC '98. London, UK, UK: Springer-Verlag, 1998, pp. 182–194.

[86] D. Bernstein and T. Lange, "Faster addition and doubling on elliptic curves," in *Proceedings of the Advances in Crypotology 13th international conference on Theory and application of cryptology and information security*. Springer-Verlag, 2007, pp. 29–50.

[87] H. Cohen, A. Miyaji, and T. Ono, "Efficient elliptic curve exponentiation using mixed coordinates," in *Advances in CryptologyASIACRYPT98*. Springer, 1998, pp. 51–65.

[88] D. Chudnovsky and G. Chudnovsky, "Sequences of numbers generated by addition in formal groups and new primality and factorization tests," *Advances in Applied Mathematics*, vol. 7, no. 4, pp. 385–434, 1986.

[89] D. Knuth, *Seminumerical algorithms, Volume 2 of The art of computer programming*. Addison-Wesley, Reading, Massachusetts,, 1981.

[90] T. Güneysu and C. Paar, "Ultra high performance ecc over nist primes on commercial fpgas," *Cryptographic Hardware and Embedded Systems–CHES 2008*, pp. 62–78, 2008.

IEEE Copyright

IET Copyright

Read Message

Message

From: "Bond,Anna" <ABond@theiet.org>
Subject: RE: request for written permission to use publication in doctoral thesis
Date: Mon, 6 Aug 2012 08:42:55 +0000
To: "leboeu3@uwindsor.ca" <leboeu3@uwindsor.ca>

Dear Karl Leboeuf,

I apologise for the delay in my reply, but this has only just been forwarded to me and I have been out of the office.

Permission to reproduce as requested is given, provided that the source of the material including the author, title, date, and publisher is acknowledged.

A reproduction fee is not due to the IET on this occasion.

Thank you for your enquiry. Please do not hesitate to contact me should you require any further information.

Kind regards,

Anna Bond
Editorial Assistant (Letters)
The IET

www.**theiet**.org

T: +44 (0)1438 767269
F: +44 (0)1438 767317

The Institution of Engineering and Technology, Michael Faraday House, Six Hills Way, Stevenage, SG1 2AY, United Kingdom

From: Karl Leboeuf [mailto:leboeu3@uwindsor.ca]
Sent: Friday, July 06, 2012 7:45 PM
To: journals
Subject: request for written permission to use publication in doctoral thesis

Dear IET,

I am writing to request written permission to include the material in an IET Circuits, Devices & Systems journal paper I have previously published in my doctoral dissertation. The papers is:

"High-speed hardware implementation of a serial-in parallel-out finite field multiplier using reordered normal basis", which was published in IET Circuits, Devices & Systems 2010, Vol. 4, Iss. 2, pp. 168-179.

Please let me know if additional information is required from me in order to process this request.

Thank you for your time.

Karl Leboeuf

-Select Mailbox- My Private Area

Vita Auctoris

Karl Leboeuf was born in 1983, in Windsor, Ontario. He received his B.A.Sc. and M.A.Sc. degrees in electrical engineering in 2006 and 2008 from the University of Windsor, in Windsor, Ontario, Canada. His research interest includes cryptography, VLSI and FPGA hardware design, finite field arithmetic, GPU acceleration, automotive radar digital signal processing, artificial neural networks, and image processing.