OPENCL IMPLEMENTATION OF MONTGOMERY MULTIPLICATION ON FPGA

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

MEHMET UFUK BÜYÜKŞAHIN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

SEPTEMBER 2014

Approval of the thesis:

OPENCL IMPLEMENTATION OF MONTGOMERY MULTIPLICATION ON FPGA

submitted by **MEHMET UFUK BÜYÜKŞAHIN** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen ______________
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Gönül Turhan Sayan ______________
Head of Department, **Electrical and Electronics Engineering**

Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı ______________
Supervisor, **Electrical and Electronics Eng. Dept., METU**

Examining Committee Members:

Prof. Dr. Semih Bilgen
Electrical and Electronics Engineering Department, METU ______________

Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı
Electrical and Electronics Engineering Department, METU ______________

Prof. Dr. Gözde B. Akar
Electrical and Electronics Engineering Department, METU ______________

Assoc. Prof. Dr. Ece Güran Schmidt
Electrical and Electronics Engineering Department, METU ______________

Assoc. Prof. Dr. Alptekin Temizel
Graduate School of Informatics, METU ______________

Date: ______________

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: MEHMET UFUK BÜYÜKŞAHIN

Signature :

ABSTRACT

OPENCL IMPLEMENTATION OF MONTGOMERY MULTIPLICATION ON FPGA

Büyükşahin, Mehmet Ufuk
M.S., Department of Electrical and Electronics Engineering
Supervisor : Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı

September 2014, 79 pages

Galois Field arithmetic has been used very frequently in popular security and error-correction applications. Montgomery multiplication is among the suitable methods used for accelerating modular multiplication, which is the most time consuming basic arithmetic operation. Montgomery multiplication is also suitable to be implemented in parallel.

OpenCL, which is a portable, heterogeneous and parallel programming framework, is recently supported by a major FPGA vendor, Altera. Therefore it is now possible to exploit the advantages of using both FPGA and C based OpenCL language simulataneously.

In this thesis, Montgomery multiplication algorithm is implemented on FPGA using OpenCL programming language. Performance of the proposed FPGA implementation is evaluated and compared with CPU and GPU platforms. Using different OpenCL specific directives, several FPGA configurations corresponding to different parallel architectures are implemented for different multiplication sizes.

Keywords: Parallel Programming, OpenCL on FPGA, Montgomery Multiplication

ÖZ

OPENCL İLE FPGA ÜZERİNDE MONTGOMERY ÇARPIMININ GERÇEKLENMESİ

Büyükşahin, Mehmet Ufuk
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi : Doç. Dr. Cüneyt F. Bazlamaçcı

Eylül 2014 , 79 sayfa

Galois alanı aritmetiği, popüler güvenlik ve hata düzeltme uygulamaları içinde sıklıkla kullanılmaktadır. En çok zaman alan temel arithmetik operasyonu olan modüler çarpma işlemi için de Montgomery çarpma işlemi uygun metotlar arsındadır. Montgomery çarpması paralellel gerçekleştirme için de uygundur.

Taşınabilir, heterojen ve paralel programlama çerçevesi sunan OpenCL, artık önemli FPGA üreticisi Altera tarafından desteklenmektedir. Böylece, uygulamalarda hem FPGA'in hem de C tabanlı OpenCL dilinin avantajlarından beraberce yararlanmak mümkündür.

Bu tez çalışmasında, Montgomery algoritması OpenCL programlama dili ile FPGA üzerinde gerçeklenmiştir. Önerilen FPGA gerçeklemesinin başarımı, CPU ve GPU platformları ile karşılaştırılmıştır. OpenCL'e özel direktiflerle, farklı paralel yapılar çeşitli çarpma boyutları için gerçeklenmiştir.

Anahtar Kelimeler: Paralel Programlama, FPGA üzerinde OpenCL, Montgomery Çarpma

To my family and friends

ACKNOWLEDGMENTS

First of all, I would like to express my gratitude to my family for their love, support and patience over the years.

I sincerely thank my supervisor Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı for all his guidance and support throughout my study.

I would like to thank to my employer, ASELSAN for supporting me.

I also express my gratitude to TÜBİTAK BİDEB "National Scholarship Program for MSc Students".

Finally, I would like to thank my friends Nusret Bayhan, Cem Tarhan and Ömer Alper Özkan for their support and letting me use their computers.

Last but not least, I would like to thank Gizem Kocalar for her support throughout the study.

TABLE OF CONTENTS

ABSTRACT v

ÖZ vi

ACKNOWLEDGMENTS viii

TABLE OF CONTENTS ix

LIST OF TABLES xiii

LIST OF FIGURES xiv

LIST OF ALGORITHMS xvi

LIST OF ABBREVIATIONS xvii

CHAPTERS

1 INTRODUCTION 1

2 BACKGROUND 7

2.1 Mathematical Background 7

2.1.1 Galois Field 7

2.1.2 Galois Field Arithmetic 7

2.1.2.1 Addition in $GF(2^m)$ 8

2.1.2.2 Multiplication in $GF(2^m)$ 9

2.2 Development Environment and Tools 10

2.2.1 FPGA 10

2.2.1.1 FPGA as Computation Unit 10

2.2.1.2 CPUs in the FPGA 11

2.2.1.3 FPGA as an OpenCL Device 11

2.2.2 OpenCL 12

2.2.3 OpenCL on FPGA vs. GPU 13

2.2.4 Development Environment 15

2.2.4.1 Host Application Development 16

2.2.4.2 Kernel Development 16

3 RELATED WORK 19

3.1 Multiplication Algorithms 19

3.1.1 Karatsuba Multiplication 19

3.1.2 Karatsuba Multiplication in $GF(2^m)$ 20

3.1.3 Montgomery Multiplication in $GF(2^m)$ 21

3.1.3.1 Parallel Implementations of Montgomery Multiplication 23

3.1.3.2 Partitioning of Separated Operand Scanning (SOS) Method 24

3.1.3.3 Partitioning of Coarsely Integrated Operand Scanning (CIOS) Method 26

3.1.4 Multiplication in $GF(2^m)$ using Residue Number System (RNS) 27

3.2 FPGA Implementations of Various Galois Field Multipliers . 28

3.2.1 Logic Level Designs 28

3.2.2 Soft Processor Designs 29

3.3 GPU Implementations 31

3.4 Other Multi-core Solutions 32

3.5 Software Solutions 33

4 IMPLEMENTATION AND EVALUATION OF MONTGOMERY MULTIPLICATION ON FPGA USING OPENCL 35

4.1 Preliminary Calculations 35

4.2 Extended Euclidean Algorithm 36

4.3 Implementation 36

4.3.1 Inputs and Outputs 38

4.3.2 Kernel Attributes 40

4.3.2.1 FPGA Resource Usages 40

4.3.2.2 Kernel Frequencies 41

4.3.2.3 Kernel Performances 43

4.3.3 Primitive Sizes 47

4.3.4 Offline Compilation 50

4.4 Functional Testing 52

4.4.1 Reference Results 52

4.4.2 Benchmarks and Profiling 53

5 COMPARISON OF OPENCL FPGA MONTGOMERY MULTIPLIER WITH GPU AND CPU PLATFORMS 59

5.1 Comparison with GPU 59

5.2 Comparison with CPU 62

5.3 Comparison with other Implementations in the Literature . . 64

6 CONCLUSION 67

REFERENCES 71

APPENDICES

A CONSTANT IRREDUCIBLE POLYNOMIALS (N) USED IN REDUCTION 75

A.1 Multiplication size = 256 75

A.2 Multiplication size = 512 75

A.3 Multiplication size = 1024 75

A.4 Multiplication size = 2048 76

A.5 Multiplication size = 4096 76

A.6 Multiplication size = 8192 77

B OPENCL CODE 79

LIST OF TABLES

TABLES

Table 4.1 Chosen irreducible polynomials for different multiplication sizes . . 39

Table 4.2 Comparison of area utilizations (in chip's resource percentage) of 1024 bit Montgomery SOS algorithm implementations for different kernel attributes 41

Table 4.3 Comparison performances (in multiplications per second) for different kernel attributes 44

Table 4.4 Multiplication size vs. work group sizes 47

Table 5.1 Specifications of GPUs tested 60

Table 5.2 Specifications of CPUs tested 63

Table 5.3 Comparison of performance of multiplier implementations in the literature. 65

LIST OF FIGURES

FIGURES

Figure 1.1 Tren in CPUs in terms of clock speed and number of cores [1] 3

Figure 2.1 Overview of of OpenCL architecture [1] 12

Figure 2.2 Simple OpenCL code mapped into custom logic [2]. 14

Figure 2.3 Branching in SIMD structure vs. pipeline structure [2] 14

Figure 2.4 Multiple compute units vs. SIMD vs. both 18

Figure 3.1 Data flow of Montgomery multiplication (SOS) 25

Figure 3.2 Partitioning of SOS given in algorithm 6. 26

Figure 3.3 An example flow of regular (left) and proposed (right) iterations in [3]. Multiplier sizes are given as KOM_{SIZE} and number of multipliers used is given next to arrows. 29

Figure 3.4 An example implementation of [4] with four soft processor cores. . 31

Figure 4.1 Flow of Montgomery multiplication. 37

Figure 4.2 Comparison of implemented (*unsigned char*) kernel frequencies for different kernel attributes and multiplication sizes 42

Figure 4.3 Comparison of implemented (*unsigned int*) Kernel frequencies for different kernel attributes and multiplication sizes 43

Figure 4.4 Comparison of normalized kernel performances for different kernel attributes (*unsigned char*) 44

Figure 4.5 Comparison of normalized kernel performances for different kernel attributes (*unsigned int*) 45

Figure 4.6 Comparison of Kernel performances for different multiplication sizes (*unsigned char*) 46

Figure 4.7 Comparison of Kernel performances for different multiplication sizes (*unsigned int*) . . . 46

Figure 4.8 Logic resource utilization for the implemented multiplier on FPGA 48

Figure 4.9 Memory utilization for the implemented multiplier on FPGA 49

Figure 4.10 C# application for compilation . . . 50

Figure 4.11 Compilation times for *unsigned char* implementations . . . 51

Figure 4.12 Compilation times for *unsigned int* implementations . . . 51

Figure 4.13 A screenshot of multiplier application written in C# for functional testing . . . 53

Figure 4.14 Benchmark results (in microseconds) obtained by host application for FPGA multiplier . . . 55

Figure 4.15 Benchmark results (in microseconds) obtained by host application for CPU/GPU multiplier . . . 57

Figure 5.1 Performance comparison of FPGA implementation with GPUs . . . 61

Figure 5.2 Performance comparison of FPGA implementation with CPUs . . . 64

LIST OF ALGORITHMS

ALGORITHMS

Algorithm 1 Addition in $GF(2^m)$ 8

Algorithm 2 Multiplication in $GF(2^m)$ 9

Algorithm 3 Host application algorithm 16

Algorithm 4 Karatsuba Multiplication in $GF(2^m)$ 20

Algorithm 5 Montgomery Product 22

Algorithm 6 Montgomery product with SOS method 25

Algorithm 7 Montgomery product with CIOS method 27

Algorithm 8 Montgomery multiplication over trinomial residues 28

Algorithm 9 Extended Euclidean Algorithm 36

LIST OF ABBREVIATIONS

AOC	Altera Offline Compiler
AOCL	Altera OpenCL
API	Application programming interface
ASIC	Application-specific integrated circuit
CPU	Central processing unit
CvP	Configuration via Protocol
CRC	Cyclic redundancy check
DSP	Digital signal processor
FPGA	Field-programmable gate array
GF	Galois Field
GPU	Graphics processing unit
JTAG	Joint Test Action Group, IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture
LUT	Lookup table
NIST	National Institute of Standards and Technology
OpenCL	Open Computing Language
PCIe	PCI Express (Peripheral Component Interconnect Express)
RNS	Residue number system
SDK	Software development kit
SIMD	Single instruction multiple data
SoC	System on chip

CHAPTER 1

INTRODUCTION

Computers, laptops, mobile devices became very important in our daily life. Most of the time those devices are connected to each other via Internet. One can watch video, listen to music, do some work, store pictures, communicate with relatives/friends, etc. and possibilities are unlimited. It is also possible to work on sensitive or even confidential information. For example, on-line banking can save time and money by eliminating the need to drive to an ATM, which is available in a 7/24 fashion.

However, all those possibilities would be practical if they are reliable. No one would like sensitive information, pictures/videos of memories to be lost or even worse, stolen. Therefore reliability and security are very important topics for our daily used devices. Security is a very important issue especially for connected devices. Reliability and security attributes have to be provided in a fast and efficient manner.

Reliability: Some extra information is used in order to increase reliability in an application. This redundant information is often called error-correcting codes. There are many methods for generating and decoding of error-correcting codes [5]. Cyclic redundancy check (CRC) is one of the most commonly used error-detecting method. CRC is used in basic hardware, mobile networks, dvd/blu-ray players, hard drives, Internet communication, etc.[1].

Security: Sensitive information should be protected against eavesdropping. It might not always be possible to prevent eavesdropping, especially when the transfer medium

[1] Some examples: CRC-1, parity bit in basic hardware; CRC-6-CDMA2000, CRC-10-CDMA2000, CRC-12-CDMA2000, CRC-16-CDMA2000, CRC-30 in mobile networks; CRC-32 in Ethernet/Internet communication, Reed-Solomon coding in storage

is air as in wireless communication. Data can also be transferred through unknown networks as in the Internet. Therefore, information is encrypted into some other form such that unwanted third-parties cannot understand it. Some widely used cryptographic systems are RSA[2], ECC (Elliptic Curve Cryptography), Schnorr signature, PGP (Pretty Good Privacy), AES (Advanced Encryption Standard), DES (Data Encryption Standard), etc.

Speed: Both redundant information for reliability and encryption/decryption for security are required to protect sensitive information. Unfortunately, both operations are computationally time consuming and are complex problems. No one would like to have huge delays in secure communication. Long waiting times during on-line secure banking/shopping that drains battery or secure but slow-motion DVD movie, etc. are not generally acceptable. Therefore, developing efficient algorithms in this domain is crucial.

Many of those algorithms such as RSA, ECC, Reed-Solomon coding, CRC etc. have mathematical basics on Galois Field. Therefore, it is important to accelerate operations in Galois Field to improve overall performance in such cryptographic and checksum algorithms. Thus Galois Field multiplication, which is one of the most time consuming operations in Galois Field, is a very hot topic.

Unfortunately, these Galois Field operations are computationally intensive calculations. Therefore, usually custom hardware solutions are preferred. Alternatively, parallel processing capabilities of recent processing devices with high processing power can be utilized.

Parallel programming has always been a trendy topic and still is in recent years. This is mainly due to the fact that major CPU manufacturers tend to produce CPUs that have more cores rather than having higher clock frequencies to increase performance as shown in Figure 1.1.

[2] RSA: initials of surnames of inventors, Ron **R**ivest, Adi **S**hamir, and Leonard **A**dleman

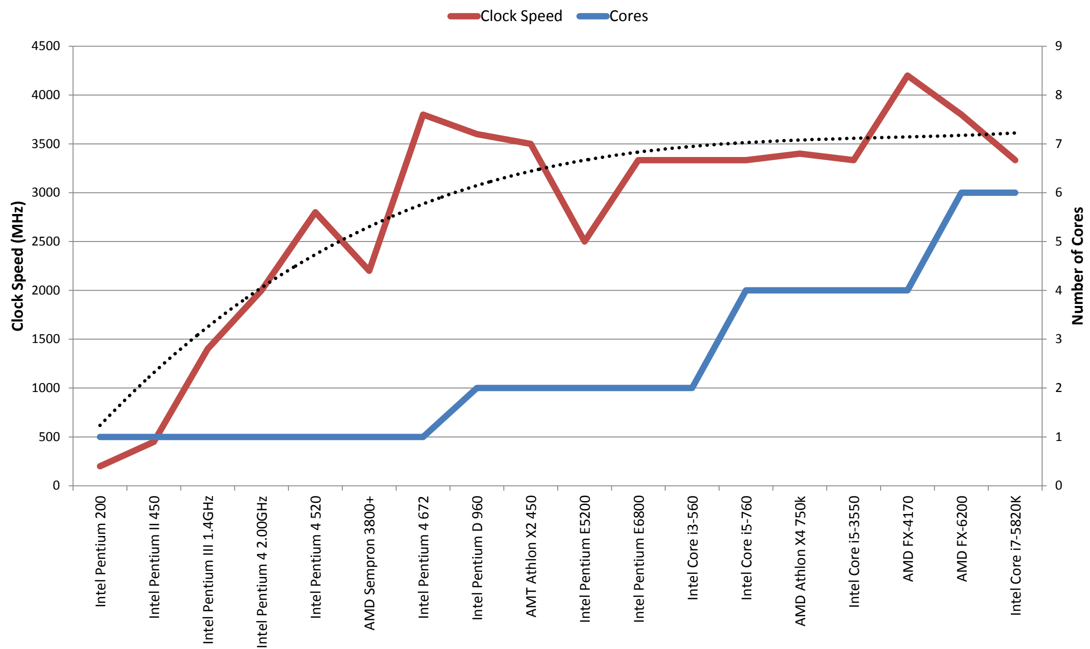


Figure 1.1: Tren in CPUs in terms of clock speed and number of cores [3]

This trend also leads graphics processing units (GPUs) to be used in general purpose programming. Results are quite promising since GPUs can offer very high computation power compared to CPUs [6].

Parallelism is offered by FPGAs as well. Additionally, FPGAs provide better flexibility compared to GPUs and CPUs. FPGAs can perform custom and high speed I/O operations. Major FPGA manufacturers introduced advanced I/O capabilities in their products. Modern FPGAs support PCIe, SATA, SAS, 10G-Ethernet, RapidIO, SDI, DDR, QDR connection interfaces [7]. FPGAs are already widely used in the industry to perform special tasks at a low supply cost. However, they lack ease of programming. Therefore development process is longer and usually more complicated than CPU/GPU development.

With recent improvements in FPGA development tools, major FPGA manufacturers start offering OpenCL support on their FPGAs. Altera already supports OpenCL on FPGA development platforms starting from Quartus 13.0sp1. The other major company Xilinx will also provide OpenCL support in their 2014.1 version Vivado

[3] Sources: ark.intel.com/products and www.amd.com/en-us/products accessed on 13 September 2014

software [8]. Therefore, developers can now benefit from both flexibilities of FPGA and easy programming of CPU/GPU by utilizing the C based programming language OpenCL.

As a consequence, OpenCL based FPGA designs may allow very promising solutions to existing or new problems. Hence, a Galois Field multiplier design on FPGA using OpenCL is potentially an efficient and worth to investigate solution approach.

There are several methods existing for Galois Field multiplication. Trivial methods like look-up table or old-school multiplication followed by long division are not suitable for large fields such as order of hundreds. Therefore, more complicated algorithms such as Karatsuba-Ofman, RNS (Residue number system) and Montgomery are developed.

Karatsuba-Ofman algorithm, actually, just divides large numbers into smaller pieces. It introduces a tree-like structure for multiplication. Unfortunately, it lacks the reduction part. So reduction must be implemented separately after multiplication. RNS algorithm provides easy method for multiplication that is highly parallel by design. RNS also lacks reduction operation and additionally it requires computationally intensive forward and backward transitions. Montgomery algorithm, on the other hand, includes both multiplication and reduction operations. It can be parallelized as well. Therefore, Montgomery algorithm is selected to be implemented on FPGA using OpenCL.

This thesis covers evaluation of Montgomery multiplier implementation on FPGA using OpenCL. Effects of OpenCL specific Kernel attributes have been investigated and performances of FPGA, GPU and CPU as computation platform have been compared.

Evaluation hardware is Nallatech P385-d5[4]. The board is connected to the host system via PCIe bus and includes an Altera Stratix V GS D5 FPGA and 8 GB on board DDR3. Altera Stratix V GS D5 FPGA has 457K logic elements, 690K registers, 28 fractional PLLs, 3550 18x18 multipliers, 1775 27x27 multipliers. However, hardware multipliers are not utilized due to mathematical differences in integer multiplication and Galois Field multiplication. Multiplication modules are coded in OpenCL. Code

[4] Details: http://www.nallatech.com/images/stories/product_briefs/openclcardspb_v1_5.pdf

is compiled using Altera OpenCL SDK with Quartus II version 13.0 service pack 1. Host application is based on Nallatech Hello World example and coded in Visual Studio 2010.

The thesis is organized as follows: First, a brief mathematical background on Galois Field arithmetic and on OpenCL framwork is given in Chapter 2. An introduction to OpenCL development environment is also given in this chapter. Chapter 3 presents a literature survey of the related work on Galois Field multiplication. Some sample implementations are briefly summarized in Chapter 3. Chapter 4 describes the implementation details of the OpenCL Montgomery multiplier on FPGA. In addition, Chapter 4 includes FPGA test results and our observations. Comparison of the performances of the implemented code on FPGA, GPU and CPU platforms and also comparison with the performances of previous works are presented in Chapter 5. Finally, Chapter 6 summarizes and concludes the thesis work.

CHAPTER 2

BACKGROUND

2.1 Mathematical Background

2.1.1 Galois Field

Galois field is a finite set of numbers with some special mathematical properties so that defined operations always results in the set.

It is required that defined operations must satisfy fixed axioms, associativity, commutativity and distributivity rules over its elements. Additionally, any element must have a unique additive inverse and any non-zero element should have a unique multiplicative inverse [9].

2.1.2 Galois Field Arithmetic

Easiest example would be on prime Galois fields, $GF(p)$. So arithmetic is identical to regular integer addition and multiplication with modulo prime p. Some examples in $GF(3)$ where elements are $(0, 1, 2)$ are as follows:

$$0 + 0 = 0 \qquad 1 + 2 = 0 \qquad 2 + 2 = 1 \ \ (2 + 2 \equiv 1 \pmod 3)$$
$$0 \times 0 = 0 \qquad 1 \times 2 = 2 \qquad 2 \times 2 = 1 \ \ (2 \times 2 \equiv 1 \pmod 3)$$

In order $GF(p)$ to form a field, p must be a prime, otherwise some elements might not have a unique multiplicative inverse. For instance, there is no x value satisfying $2 \times x = 1 \mod 6$.

We can also use vectors to enhance the field, so we can use $GF(m)$ such that $m = p^n$ where p is a prime and n is an integer. Furthermore, given $n > 1$, finite field $GF(p^n)$ can be represented as the field of equivalence classes of polynomials in which coefficients are in the field $GF(p)$. Therefore the elements of $GF(p^n)$ can be represented by polynomials with degree less than n [10].

Moreover, addition in a vector field is relatively easier than addition of integers because integer addition has carry. Since there is no carry generated in vector addition, all computations are guaranteed to be in the finite set. Here are some examples of addition operation in $GF(2^3)$:

$$000 + 000 = 000 \qquad 010 + 100 = 110 \qquad 011 + 110 = 101$$

Multiplication, on the other hand, could lead to larger results that do not fit into finite space [11]. Therefore multiplication includes one more step referred as reduction. Actually, GF multiplication is done by first doing a regular multiplication using carryless additions, then by dividing the result with the reduction polynomial and by noting the remainder as the final result. Therefore, both p, n in $GF(p^n)$ and the reduction polynomial must be known in order to carry out multiplication in GF.

Multiplication with randomly chosen reduction polynomials can be very costly and ineffective in certain operations. Therefore, National Institute of Standards and Technology (NIST) has chosen several polynomials which are optimized for the efficiency of the elliptic curve operations [12].

2.1.2.1 Addition in $GF(2^m)$

Addition is very easy for computers since it is only an exclusive or (XOR) operation for each bit (see algorithm 1).

Algorithm 1: Addition in $GF(2^m)$

Input : $a(x) = \sum_0^{m-1} a_i x^i$ and $b(x) = \sum_0^{m-1} b_i x^i$
Output: $c(x) = \sum_0^{m-1} c_i x^i = a(x) + b(x)$

```
1 Procedure Sum(a(x), b(x))
2     for i = 0 to m − 1 do
3         c_i = a_i ⊕ b_i
4     return c(x)
```

2.1.2.2 Multiplication in $GF(2^m)$

Straight and old fashioned school multiplication followed by a long division is very costly in GF multiplication. Since GF multiplication is widely used in the core of many applications such as cryptography/security applications, it is crucial to have a fast and efficient multiplier.

A simple method for multiplication is shift-and-add [9].
Given,

$$a(x) = \sum_{0}^{m-1} a_i x^i \qquad b(x) = \sum_{0}^{m-1} b_i x^i \qquad c(x) = \sum_{0}^{m-1} c_i x^i \qquad f(x) = \sum_{0}^{m-1} f_i x^i$$

$$\begin{aligned} c(x) &= a(x) \cdot b(x) \mod f(x) \\ &= \big(a_{m-1}x^{m-1}b(x) + \cdots + a_2x^2b(x) + a_1xb(x) + a_0b(x)\big) \mod f(x) \end{aligned} \tag{2.1}$$

We observe that Equation 2.1, iterating through i (on a_i), calculates $x^i b(x) \mod f(x)$ and accumulates the result if a_i is non-zero.

$$\begin{aligned} b(x)x &= \big(b_{m-1}x^m + b_{m-2}x^{m-1} + \cdots + b_2x^3 + b_1x^2 + b_0x\big) \mod f(x) \\ &= b_{m-1}x^m \mod f(x) + \big(b_{m-2}x^{m-1} + \cdots + b_2x^3 + b_1x^2 + b_0x\big) \mod f(x) \\ &= b_{m-1}r(x) + \big(b_{m-2}x^{m-1} + \cdots + b_2x^3 + b_1x^2 + b_0x\big) \mod f(x) \end{aligned} \tag{2.2}$$

Therefore, $b(x)x \mod f(x)$ can be calculated iteratively by a shift operation and then adding $r(x) = x^m \mod f(x)$ to $b(x)$ if the most significant bit, b_{m-1}, is 1 (see algorithm 2).

Algorithm 2: Multiplication in $GF(2^m)$

```
Input : a(x) = Σ_0^{m-1} a_i x^i and b(x) = Σ_0^{m-1} b_i x^i
        Reduction Polynomial: f(x) = Σ_0^{m-1} f_i x^i
Output: c(x) = Σ_0^{m-1} c_i x^i = a(x) · b(x) mod f(x)
1  Procedure Multipy(a(x), b(x))
2      if a_0 = 1 then
3          c(x) ← b(x)
4      else
5          c(x) ← 0
6      for i = 1 to m − 1 do          /* i = 0 already processed */
7          b(x) ← b(x)x mod f(x)
8          if a_i = 1 then
9              c(x) ← c(x) ⊕ b(x)
10     return c(x)
```

2.2 Development Environment and Tools

2.2.1 FPGA

A field-programmable gate array (FPGA) is a large integrated circuit that can be configured to perform specific tasks. Although FPGAs have flexible structure, they can offer quite high computation power because they perform calculations at the gate level. Nowadays, FPGAs are very rich in terms of resources. It is possible to find a single FPGA chip that includes logic elements up to millions, a large memory up to tens of Mbits, hard peripheral blocks/transceivers and hard computation units such as multiple CPU cores, DSP cores, thousands of multipliers, etc. [13] [14].

FPGAs require special equipment for programming and a very common way is using a JTAG connection. Reprogramming of an FPGA completely restarts the device. This is not a problem for most of the times because it happens at system start-up. However, this may not be so practical for some applications such as an application where FPGA is connected as a PCIe device to the host computer. Any change in FPGA requires complete reprogramming and hence PCIe core requires to be restarted. That makes the FPGA inaccessible by the host until a complete restart of the host system. The solution is to use partial reconfiguration. Partial reconfiguration, as the name implies, allows FPGA to be programmed partially. Moreover, this method is further enhanced to use PCIe connection to eliminate the need for special equipment. This is called as configuration via protocol (CvP) by Altera [15].

2.2.1.1 FPGA as Computation Unit

FPGAs are massively parallel processors by design. Unlike sequential C programs, it could be very hard and time consuming to design, debug and verify an FPGA system. Moreover, compilation time may easily exceed several hours. Also, abstraction is limited because a programmer may need to consider very low level hardware related issues such as timing.

2.2.1.2 CPUs in the FPGA

In order to speed up the design process, major FPGA manufacturers introduced soft processors. Altera named their soft processor as NIOS II and Xilinx named theirs as MicroBlaze. Soft processor is basically a simple CPU core using logic resources of the FPGA. With the increasing number of logic elements in an FPGA, it is even possible to implement many soft CPU cores in a single chip. Moreover, major FPGA vendors started manufacturing chips including single or many hard CPU cores [1] [16] [17].

2.2.1.3 FPGA as an OpenCL Device

Recently, Altera released a high performance computation solution using a Stratix V FPGA as an OpenCL device. Therefore, an FPGA can be used as a parallel computing device similar to a GPU. Moreover, the solution provides a highly customizable architecture that regular GPUs do not have. Additionally, this solution may decrease power consumption dramatically while increasing the throughput compared to CPU or GPU based solutions [18].

OpenCL is a portable programming language, meaning that applications can run on different hardware. However, it may be very time consuming to optimize an OpenCL code for different brands/models of GPU hardware. Therefore, migration is easy from functional point of view but may be hard is an efficient migration is desired. Since GPU hardware is fixed and varies a a lot among brands/models, a code optimized for "GPU-A" must be re-optimized manually for "GPU-B". FPGA hardware on the other hand can adapt itself to a specific piece of code. Therefore, in an FPGA solution, which includes an optimal hardware-software co-design, migration to another FPGA just becomes re-compilation of the code. For instance, work group sizes should be optimized depending on number of processing units in the GPU in order to maximize utilization. However, FPGA implementation will generate required number of cores during compilation.

[1] Xilinx released Zynq-7000 series with dual ARM Cortex-A9 based application processor unit with CPU frequency up to 1 GHz [16]. Similarly, Altera has ARM-based hard processor system that utilizes dual-core ARM Cortex-A9 MPCore processor [17].

2.2.2 OpenCL

OpenCL is a framework for parallel programming. Its applications run on heterogeneous platforms consisting of one or more single/multi core CPUs, GPUs, DSPs, FPGAs, and other processing units [19]. It uses the heterogeneous programming model. Operations such as memory management, data transfers to/from devices, queuing tasks to devices, and error management are handled by the host device. It is based on C99 programming language with additional keywords.

OpenCL is maintained by Khronos Group and supported by a variety of companies including Intel, AMD, Qualcomm, IBM, Samsung, Apple, nVidia, Nokia, Altera, Xilinx, ARM, Broadcom, Ericsson, Freescale [19].

An OpenCL application basically has two parts: (i) the main code that runs on the host to prepare and orchestrate the heterogeneous platform and (ii) kernels that run on OpenCL device(s) that perform the actual computation (see Figure 2.1). The application running on the host submits tasks to OpenCL devices.

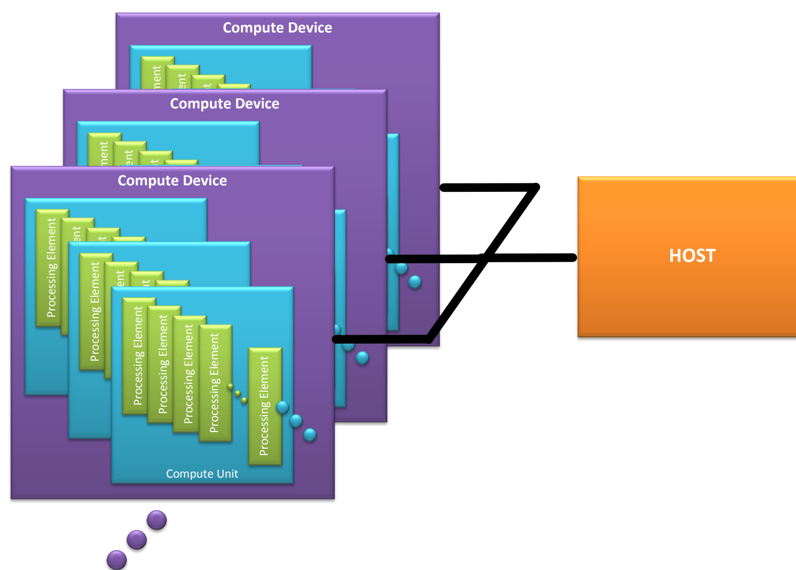


Figure 2.1: Overview of of OpenCL architecture [1]

First, the OpenCL host application (usually a sequential code written in C++) queries and selects computation devices using OpenCL API. Then it manages them using work queues. Kernels written in OpenCL runs on each computation unit in parallel.

There are basically two models in parallelizing a computation task, which are called as data-parallel and task-parallel.

Data-parallel model: A sequence of instructions are executed on a unique element of an array, which are mapped by unique-ids of each processing unit. For example, $\sum_{1}^{NumOfPU} A_i + B_i$ can be calculated at once by each core summing two inputs mapped by ids.

Task-parallel model: Each processing unit can be used independently to execute given task.

Memory management is explicit. Host application must transfer data from host memory to OpenCL devices' memories and then get the results back [19].

Memory regions in OpenCL are differentiated depending on access type and scope.

Global memory can be accessed by all work-items of all work-groups. Access type is read/write.

Local memory can be accessed by all work-items of the same work-group. Access type is read/write.

Constant memory is read-only accessible by all-work items.

Private memory is read/write accessible by individual work-items.

2.2.3 OpenCL on FPGA vs. GPU

Kernel execution is handled differently on FPGA and GPU. GPU consists of many (usually in the order of hundreds) simple SIMD processing units on which work-items are computed instruction-by-instruction. Due to fixed size SIMD architecture same instruction must be executed on a number of processing units. However, each kernel is mapped into a custom dedicated logic on FPGA [2]. All data paths (including conditional paths) in the code are converted into piece of hardware as illustrated in Figure 2.2.

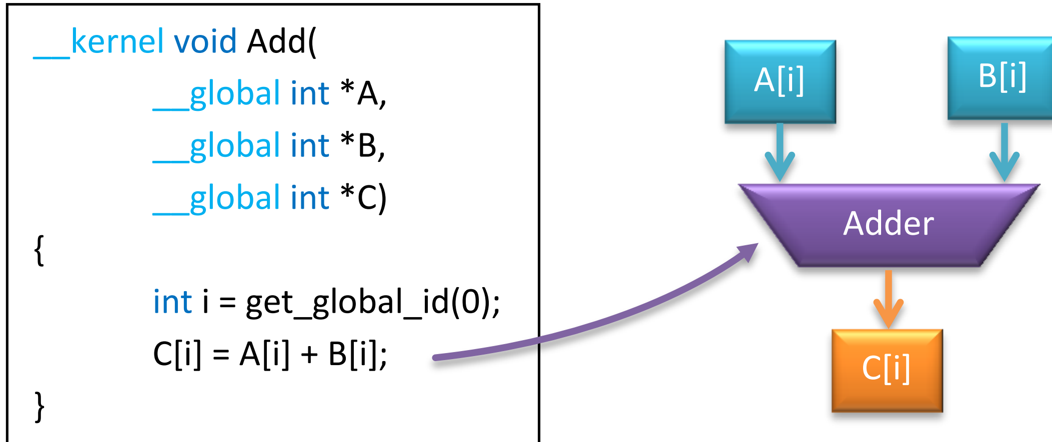


Figure 2.2: Simple OpenCL code mapped into custom logic [2].

FPGA, on the other hand, utilizes pipeline parallelism. Therefore, its branching behavior is different than SIMD, which is usually found in GPUs. Because of having SIMD architecture on GPU, following different paths across work items after a branching would cause idle times in the process. This is because only a single instruction can be executed at a time [2]. On the other hand, programmers are not faced with this issue on FPGA. Because all possible branchings are already built into the custom hardware and any path could be followed at any place of the pipeline. A simple behavior of pipelined FPGA and SIMD GPU are compared in Figure 2.3 where all three work-items first execute A and then B, C and D stages are executed conditionally.

```
...
A();
if( COND1 ) { B1(); C1(); }
else if( COND2 ) { B2(); C2(); }
else if( COND3 ) { B3(); C3(); }
// poor coding, just an example
...
```

SIMD Parallelism	A	B_1	C_1	IDLE			
	A	IDLE		B_2	C_2	IDLE	
	A	IDLE				B_3	C_3
Pipeline Parallelism	A	A	A				
		$B_{1,2,3}$	$B_{1,2,3}$	$B_{1,2,3}$			
			$C_{1,2,3}$	$C_{1,2,3}$	$C_{1,2,3}$		

Figure 2.3: Branching in SIMD structure vs. pipeline structure [2]

Scenario: First execute A on all CUs (compute units) then conditionally execute B_i and C_i on ith CU.

- GPU
  1. Execute A on SIMD processor in parallel. A is common for all Kernels.
  2. Execute B_1C_1 which are conditionally executed on CU 1.
  3. Execute B_2C_2 which are conditionally executed on CU 2.
  4. Execute B_3C_3 which are conditionally executed on CU 3.
- FPGA
  1. Execute A for CU 1.
  2.
     - Execute either B_1, B_2 or B_3 for CU 1 depending on condition.
     - Execute A for CU 2.
  3.
     - Execute either B_1, B_2 or B_3 for CU 2 depending on condition.
     - Execute either C_1, C_2 or C_3 for CU 1 depending on condition.
     - Execute A for CU 3.
  4.
     - Execute either B_1, B_2 or B_3 for CU 3 depending on condition.
     - Execute either C_1, C_2 or C_3 for CU 2 depending on condition.
  5.
     - Execute either C_1, C_2 or C_3 for CU 3 depending on condition.

One other difference between GPU and FPGA exists in the design process, which will be explained in more detail in the following sections. Normally, OpenCL kernels are compiled at runtime for the target device, i.e., for CPU or GPU. Compiling kernels for GPU usually takes just a few seconds. However, FPGA logic synthesis is a computationally intensive work and may take quite a long time. Therefore, Altera provides an offline compiler (Altera Offline Compiler, AOC) to prepare FPGA logic from OpenCL code. So, instead of runtime compilation, Altera just loads prepared FPGA content. At this point, portability of OpenCL can shorten the design process dramatically. It would be a good practice to test OpenCL code first on CPU or GPU, then compile it for FPGA using AOC.

2.2.4 Development Environment

Development of software for OpenCL based Altera FPGA consists of two parts, software focused host application development and hardware focused kernel develop-

ment.

2.2.4.1 Host Application Development

Host application is a C++ program that uses OpenCL API. Host application runs on a regular CPU. It uses OpenCL APIs to manage compute devices as described in algorithm summarized in algorithm 3. Normally kernels (at line 5) would be compiled at runtime for the selected device [20]. However, in FPGA kernels are compiled by AOC [21] and image is loaded by partial reconfiguration (CvP).

Algorithm 3: Host application algorithm

```
1  Function Main()
2      Discover OpenCL devices.
3      Query their capabilities and decide which ones to use.
4      Initiate OpenCL device. Create context, command queue etc.
5      Prepare kernel(s).
6      repeat
7          Allocate memory buffers, prepare kernel arguments.
8          Launch kernel.
9          Collect results.
10     until Application exits
11     Exit program
```

Host application provides timing information for performance measurements. Additionally, host application can perform computations using software libraries to compare results.

2.2.4.2 Kernel Development

OpenCL is a portable solution, which means that the same code could be executed on different hardware. Therefore optimizing kernel for all devices is almost impossible as this task is highly dependent on hardware. On the other hand, an FPGA based OpenCL system could overcome this situation by configuring itself depending on the kernel, i.e., by producing an optimal hardware for each specific problem. So, the developer can only focus on solving the problem in a parallel manner. Therefore, Altera offers Altera SDK for OpenCL, which allows designers to use OpenCL C. Generally, OpenCL Kernels would be compiled at runtime depending on target hardware.

However, Altera solution uses pre-compiled hardware binary file [21].

Data processing efficiency can further be increased by instructing the compiler to use specific architectures. Altera provides several kernel attributes for this kind of customizations, details of which are given in [22]. Some exemples are as follows:

`#pragma unroll` tells AOC to try to unroll loops to decrease the number of iterations at the expense of increased hardware resource usage. Loop unrolling will fail if loop bounds are not constant, loop contains too much data dependency, or loop is too large so that it does not fit into hardware (in this case `#pragma unroll <N>` could be used to limit unrolling).

`max_work_group_size` instructs compiler to limit work group size. Compiler assumes 256 work-items for work group size by default. Therefore, software requiring smaller work groups would lead unnecessary hardware to be generated when this attribute is not set.

`reqd_work_group_size` is similar to `max_work_group_size` attribute but specifies the exact size per work group to allow further hardware resource optimization.

`num_compute_units` allows the compiler to generate multiple compute units per kernel in order to increase throughput. It increases both global memory bandwidth requirement and hardware resource utilization (see Figure 2.4a).

`num_simd_work_items` is similar to `num_compute_units` but it also requires `reqd_work_group_size` to be specified. It increases throughput by vectorizing kernel, which enables multiple work-items to be processed in SIMD fashion (see Figure 2.4b). Using SIMD compute units usually results in more efficient hardware than using multiple compute units because SIMD compute units only duplicates data paths [22]. Using both could be a better option (see Figure 2.4c). A comparison is depicted in Figure 2.4.

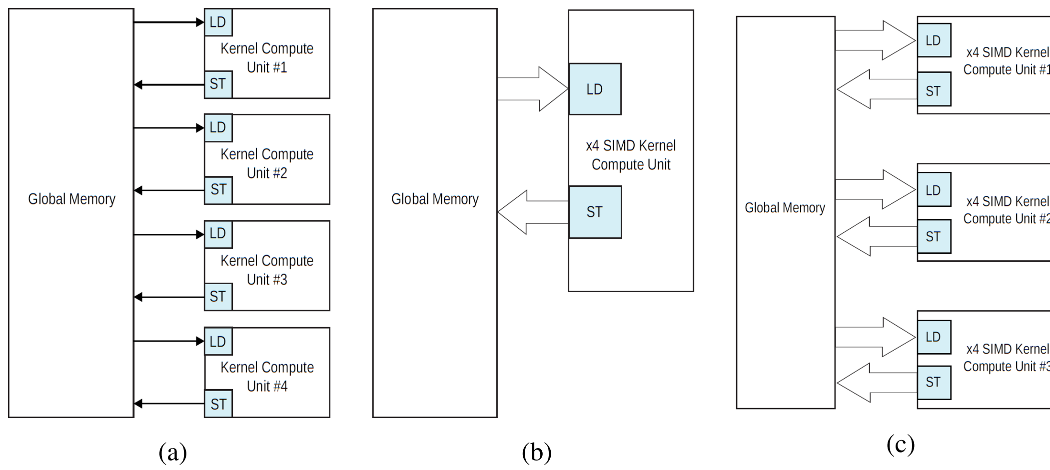


Figure 2.4: (a) Multiple compute units (using `num_compute_units(4)` attribute) (b) Compute unit with multiple SIMD lanes (using `num_simd_work_items(4)` attribute) (c) Multiple compute unit with multiple SIMD lanes (using `num_compute_units(3)` and `num_simd_work_items(4)` attributes).

CHAPTER 3

RELATED WORK

Parallel implementations of various multiplication algorithms exist in the literature. The present chapter gives an overview of a sample of such implementations with a focus on FPGA, GPU and multi-core solutions. The implemented multiplication algorithms are also briefly summarized first.

3.1 Multiplication Algorithms

3.1.1 Karatsuba Multiplication

Karatsuba algorithm was introduced as a general integer multiplication method based on divide and conquer approach. Basic idea is to replace multiplication with less complex addition/subtraction operations [23].

Suppose, we need to calculate $c = ab$. First divide inputs as $a = a_H x^m + a_L$ where $x^m > a_L$ and $b = b_H x^m + b_L$ where $x^m > b_L$. Then,

$$\begin{aligned} c &= ab = (a_H X^m + a_L)(b_H x^m + b_L) \\ c &= c_1 x^{2m} + c_2 x^m + c_3 \\ c &= a_H b_H x^{2m} + (a_H b_L + a_L b_H) x^m + a_L b_L \end{aligned} \tag{3.1}$$

Equation 3.1 would require four multiplications of size $m/2$. On the other hand,

$$\begin{aligned} a_H b_L + a_L b_H &= (a_H + a_L)(b_H + b_L) - a_H b_H - a_L b_L \\ c_2 &= (a_H + a_L)(b_H + b_L) - c_1 - c_3 \end{aligned}$$

Therefore, c_2 can be calculated using one multiplication instead of two and the whole result can be expressed as in Equation 3.2 with three multiplications plus some shift and add operations.

$$c = a_H b_H x^{2m} + ((a_H + a_L)(b_H + b_L) - a_H b_H - a_L b_L)x^m + a_L b_L \tag{3.2}$$

This method is especially useful for recursive multiplication of very large integers using limited size multipliers such as the ones we encounter in modern day processors, which have 32, 64 bit multipliers. For example, Intel's *PCLMULQDQ* instruction uses this method to compute carryless multiplication of large numbers on 64 bit multipliers [24]. Any type of multiplication can be employed at the end of recursion.

3.1.2 Karatsuba Multiplication in $GF(2^m)$

Finite field multiplication is performed in two steps; first one is classic multiplication and the second one is modular reduction as described in algorithm 4 using multipliers of size $N_{MultiplierSize}$ in line 8 [3]. Note that all summations (additions and subtractions) in line 12 are the same operation exor in $GF(2^m)$.

Algorithm 4: Karatsuba Multiplication in $GF(2^m)$

Input : $A(x) = \sum_0^{2^n-1} A_i x^i$ and $B(x) = \sum_0^{2^n-1} B_i x^i$ where n is an integer
Output: $C(x) = \sum_0^{2^n-1} C_i x^i = AB \mod f(x)$ where n is an integer
1 **Function** *Multiply(A, B)*
2 $C_{Partial} = Karatsuba(A, B)$
3 $C = ModularReduction(C_{Partial})$
4 **return** C

Input : $a(x) = \sum_0^{2^n-1} a_i x^i$ and $b(x) = \sum_0^{2^n-1} b_i x^i$ where n is an integer
Output: $c(x) = \sum_0^{2\times 2^n-2} c_i x^i = ab$ where n is an integer
5 **Function** *Karatsuba(a, b)*
6 $N \leftarrow max(degree(a), degree(b))$
7 **if** $N > N_{MultiplierSize}$ **then**
8 **return** $Mult(a, b)$
Let: $a = a_H x^{N/2} + a_L$ and $b = b_H x^{N/2} + b_L$
9 $c_{HH} \leftarrow Karatsuba(a_H, b_H)$
10 $c_{HL} \leftarrow Karatsuba(a_H + b_L, a_L + b_H)$
11 $c_{LL} \leftarrow Karatsuba(a_L, b_L)$
12 **return** $c_{HH} x^N + (c_{HL} - c_{HH} - c_{LL})x^{N/2} + c_{LL}$

Any method can be used for $Mult(a, b)$. Karatsuba algorithm calculates multiplication of two numbers but it does not perform reduction operation, which has to be

performed separately.

3.1.3 Montgomery Multiplication in $GF(2^m)$

Montgomery multiplication, first introduced in 1985, replaces time consuming division and reduction operations in GF multiplication with less costly operations [25].

Suppose we need to calculate $c(x)$ in $GF(2^m)$, i.e.,

$$c(x) = \sum_{0}^{m-1} c_i x^i = a(x)b(x) \mod f(x)$$

Instead of directly working on $a(x)$ and $b(x)$, Montgomery algorithm suggests to use $\bar{a} \equiv aR(x) \mod f(x)$ and $\bar{b} \equiv bR(x) \mod f(x)$ where $R(x)$ is chosen such that $R(x) > f(x)$ and is relatively prime to $f(x)$ (that is $gcd(R(x), f(x)) = 1$). Here, $gcd(a, b)$ stands for greatest common divisor of a and b.

As described in [26], for modulo $f(x) = N$,

$$\begin{aligned} c &= ab \mod N \\ \bar{c} &= cR \mod N \\ &= abR \mod N = (aRbRR^{-1}) \mod N = (\bar{a}\bar{b}R^{-1}) \mod N \\ &= (\bar{a}\bar{b}RR^{-1}/R) \mod N \end{aligned} \tag{3.3}$$

Using identity $RR^{-1} - NN' = 1$,

$$\begin{aligned} \bar{c} &= (\bar{a}\bar{b}(1 + NN')/R) \mod N \\ &= ((\bar{a}\bar{b} + \bar{a}\bar{b}NN')/R) \mod N \end{aligned} \tag{3.4}$$

For any integer k,

$$\begin{aligned} \bar{c} &= ((\bar{a}\bar{b} + \bar{a}\bar{b}NN')/R + kN) \mod N \\ &= ((\bar{a}\bar{b} + \bar{a}\bar{b}NN' + kNR)/R) \mod N \\ &= ((\bar{a}\bar{b} + (\bar{a}\bar{b}N' + kR)N)/R) \mod N \\ &= ((\bar{a}\bar{b} + ((\bar{a}\bar{b}N') \mod R)N)/R) \mod N \end{aligned} \tag{3.5}$$

Notice that, $(\bar{a}\bar{b}N') \mod R < R$ and $\bar{a} < N$, $\bar{b} < N$ therefore $\bar{a}\bar{b} < N^2$. So,

$$(\bar{a}\bar{b} + ((\bar{a}\bar{b}N') \mod R)N)/R < (N^2 + RN)/R$$

Since $R > N$ and addition is xor operation in GF, expression $\bar{a}\bar{b}+((\bar{a}\bar{b}N') \mod R)N)/R$ will always be less than N. Therefore $\mod N$ in the last line of Equation 3.5 has no effect for GF. Therefore $\bar{c} = \bar{a}\bar{b} \mod N$ can be calculated by algorithm 5.

Algorithm 5: Montgomery Product

Given : R a power of 2, $R > N$ and $gcd(R, N) = 1$
N' such that $RR^{-1} - NN' = 1$
Input : $\bar{a} \equiv aR \mod N$
$\bar{b} \equiv bR \mod N$
Output: $\bar{c} \equiv cR \mod N$
1 **Function** *MontProd*($\bar{a}, \bar{b}$)
2 $t = \bar{a}\bar{b}$
3 $\bar{c} = (t \oplus (tN' \mod R)N)/R$
4 **return** $\bar{c}$

Modulo and division by R are both easy operations for computers as R is a power of 2. However, switching to $N\ residue$ and computation on N' are costly. Therefore, this algorithm is more appropriate for operations where several multiplication products are required (i.e. exponentiation) with the same modulus [27].

Montgomery algorithm is also valid for integers. However, Montgomery for integers has one more step in the end. Expression $\bar{a}\bar{b}+((\bar{a}\bar{b}N') \mod R)N)/R$ in Equation 3.5 is smaller than $2N$ for integers. Therefore, additional subtraction operation is required if expression is larger than N. Following example illustrates Montgomery algorithm for integer multiplication:

Example:

Suppose we would like to perform $25 \times 53 \mod 97$, take $R = 100$

So, pre-calculated constants $R^{-1} = 65 \mod 97$ and $RR^{-1} - NN' = 1 \Rightarrow N' = 67$

First, transforms inputs into Montgomery domain,

$$a = 25 \qquad \bar{a} = 25 \times 100 \mod 97 = 75$$
$$b = 53 \qquad \bar{b} = 53 \times 100 \mod 97 = 62$$

Then, perform Montgomery reduction as given in algorithm 5.

$$
\begin{aligned}
t &= \bar{a}\bar{b} = 75 \times 62 = 4650 \\
\bar{c} &= (4650 + (4650 \times 67 \mod 100) \times 97)/100 \\
&= (4650 + (311550 \mod 100) \times 97)/100 \\
&= (4650 + (50 \times 97)/100) \\
&= (4650 + 4850)/100 \\
&= 9500/100 \\
\bar{c} &= 95
\end{aligned}
$$

Finally, convert result back from Montgomery domain.

$$
\begin{aligned}
c &= \bar{c}R^{-1} \mod 97 = 95 \times 65 \mod 97 \\
c &= 64 = 25 \times 53 \mod 97
\end{aligned}
$$

As illustrated in the example, Montgomery method converts expensive modulo operation with less costly divisions and modulo operations. For computers those division and modulo operations will become just shifting and neglecting.

Montgomery reduction method is not suitable for single multiplication due to forward and backward conversions. However for repeated multiplications like exponentiation, it is very useful. Because, all computations can remain in Montgomery domain. Therefore only initial and final domain conversions would be enough.

3.1.3.1 Parallel Implementations of Montgomery Multiplication

Montgomery based multiplication is performed when modulo multiplication of two s-word numbers is required, where s is relatively large and a multiplier hardware exits for multiplying word size numbers. Montgomery based algorithms can basically be categorized in terms of two factors [27]:

1. whether multiplication and reduction stages are integrated[1] or separate, and
2. whether the algorithm loops on operand's words or product's words.

[1] called as *finely* when reduction is performed just after a word or *coarsely* when reduction is performed on an array of words

Separated Operand Scanning (SOS): Multiplication and reduction steps are separated in this technique. First, 2s-word product of two s-word integers is calculated and then reduction is performed to obtain the final result.

Coarsely Integrated Operand Scanning (CIOS): Unlike SOS, this methods switches between multiplication and reduction in the loop, therefore directly producing the s-word final result instead of computing 2s-word complete product.

Finely Integrated Operand Scanning (FIOS): This method unrolls nested loops of CIOS into singe loops to perform reduction word by word.

Finely Integrated Product Scanning (FIPS): This method loops on final product's words. It would be beneficial for microprocessors as most of the read, write operations are on accumulator words which would most likely be placed in registers.

Coarsely Integrated Hybrid Scanning (CIHS): This is similar to SOS but requires less space by hybrid design. The method mixes product scanning and operand scanning.

There are many studies on Montgomery multiplication algorithm. Some of them are [27], [28], [29], [4], [30].

3.1.3.2 Partitioning of Separated Operand Scanning (SOS) Method

SOS Methodology is given in algorithm 6, which is a more detailed version of algorithm 5 for modular multiplication of multi word numbers. It is observed that the most time consuming parts are the inner loops at line 4 and line 8. Therefore parallelization

should be targeted in these loops. Additionally, data flow is given in Figure 3.1.

Algorithm 6: Montgomery product with SOS method

```
   Input  : s-word operands A, B; an odd modulus N.
            Constant n' = -n_0^{-1} mod 2^w where w is the word length
   Output: s-word C = AB mod N
 1 Procedure MontMultSOS(A, B)
 2     for i = 0 to s - 1 do
 3         for j = 0 to s - 1 do
 4             t[i + j] = t[i + j] + a[j] × b[i]
       // Multiplication is OK. Now, reduction.
 5     for i = 0 to s - 1 do
 6         m_i = t[i] × n' mod 2^w
 7         for j = 0 to s - 1 do
 8             t[i + j] = t[i + j] + m_i × n[j]
 9     C = (t_{s-1}, ··· , t_1, t_0)                // Lower s-word of t
10     return C
```

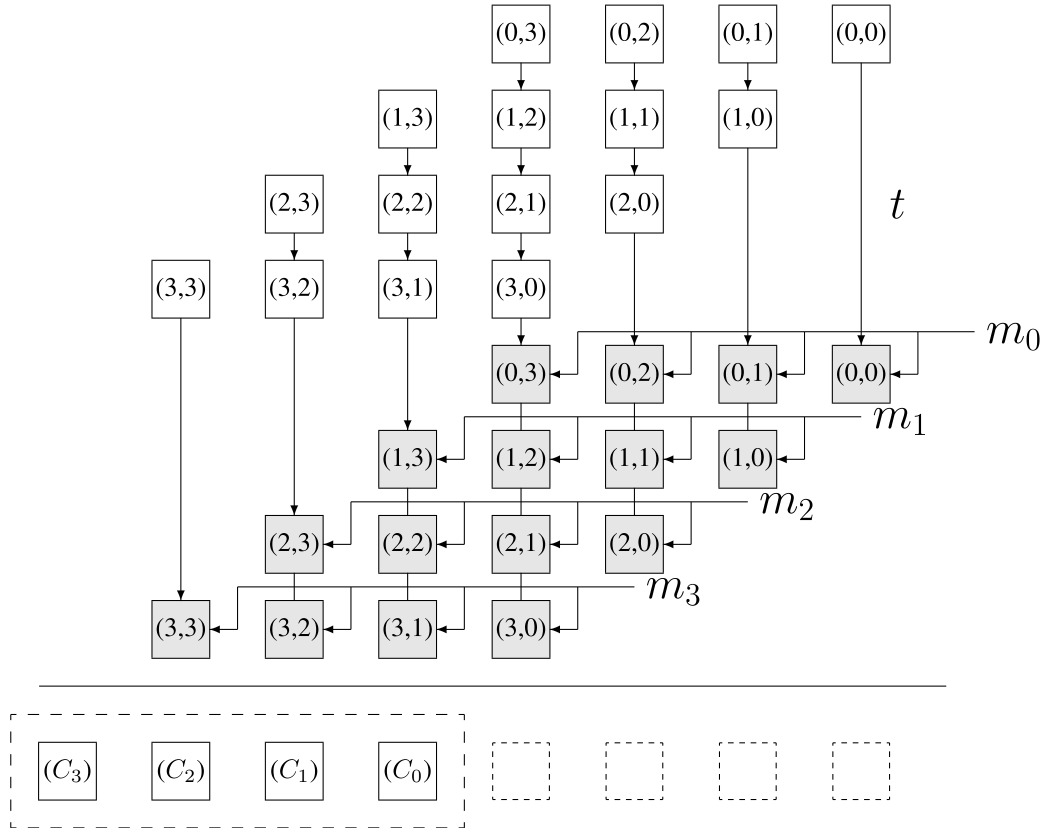


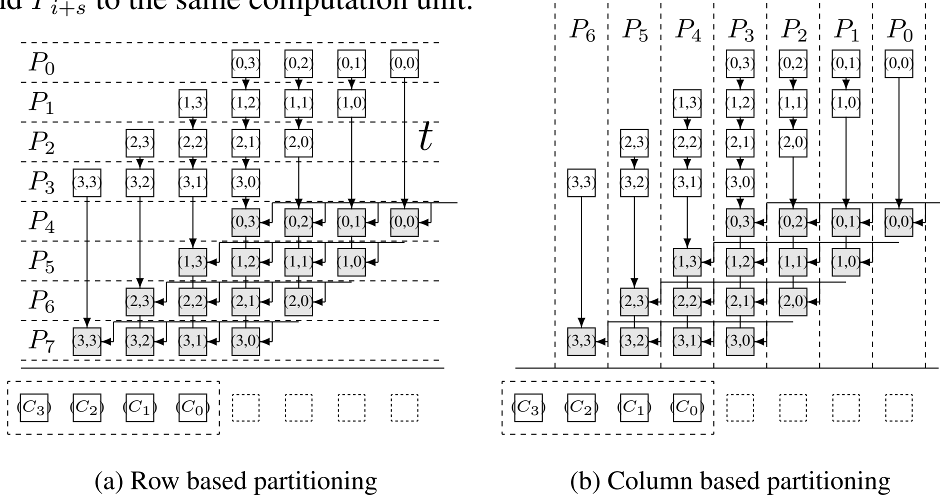
Figure 3.1: Data flow of Montgomery multiplication (SOS) given in algorithm 6. White boxes show multiplication stages and gray boxes represent reduction steps.

With the analysis of data flow given in Figure 3.1, it is observed that there are basically

two options for partitioning, which can be row based or column based. Both methods are illustrated in Figure 3.2, where P_i denotes the i-th partition.

Row Based Partitioning: Row based partitioning is depicted in Figure 3.2a. One can easily observe that it has perfect task balancing since all partitions have equal number of boxes. However, t must be transferred between partitions, which introduces communication overhead.

Column Based Partitioning: Column based partitioning is depicted in Figure 3.2b. First of all, it has better communication overhead compared to row based partitioning. Only m_i terms are transferred. On the other hand, it could be argued that it does not have a good task balancing. However, tasks can be balanced by computing for example P_0 and P_4 on same computation unit. This case can be generalized by assigning P_i and P_{i+s} to the same computation unit.



(a) Row based partitioning

(b) Column based partitioning

Figure 3.2: Partitioning of SOS given in algorithm 6.

3.1.3.3 Partitioning of Coarsely Integrated Operand Scanning (CIOS) Method

Instead of calculating the whole product first and than reducing it as in algorithm 6, CIOS combines two large loops at line 2 and line 5 into a single loop. Because calculation of m at line 6 depends only on i. Then algorithm 6 becomes algorithm 7.

This method requires less amount of temporary memory space compared to SOS method. Therefore, it is suitable for GPU implementations since most variables could

fit into GPU registers [28].

Algorithm 7: Montgomery product with CIOS method

```
   Input  : s-word operands A, B; an odd modulus N.
            Constant n' = -n_0^{-1} mod 2^w where w is the word length
   Output: s-word C = AB mod N
1  Procedure MontMultSOS(A, B)
2      for i = 0 to s - 1 do
3          for j = 0 to s - 1 do
4              t[i + j] = t[i + j] + a[j] × b[i]
5          m = t[i] × n' mod 2^w
6          for j = 0 to s - 1 do
7              t[i + j] = t[i + j] + m × n[j]
8      C = (t_{s-1}, ··· , t_1, t_0)                  // Lower s-word of t
9      return C
```

3.1.4 Multiplication in $GF(2^m)$ using Residue Number System (RNS)

RNS is a number representation, which divides a large integer into smaller size integers [31]. Suppose that we have pairwise relatively prime moduli set $\{m_1, m_2, \cdots, m_N\}$ with the least common multiple of m_i being M. Then any number $X < M$ has a unique representation in the defined residue number system as $\{x_1, x_2, \cdots, x_N\}$, which satisfies $x_i = X \mod m_i$.

RNS could be used in the computation of $C = A \cdot B \mod M$. Then the product can be obtained by calculating $c_i = a_i \cdot b_i \mod m_i$, which provides perfect parallelism by design.

A highly parallel multiplication method using RNS is given in algorithm 8. It is based on Montgomery method in RNS [32]. The algorithm is fully parallel except two base extensions computed in line 4 and line 7. Unfortunately, these base extensions are

quite time consuming.

Algorithm 8: Montgomery multiplication over trinomial residues

```
   Given  : Precomputed constant matrices of multiplications by p_i^{-1} (mod t_i),
            p_{n+i} (mod t_{n+i}), m_{n+i}^{-1} (mod t_{n+i})
   Input  : A : {a_1, ··· , a_{2n}}, B : {b_1, ··· , b_{2n}}, P : {p_1, ··· , p_{2n}}
   Output : R : {r_1, ··· , r_{2n}} where r_i = a_i b_i m^{-1} (mod p_i) and m is
            Montgomery factor
 1 Function Mult(A, B)
 2     {c_1, ··· , c_{2n}} = {a_1, ··· , a_{2n}} × {b_1, ··· , b_{2n}}
 3     {q_1, ··· , q_n} = {c_1, ··· , c_n} × {p_1^{-1}, ··· , p_n^{-1}}
 4     {q_{n+1}, ··· , q_{2n}} = BaseExt({q_1, ··· , q_n})
 5     {r_{n+1}, ··· , r_{2n}} = {c_{n+1}, ··· , c_{2n}} + {q_{n+1}, ··· , q_{2n}} × {p_1, ··· , p_{2n}}
 6     {r_{n+1}, ··· , r_{2n}} = {r_{n+1}, ··· , r_{2n}} × {m_{n+1}^{-1}, ··· , m_{2n}^{-1}}
 7     {r_1, ··· , r_n} = BaseExt({r_{n+1}, ··· , r_{2n}})
 8     return {r_1, ··· , r_{2n}}

   Input  : {q_1, ··· , q_n residue representation of Q (mod M)
   Input  : {q_{n+1}, ··· , q_{2n} residue representation of Q (mod M')
 9 Function BaseExt(Q)                  // Newton's interpolation[32]
10     tmp_1 = q_1
11     for i = 2 to n do                // Can be calculated in parallel
12         tmp_i = q_i
13         for j = 0 to i − 1 do
14             tmp_i = ((tmp_i + tmp_i) × t_j^{-1})  mod t_i
15     for i = 2 to n do                // Can be calculated in parallel
16         q_{n+i} = tmp_n  mod t_{n+i}
17         for j = 0 to i − 1 do
18             q_{n+i} = ((q_{n+i} × t_j + tmp_j)  mod t_{n+i}
19     return {q_{n+1}, ··· , q_{2n}}
```

3.2 FPGA Implementations of Various Galois Field Multipliers

3.2.1 Logic Level Designs

Although it requires more effort and it takes more time to implement a low level design, it usually results in high performance due to highly customizing and optimizing the circuit.

There are many research works on Galois Field multipliers that use bare FPGAs as their computation units. One example is [3]. In [3], the authors analyzed complexities of bit parallel Karatsuba-Ofman multiplier for both FPGA and ASIC. They have com-

pared the area-time product of their design with previous designs and achieved the lowest area utilization in terms of logic resources for ASIC and LUTs for FPGA. Normally, i step Karatsuba-Ofman algorith can operate on $m = 2^i n$ bits long operands. However, usual operand lengths are not powers of two but usually prime numbers as recommended by NIST. Therefore most designs pad zeroes to achieve a size such that it is a power of two. But authors in [3] selected an asymmetrical method for iterations to achieve non-power of two lengths as illustrated in Figure 3.3.

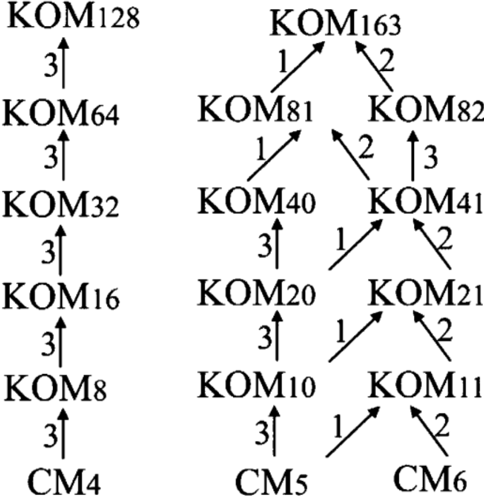


Figure 3.3: An example flow of regular (left) and proposed (right) iterations in [3]. Multiplier sizes are given as KOM_{SIZE} and number of multipliers used is given next to arrows.

3.2.2 Soft Processor Designs

Another approach is to make use of soft processors in FPGAs, which provides both easy programming of C language and flexibility of FPGA. In soft processor use in FPGA, the processors are created out of FPGA resources. Therefore such designs can be scalable that is more processors can be added and also whole system can be migrated to another FPGA brand or model as long as the FPGA resources are sufficient. In soft processor based systems, mostly the clock frequency becomes the bottleneck of the system, which has a big impact on the overall performance. Clock frequencies in soft processors are usually around 100-200 MHz.

Practical applications of Galois field multiplication usually requires lengths of mul-

tiplicands to be many times larger than word length of computation unit. First part of the multiplication, that is straightforward multiplication to obtain partial products, has quadratic complexity. In addition, reduction part at the end will make the computation even longer. For instance, a straightforward implementation of a 2048-bit long multiplier would require 4096 32-bit multiplications. Therefore, parallelization could lead to huge performance improvements.

In general, some key properties such as *balanced task partitioning, low intercommunication delay, high scalability* should be considered in order to maximize the efficiency of a parallel design [4]. Consequently, parallel designs differ from sequential designs in many aspects.

In [4] authors suggest a parallel Montgomery multiplier and compare row and column based partitioning in terms of task partitioning balance and communication overheads. Finally, they suggest a novel method, called as parallel Separated Hybrid Scanning (pSHS). They implement a prototype on Xilinx Virtex 5 FPGA using two, four and eight 32-bit MicroBlaze soft processor cores running at 100 MHz. Each MicroBlaze soft processor core has an independent local memory and are connected with each other via Fast Simplex Link (FSL). An example with four soft processor cores is illustrated in Figure 3.4 where timer connected to MicroBlaze0 measures the execution time.

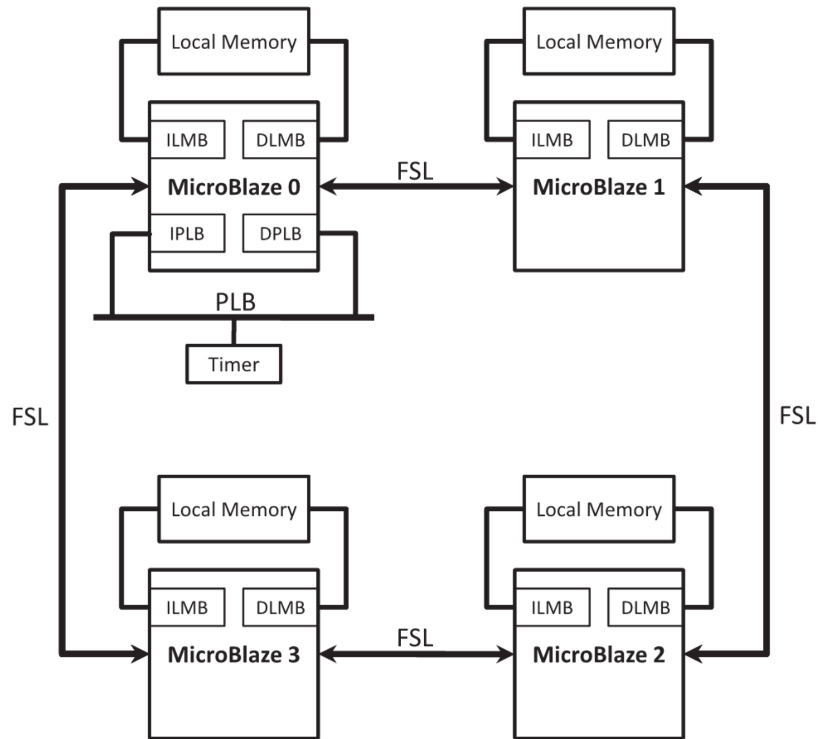


Figure 3.4: An example implementation of [4] with four soft processor cores.

3.3 GPU Implementations

High number of computation units in modern GPUs provide a great computation power and allow parallel operations to be performed very efficiently on GPUs. Therefore, major GPU manufacturers created lots of tools to make GPU kernel development process fast and easy. As a result, idea of using GPU as computation unit is widely accepted among authors especially for heavily parallel workloads. Even GPGPU (General purpose GPU) applications have emerged recently.

GPU implementations tend to differ from multi-core CPU implementations in the sense of parallelization. GPU implementations are supposed to be massively parallel due to GPU hardware design. One can simply think of a multicore CPU as a small number of large compute units whereas a GPU as a large number of small compute units.

In [28] authors proposed a method to implement Montgomery multiplication and optimized it for SIMD architecture of GPUs. They have implemented a design on nVidia

GTX-480 GPU.

GPUs can work not only in data-parallel way but also in task-parallel manner. Capability of performing different tasks at the same time makes GPUs suitable for RNS algorithm which basically performs similar but not the same operations on different numbers. In [30] authors proposed residue number system (RNS) based Montgomery multiplication. The advantage of RNS is that it is inherently parallel. They evaluated their design on nVidia 285 GTX GPU. In [29], it is shown that alternative Montgomery RNS designs are possible with same cost in terms of number of additions, multiplications and base extension/conversions. RNS algorithm is also safe for side channel attacks due to fully independent parallelism and arbitrary selection of numbers.

Although RNS algorithm introduces very good parallelism for multiplication, reduction process requires too much cross thread communication. In addition, the process needs preliminary and post conversion computations.

3.4 Other Multi-core Solutions

Cell Broadband Engine is used in [33] to perform multiplication in parallel. Cell Broadband Engine is the processor found in famous gaming console Sony Play Station 3, blade servers such as IBM QS20/21 and laptops such as Toshiba Qosmio. Cell Engine is a 64 bit variant of PowerPC running at 3.2GHz. It acts as central processor unit of a multi-processor system consisting of 8 Synergistic Processor Units (SPUs). Play Station 3 allows 6 of 8 SPUs to be used for general purpose computations and authors used all 6 in [33].

SPUs are specialized processors with SIMD capabilities and each have two pipelines therefore it can dispatch two instruction per cycle.

Authors of [33] used IBM multi-precision math (MPM) library and compared their results with Intel Core 2 Quad Q9300 processor. Their results are comparable but Sony Play Station 3 costs less than a desktop computer equipped with quad core Intel Core 2 processor.

However, the authors mention that these comparisons are not fair as AMD processors would yield similar performance at a lower cost. Also, Play Station 3 has 2 idle cores (6 of 8 cores utilized), which can perform other tasks while Intel processor is on full load. Lastly, prices may fluctuate in the market. Nevertheless, Cell Engine is demonstrated to be a promising computation unit for security applications.

3.5 Software Solutions

There are several software solutions for cryptographic applications that does not require additional hardware and run directly on main processor. The problem is that, most of the systems cannot share that much processor resource in a cryptographic applications.

In order to accelerate software solutions, Intel introduced a special instruction for their CPUs, *PCLMULQDQ*. It is based on Karatsuba-Ofman algorithm, [24]. The instruction divides multiplication with large sized operands such as 256 into 64 bit multiplications.

Authors of [34] use certain techniques, such as eliminating conditional branches (if statements), decreasing data dependencies and using pipeline stalls, to speed up finite field arithmetic operations on x86 and x64 based processors. They present test results for Intel Atom N450, Core 2 Duo E6750, Xeon E5440 and AMD Opteron 252.

CHAPTER 4

IMPLEMENTATION AND EVALUATION OF MONTGOMERY MULTIPLICATION ON FPGA USING OPENCL

4.1 Preliminary Calculations

As was presented in Chapter 3, Montgomery algorithm requires some constants for a given field. The field can be described by an odd modulo function $N(x) = \sum_0^{n-1} a_i 2^i + 1$, where $a_i = 1$ and $a_i = 0$ or 1 $\forall i \in [1..n]$. Montgomery multiplication algorithm requires constants R, R^{-1}, and N'.

Montgomery algorithm basically converts reduction by N into reduction by R. This constant is actually the tricky part of Montgomery algorithm. When R is chosen to be a power of 2, costly reduction and division parts are converted into just shift and erase operations. R should be chosen to be larger and relatively prime to N, that is $gcd(N, R) = 1$ and $R > N$.

R^{-1} is the multiplicative inverse of R and can easily be calculated by the Extended Euclidean algorithm given in the following subsection. R^{-1} is a constant for a given field.

For a given field, N' is another constant that satisfies $RR^{-1} = NN' + 1$, which can also be easily calculated by the Extended Euclidean algorithm.

4.2 Extended Euclidean Algorithm

Euclidean algorithm computes greatest common divisor of two integers [35]. Extended Euclidean algorithm additionally computes the coefficients of Bézout's identity (x, y) for given a and b that satisfies $ax + by = gcd(a, b)$ [35]. Pseudocode of Extended Euclidean algorithm is given in algorithm 9[35].

Algorithm 9: Extended Euclidean Algorithm

```
1  Function ExtEuclidean(a, b)
2      r_0 = a   r_1 = b
3      s_0 = 1   s_1 = 0
4      t_0 = 0   t_1 = 1
5      while r_{k+1} ≠ 0 do
6          q_i ← r_{i-1}/r_i                    // Integer division
7          r_{i+1} = r_{i-1} − q_i r_i
8          s_{i+1} = s_{i-1} − q_i s_i
9          t_{i+1} = t_{i-1} − q_i t_i
       // Euclidian Algorihm would return r_k = gcd(a,b)
10     return s_k, t_k                          // as_k + btk = gcd(a,b)
```

Extended Euclidean algorithm is very suitable to calculate constants for Montgomery, R^{-1} and N' as given in Equation 4.1

$$\begin{aligned} ax + by &= gcd(a, b) \\ RR^{-1} + NN' &= gcd(R, N) = 1 \end{aligned} \tag{4.1}$$

R, R^{-1}, N, and N' are all constants for a given field. Therefore, there is no need to calculate again and again.

4.3 Implementation

There are several Montgomery multiplication methods described in [27] and some examples are given in subsection 3.1.3. We have chosen SOS, Separated Operand Scanning, method because of its easy coding and easier/faster debugging along different CPU and GPU platforms. Moreover, it provides better readability of the code due to its simple data flow. On the other hand, integrated methods (CIOS and FIOS)

may actually improve the performance and decrease the memory usage. However, debugging process gets more complicated since the algorithm alternates between multiplication and reduction stages. SOS method first performs multiplication and then starts reduction process. Therefore, intermediate results can easily be compared using the results of other methods, such as Karatsuba or even simple school multiplication.

Any method, Karatsuba, RNS or simple school multiplication, can be used for the first part of SOS method, which is just multiplication. RNS method requires time consuming base transformations and conversion to/from RNS representation. Karatsuba method divides large numbers into smaller numbers and decreases multiplication width. It does not introduce any extra computation overhead but it is not balanced for parallelization due to its tree like structure. Therefore, it is hard to code Karatsuba into parallel kernels. Because of its perfect balance and easy coding, we chose simple school multiplication for the first part. Whole OpenCL code example is given in Appendix B. Note that this is just a way of implementing SOS algorithm, it is possible to further optimize the OpenCL code.

Data flow of algorithm 6, which is preferred in our implementation, is given in Figure 4.1 where A and B are two numbers in Montgomery domain.

Given, $R = 2^4$

$$\begin{aligned} &\dots \\ &t = AB \\ &c = (t \oplus (tN' \mod R)N)/R \\ &\dots \end{aligned}$$

$$\begin{array}{ccccccccc}
 & & & & & & A_2 & A_1 & A_0 \\
 & & & & & \times & B_2 & B_1 & B_0 \\
\hline
 & & & t_5 & t_4 & t_3 & t_2 & t_1 & t_0 \\
 & & & & & \times & N'_2 & N'_1 & N'_0 \\
\hline
\not{t'_8} & \not{t'_7} & \not{t'_6} & \not{t'_5} & \not{t'_4} & \not{t'_3} & t'_2 & t'_1 & t'_0 \\
 & & & & & \times & N_2 & N_1 & N_0 \\
\hline
 & & & t''_5 & t''_4 & t''_3 & t''_2 & t''_1 & t''_0 \\
 & & \oplus & t_5 & t_4 & t_3 & t_2 & t_1 & t_0 \\
\hline
 & & & c_5 & c_4 & c_3 & \not{c_2} & \not{c_1} & \not{c_0}
\end{array}$$

$t = AB$; $t' = tN' \mod R$; $t'' = t'N$; $u = (t' \oplus t'')/R$

Figure 4.1: Flow of Montgomery multiplication.

For this thesis work, multiplication lengths of 256, 512, 1024, 2048, 4096, 8192 are chosen and implemented for different scenarios. Following scenarios for both *unsigned char* and *unsigned int* data structures for all sizes are implemented:

- Multiplication block fully unrolled, size of SIMD processor is 4
- Multiplication block fully unrolled, number of computation units is 2
- Multiplication block fully unrolled, number of computation units is 2, size of SIMD processor is 2
- Half of multiplication block unrolled, number of computation units is 2, size of SIMD processor is 4
- Multiplication block fully unrolled, number of computation units is 2, size of SIMD processor is 2, 1-lvl Karatsuba algorithm used for primitive multiplication

Unfortunately, one scenario with multiplication size of 8192 with *unsigned int* basic data type which uses 1-level Karatsuba multiplication did not fit into FPGA and the compilation failed. Therefore corresponding results are missing.

4.3.1 Inputs and Outputs

Inputs A and B are divided into 8 bit or 32 bit words depending on the data structure used. 8 bit structure will be referred as *unsigned char* implementation and 32 bit structure will be referred as *unsigned int* implementation throughout the thesis. Inputs are converted into Montgomery domain by the host application and are then passed to the FPGA as kernel arguments. Pre-calculated constants N, N' are also passed to FPGA, separately. Constant N is an irreducible polynomial and is different for each multiplication size. Values of N are chosen from [36] as trinomials and are given in Table 4.1 and in section A.1.

Table 4.1: Chosen irreducible polynomials for different multiplication sizes

Multiplication Size	Irreducible Polynomial, N
256	$2^{255} + 2^{82} + 1$
512	$2^{511} + 2^{216} + 1$
1024	$2^{1014} + 2^{385} + 1$
2048	$2^{2044} + 2^{45} + 1$
4096	$2^{4074} + 2^{595} + 1$
8192	$2^{8145} + 2^{728} + 1$

In addition, empty buffers t and m, which are double the size of inputs, are provided to the FPGA. Buffer t holds the partial product, while buffer m holds m_i values calculated right after AB product is formed in algorithm 6. Finally, an output buffer c is created at equal size with inputs. The multiplier provides an out in Montgomery domain similar to inputs.

Constant R is hard coded as $2^{(k+1)*8}$ or $2^{(k+1)*32}$. So R is not passed to FPGA. Also, constant R^{-1} is not passed to the kernel because output is provided in Montgomery domain anyway. R^{-1} is used by the host application at the end to convert the result back from Montgomery domain.

Partitioning is similar to the one described in Figure 3.2b. A whole row is computed in parallel at once. For inputs A and B, all elements of A are multiplied with the first element of B and then the second, the third, etc. in a loop. Hence loop size is equal to word count in B which is related to multiplication size as in Equation 4.2.

$$\text{Word Count} = \begin{cases} \text{Multiplication Size}/8, & \text{for } \textit{unsigned char} \\ \text{Multiplication Size}/32, & \text{for } \textit{unsigned int} \end{cases} \tag{4.2}$$

Accumulations are performed at the end of each loop in global buffer t.

Similar steps are used for following multiplications in reduction part. Modulo and division operations are done by just neglecting and changing array indexes since modulus and division (by R) is now a power of two. This is the main advantage in Montgomery multiplication.

4.3.2 Kernel Attributes

Since the implementation platform is FPGA, our architecture is now very flexible. We can easily manipulated it using specific kernel attributes mentioned earlier. Actually, these attributes shape the structure of FPGA. Therefore, changing these attributes might have a huge impact on FPGA resource usage, performance and kernel clock frequency. So, multiple trade-offs might appear with no certain winner.

In this thesis, different values for the following abbreviated attributes are investigated:

- **RWS (Required workgroup size)** allows compiler to further optimize the design by giving an exact value. Defaults to 256 when omitted. Unless otherwise specified it is equal to the word size given in Equation 4.2.
- **CU (Number of computation units)**
- **SWI (SIMD work items)** determines the width of the SIMD processor.
- **MBU (Multiplication block Unrolled)** Basic multiplication block has a constant loop length equal to the width of basic unit, i.e., 8 or 32. If multiplication block is unrolled, performance increases dramatically at the expense of more FPGA resources. Unless otherwise specified, this loop is unrolled (at least at half size).

4.3.2.1 FPGA Resource Usages

Altera Offline Compiler (AOC) estimates area usage based on OpenCL code before the full compilation. Estimated area depends on code, number of computation units, size of SIMD processor and depth of loop unrolling. Comparison of estimated and implemented FPGA resource utiliziations are given in Table 4.2. Multiplication size does not change logic resource utilization, but small effect on memory utilization due to pipelined parallel structure of Altera OpenCL implementation.

Table 4.2: Comparison of area utilizations (in chip's resource percentage) of 1024 bit Montgomery SOS algorithm implementations for different kernel attributes

Attribute	Estimated (%)		Implemented (%)	
	Logic	Mem.	Logic	Mem.
None	42	70	48	36
RWS = 32	42	42	48	11
RWS = 32 *SWI = 8*	68	59	74	15
RWS = 32 *CU = 2, SWI = 4*	90	80	94	22
RWS = 32 *SWI = 4* *MBU*	92	44	69	31
RWS = 32 *CU = 2* *MBU*	58	57	64	22
RWS = 32 *CU = 2, SWI = 2* *MBU*	69	64	80	37
RWS = 32 *CU = 2, SWI = 4* *Half MBU*	82	44	91	21
RWS = 32 *CU = 2, SWI = 2* *1-lvl Karatsuba* *MBU*	70	68	87	42

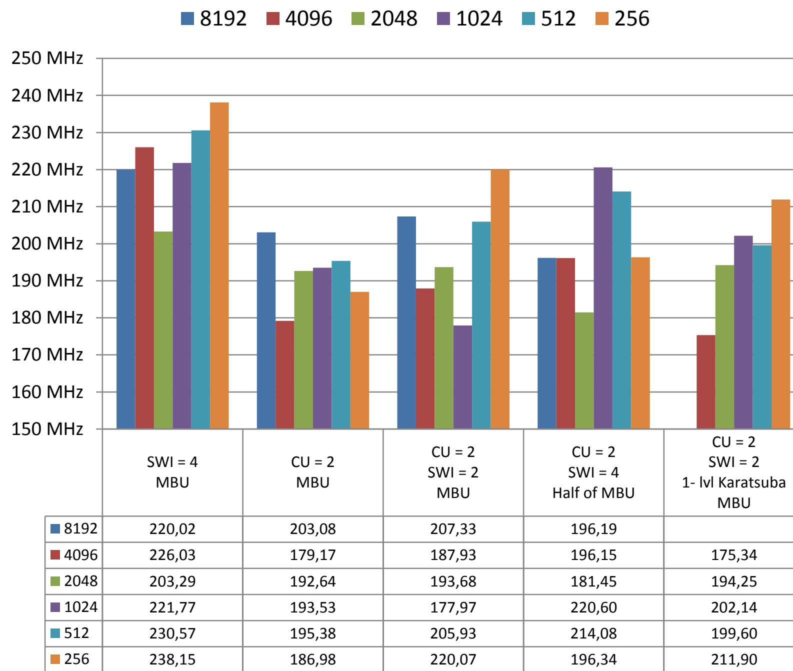
RWS: Required Workgroup Size, CU: Number of Compute Units, SWI: Number of SIMD Work Items, MBU: Multiplication block unrolled

We observe that compiler estimations do not exactly match actual implementation results. The compiler seems to underestimate logic utilization but overestimate memory utilization in general. However, estimations may still be a good clue to foresee a possible compilation failure later. Since the compiler attempts multiple times to fit a design before giving up, estimations could save several hours of compilation. If estimated logic or memory usage far exceeds 100%, it will probably not fit to chip anyway.

4.3.2.2 Kernel Frequencies

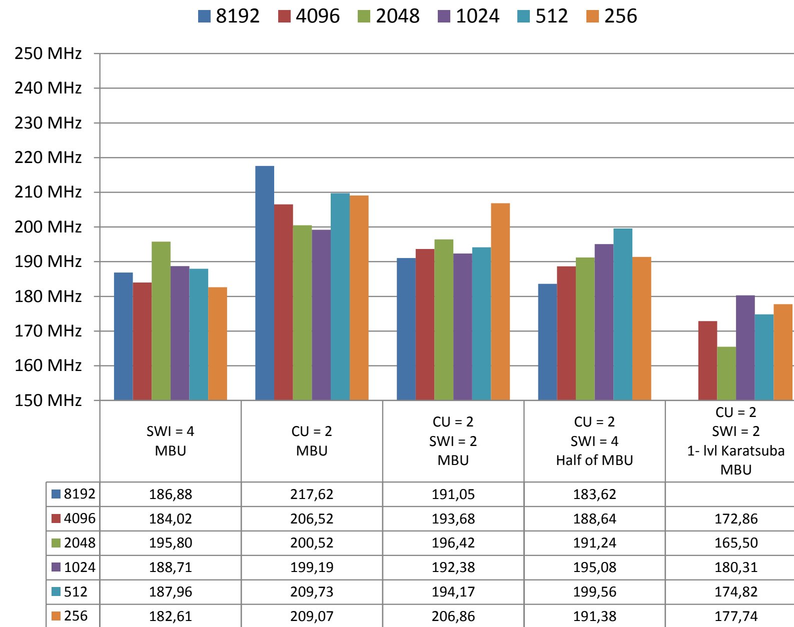
Throughput is supposed to be affected by the kernel frequency. However, its effect is negligible compared to the effect of architecture. This is because the frequency

values are found to be close to each other and around 175 MHz to 200 Mhz as given in Figure 4.2 for *unsigned char* implementation and Figure 4.3 for *unsigned int* implementation.



	SWI = 4 MBU	CU = 2 MBU	CU = 2 SWI = 2 MBU	CU = 2 SWI = 4 Half of MBU	CU = 2 SWI = 2 1- lvl Karatsuba MBU
8192	220,02	203,08	207,33	196,19	
4096	226,03	179,17	187,93	196,15	175,34
2048	203,29	192,64	193,68	181,45	194,25
1024	221,77	193,53	177,97	220,60	202,14
512	230,57	195,38	205,93	214,08	199,60
256	238,15	186,98	220,07	196,34	211,90

Figure 4.2: Comparison of implemented (*unsigned char*) kernel frequencies for different kernel attributes and multiplication sizes



	SWI = 4 MBU	CU = 2 MBU	CU = 2 SWI = 2 MBU	CU = 2 SWI = 4 Half of MBU	CU = 2 SWI = 2 1- lvl Karatsuba MBU
8192	186,88	217,62	191,05	183,62	
4096	184,02	206,52	193,68	188,64	172,86
2048	195,80	200,52	196,42	191,24	165,50
1024	188,71	199,19	192,38	195,08	180,31
512	187,96	209,73	194,17	199,56	174,82
256	182,61	209,07	206,86	191,38	177,74

Figure 4.3: Comparison of implemented (*unsigned int*) Kernel frequencies for different kernel attributes and multiplication sizes

The above figures illustrate that the effect of multiplication size is smaller compared to the effect of architecture. Moreover, there are big fluctuations due to random processes of compiler during the compilation.

Kernel frequencies tend to be lower for the scenario that contains 1-level Karatsuba multiplication. This is because of having a longer combinational path introduced by Karatsuba method when multiplication is divided into three half size multiplications and their results are XORed to obtain the final result.

Since *unsigned int* (32 bit) implementation has a larger primitive data size (meaning a wider data bus in FPGA) than *unsigned char* (8 bit) implementation, *unsigned int* implementations have lower frequencies in general.

4.3.2.3 Kernel Performances

The architecture dramatically affects performance. Initial scenarios and experiments without loop unrolling have performed very poorly and therefore we do not include such cases in the results presented thereafter. A comparison of performances for

different architectures is presented in Table 4.3 and visualized in Figure 4.4 and Figure 4.5 for *unsigned char* and *unsigned int*, respectively. Unfortunately, results of the last scenario with size of 8192 using *unsigned int* and 1-level Karatsuba method is missing since the compiler could not fit this design into hardware and failed compilation.

Table 4.3: Comparison performances (in multiplications per second) for different kernel attributes

Attribute	Unsigned char						Unsigned int					
	8192	4096	2048	1024	512	256	8192	4096	2048	1024	512	256
SWI = 4 MBU	54	150	334	969	2331	5155	318	840	2092	4082	8264	13699
CU = 2 MBU	20	59	191	515	1252	3268	217	558	1309	3413	8547	16667
CU = 2 SWI = 2 MBU	34	93	257	560	1748	4405	260	651	1751	4132	7937	16129
CU = 2 SWI = 4 Half MBU	48	131	291	954	2188	4425	193	524	1481	3268	6711	10989
CU = 2 SWI = 2 1-lvl Krsba MBU	32	99	267	631	1082	4132	*	588	1508	3817	7692	14706

SWI: Number of SIMD Work Items, CU: Number of Compute Units, MBU: Multiplication block unrolled
* Compiler failure

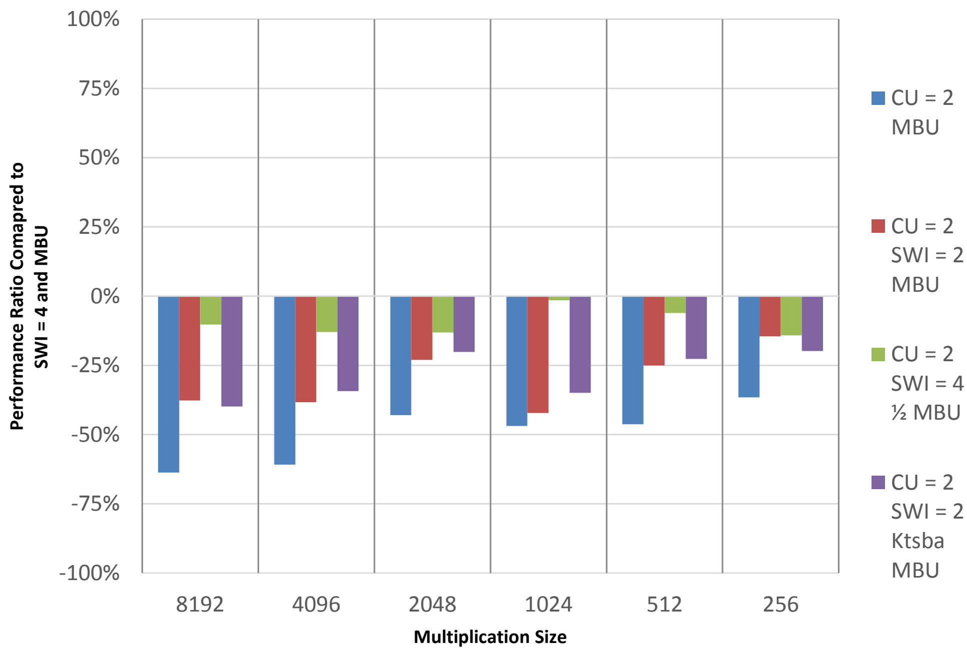


Figure 4.4: Comparison of normalized kernel performances for different kernel attributes (*unsigned char*)

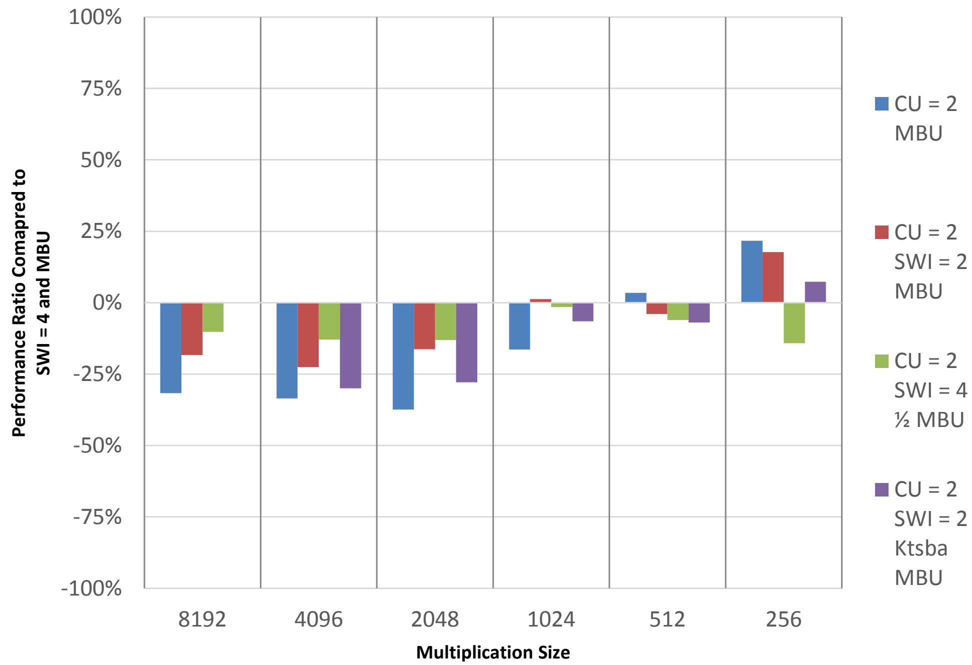


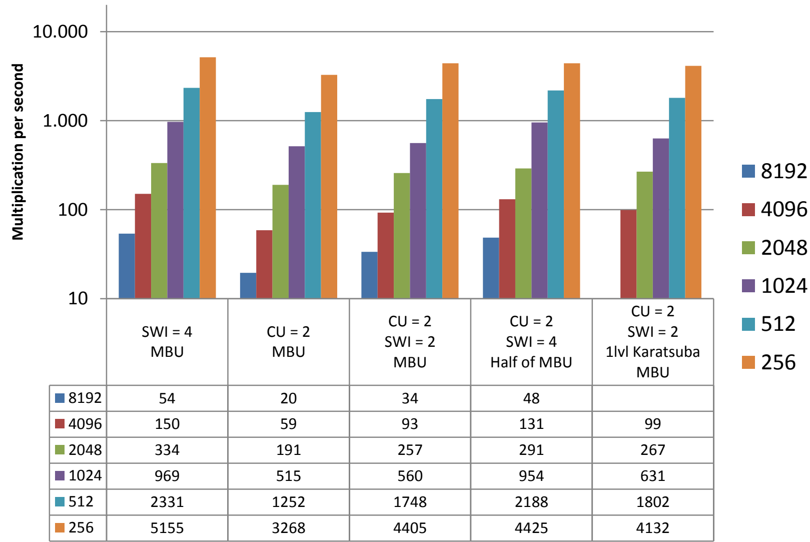
Figure 4.5: Comparison of normalized kernel performances for different kernel attributes (*unsigned int*)

Figure 4.6 and Figure 4.7 illustrates performance differences relative to first scenario where number of computations units is 1, SIMD processor size is 4 and multiplication block is fully unrolled. Zero level is the reference level, negative values indicate lower number of multiplication per second for given scenario. Charts show that there is up to 50 percent difference for 8 bit unsigned char and up to 25 percent difference for 32 bit unsigned int implementation. This actually implies there is a very good opportunity for designer to optimize design depending of the problem.

Implemented algorithm has two identical and constant nested loops size of word length in for each multiplication in the sequential school multiplication (line 2 and line 5 in algorithm 6). Outer loop is executed in parallel on each kernel. However, internal loop cannot easily be parallelized because of having data dependencies. Therefore, loop size (hence multiplication size) is inversely proportional to performance. Additionally, when multiplication size is doubled, size of the result is also doubled. Therefore, number of multiplications per second is observed to be inversely proportional to the square of multiplication size.

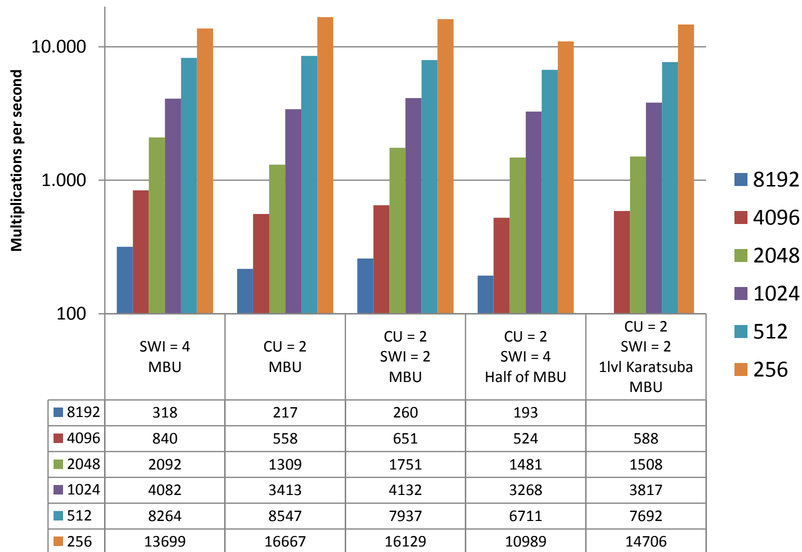
Effect of multiplication size is observed to be exponential as expected and as depicted

in Figure 4.6 and Figure 4.7 for *unsigned char* and *unsigned int*, respectively. We note that charts are in logarithmic scale.



	SWI = 4 MBU	CU = 2 MBU	CU = 2 SWI = 2 MBU	CU = 2 SWI = 4 Half of MBU	CU = 2 SWI = 2 1lvl Karatsuba MBU
8192	54	20	34	48	
4096	150	59	93	131	99
2048	334	191	257	291	267
1024	969	515	560	954	631
512	2331	1252	1748	2188	1802
256	5155	3268	4405	4425	4132

Figure 4.6: Comparison of Kernel performances for different multiplication sizes (*unsigned char*)



	SWI = 4 MBU	CU = 2 MBU	CU = 2 SWI = 2 MBU	CU = 2 SWI = 4 Half of MBU	CU = 2 SWI = 2 1lvl Karatsuba MBU
8192	318	217	260	193	
4096	840	558	651	524	588
2048	2092	1309	1751	1481	1508
1024	4082	3413	4132	3268	3817
512	8264	8547	7937	6711	7692
256	13699	16667	16129	10989	14706

Figure 4.7: Comparison of Kernel performances for different multiplication sizes (*unsigned int*)

4.3.3 Primitive Sizes

Our kernels are constructed using both 8 bit *unsigned char* and 32 bit *unsigned integer* primitive data stores. So that our Montgomery multiplier design can be scaled for $8 \times k$ bits (for *unsigned char*) or $32 \times k$ bits (for *unsigned int*), where k is an integer. *unsigned int* and *unsigned char* implementations differ in (i) their basic multiplication size and (ii) their work group/word counts and hence the size of multiplication loops. The algorithms and data flows are exactly the same in both. This provides more flexibility for 8 bit and 32 bit data processors and can further be extended to 64 bit with some modifications in the code. Size of multiplication can be changed with a simple modification in a *C header file* and 256, 512, 1024, 2048, 4096, 8192 or any other multiples of 8 or 32 can be used as long as it fits into the FPGA. However, the design should be re-compiled for each multiplication size since kernel attributes and constant loop counters depend on multiplication size. Work group size is equal to word count and depends on multiplication size as given in Equation 4.2 Table 4.4 lists work group size for different multiplication sizes and for *unsigned char* and textitunsigned int.

Table 4.4: Multiplication size vs. work group sizes

Multiplication Size	Unsigned char	Unsigned int
256	32	8
512	64	16
1024	128	32
2048	256	64
4096	512	128
8192	1024	256

FPGA resource usage does not directly depend on workgroup size and multiplication size due to pipelined structure of Altera OpenCL implementation. However, primitive size has impact on FPGA resource usage because register sizes and multiplier sizes (therefore size of computation unit) directly depend on the most basic storage unit. Increasing basic data width increases logic and memory usage. More detailed information on FPGA resource and memory usage is presented in Figure 4.8 and Figure 4.9i respectively. We note that the pattern within the same multiplication size is repeated for other sizes as well.

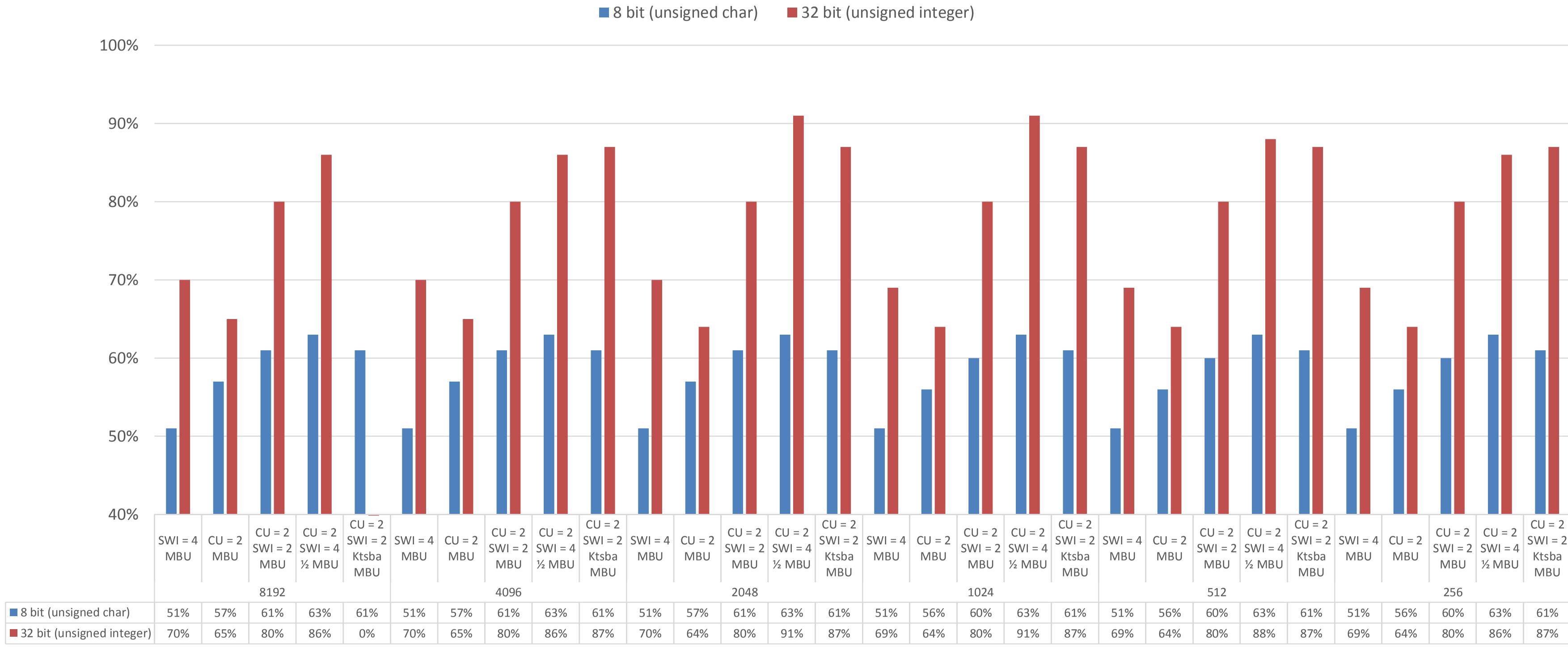


Figure 4.8: Logic resource utilization for the implemented multiplier on FPGA

■ 8 bit (unsigned char) ■ 32 bit (unsigned integer)

8192	SWI = 4 MBU	CU = 2 MBU	CU = 2 SWI = 2 MBU	CU = 2 SWI = 4 ½ MBU	CU = 2 SWI = 2 Ktsba MBU
8 bit (unsigned char)	14%	24%	24%	23%	23%
32 bit (unsigned integer)	46%	32%	48%	22%	0%

4096	SWI = 4 MBU	CU = 2 MBU	CU = 2 SWI = 2 MBU	CU = 2 SWI = 4 ½ MBU	CU = 2 SWI = 2 Ktsba MBU
8 bit (unsigned char)	12%	19%	21%	21%	20%
32 bit (unsigned integer)	31%	30%	37%	22%	42%

2048	SWI = 4 MBU	CU = 2 MBU	CU = 2 SWI = 2 MBU	CU = 2 SWI = 4 ½ MBU	CU = 2 SWI = 2 Ktsba MBU
8 bit (unsigned char)	11%	18%	19%	18%	18%
32 bit (unsigned integer)	31%	23%	37%	21%	42%

1024	SWI = 4 MBU	CU = 2 MBU	CU = 2 SWI = 2 MBU	CU = 2 SWI = 4 ½ MBU	CU = 2 SWI = 2 Ktsba MBU
8 bit (unsigned char)	11%	16%	17%	18%	17%
32 bit (unsigned integer)	31%	22%	37%	21%	42%

512	SWI = 4 MBU	CU = 2 MBU	CU = 2 SWI = 2 MBU	CU = 2 SWI = 4 ½ MBU	CU = 2 SWI = 2 Ktsba MBU
8 bit (unsigned char)	11%	15%	17%	18%	17%
32 bit (unsigned integer)	31%	22%	37%	21%	42%

256	SWI = 4 MBU	CU = 2 MBU	CU = 2 SWI = 2 MBU	CU = 2 SWI = 4 ½ MBU	CU = 2 SWI = 2 Ktsba MBU
8 bit (unsigned char)	11%	15%	17%	18%	17%
32 bit (unsigned integer)	31%	22%	37%	21%	42%

Figure 4.9: Memory utilization for the implemented multiplier on FPGA

4.3.4 Offline Compilation

As was mentioned earlier, OpenCL compiles kernels during run time. On the other hand, Altera OpenCL implementation uses pre-compiled kernels. Altera is currently providing an offline compiler (AOC) in OpenCL SDK for this task. Altera OpenCL SDK version 13.0 with service pack 1 is used in this thesis. In this version, AOC runs on command line. Therefore, a simple C# application is coded in order to speed up and make batch processing possible (see Figure 4.10). Batch processing is important for this thesis also due to quite long compilation times, which are given in Figure 4.11, and Figure 4.12 for *unsigned char* and *unsigned int*, respectively. Compilations are performed on a 64 bit Windows 7 workstation equipped with an 2.66 GHz dual 6 core Intel Xeon X5650 (24 threads with hyper-threading) with 48 GB of RAM. Compilations take even longer time when resource usage gets closer to 100%. Failed compilations took around 9-10 hours because the compiler tries to fit the design again and again before eventually giving it up.

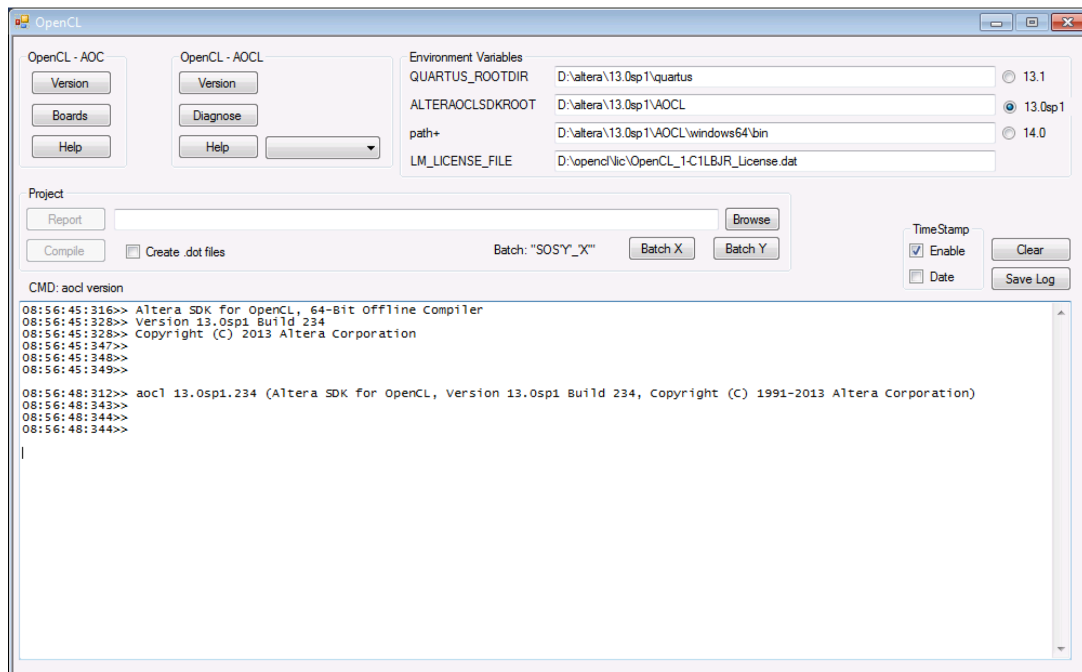


Figure 4.10: C# application for compilation

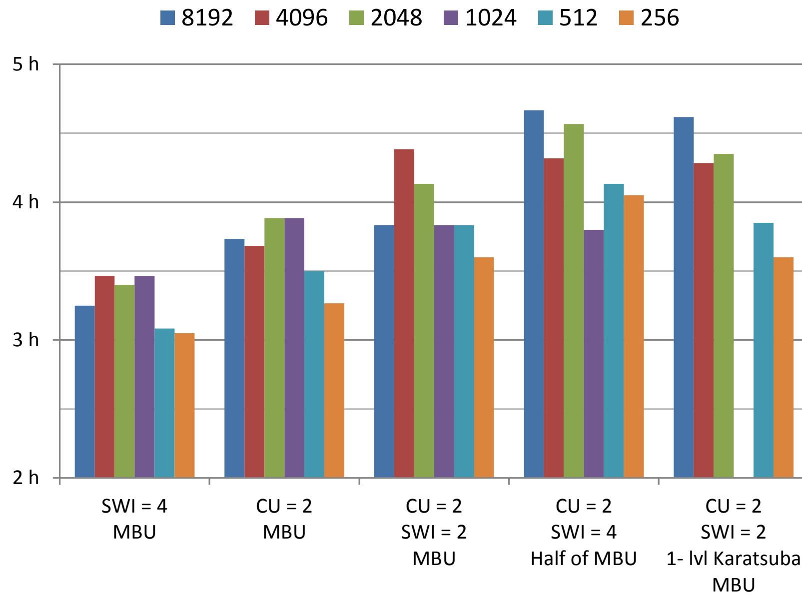


Figure 4.11: Compilation times for *unsigned char* implementations

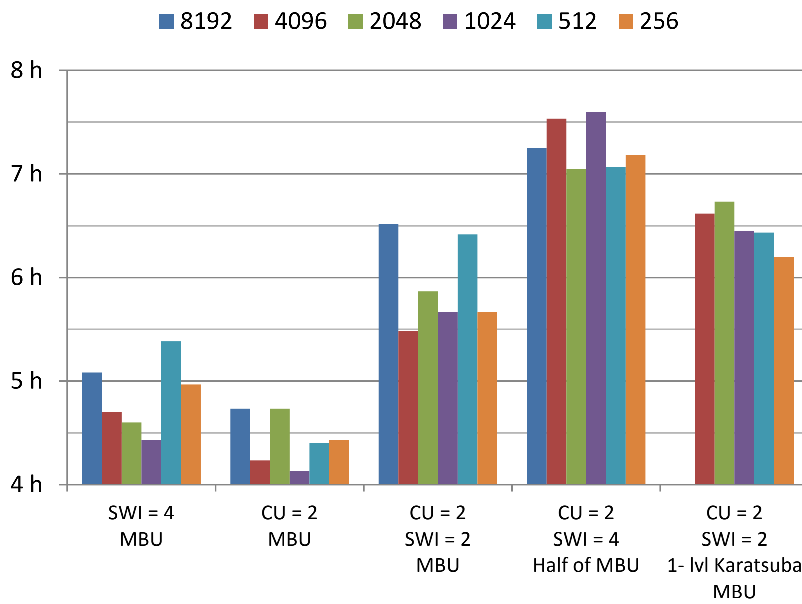


Figure 4.12: Compilation times for *unsigned int* implementations

4.4 Functional Testing

4.4.1 Reference Results

A reference application that performs Galois Field multiplication of large numbers is required to verify the results. The first choice would be to use Matlab, well known numerical computation environment. However, by default, Matlab supports field sizes of up to 2^{16} only if $gf(x, m)$ function is to be used[1]. Unfortunately this is far too small for our required range of computations (lowest required field is 2^{256} and largest is 2^{8192})

Although there are some online tools that are capable of doing arithmetic operations in Galois Field, they are either unreliable or too slow for large numbers. Moreover, they are not suitable to be used in an automated tool for fast verification.

Therefore, we implemented a simple application in C#, which uses *BigInteger* class[2], to do computations using large numbers. The application uses several methods (simple school multiplication, Karatsuba, Montgomery, RNS) for both regular integer and Galois Field arithmetic. Moreover, our application performs computations with the same data flow as both *unsigned int* and *unsigned char* OpenCL implementations. In this way, intermediate values of calculations can be debugged even for large numbers up to 8192 bits. However, this application is not optimized neither for performance nor for user experience. A screenshot from this multiplier application can be seen in Figure 4.13.

[1] For current release, R2014a: http://www.mathworks.com/help/comm/ref/gf.html
[2] Introduced in .NET Framework 4,
http://msdn.microsoft.com/en-us/library/system.numerics.biginteger(v=vs.100).aspx

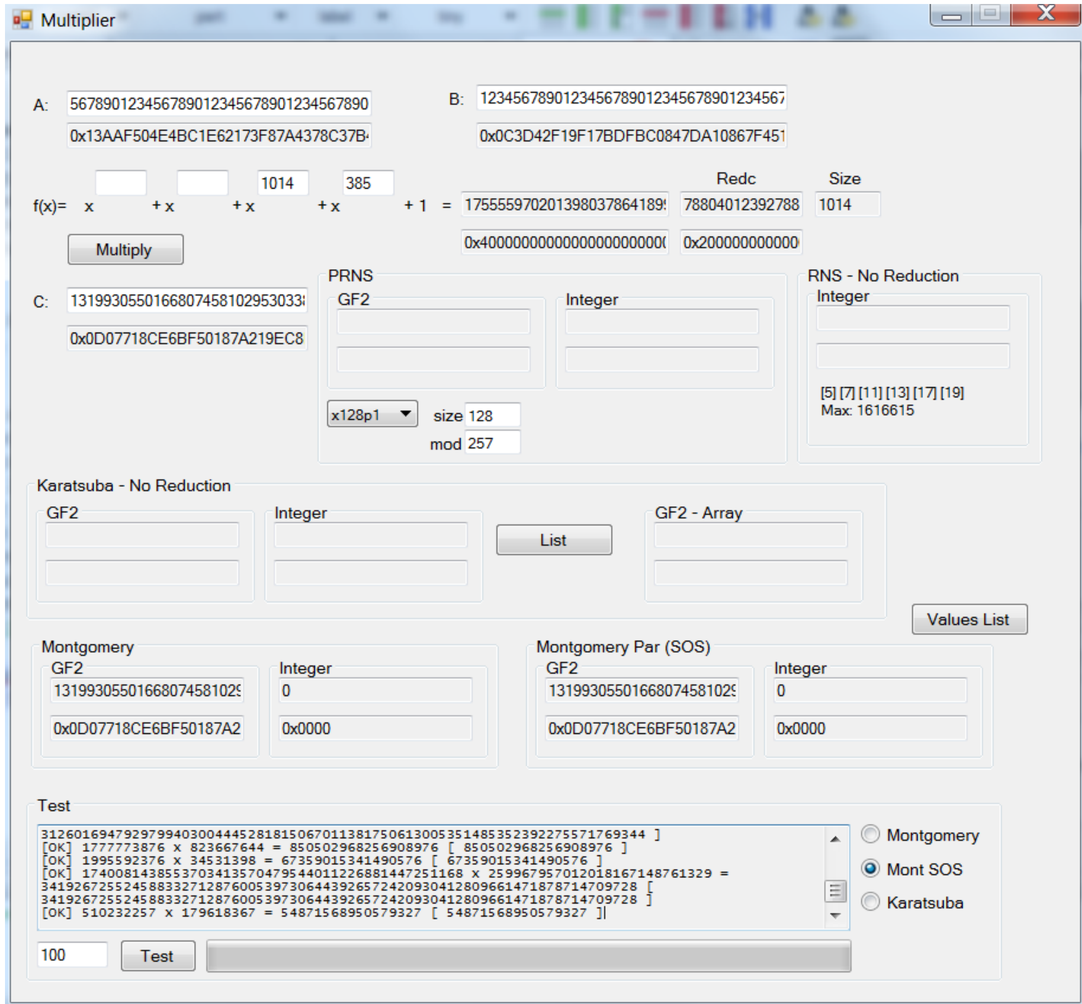


Figure 4.13: A screenshot of multiplier application written in C# for functional testing

OpenCL supports `printf()` function located in standard C libraries. It provides great benefits during coding on CPU and GPU. FPGA designs must also be recompiled even for a small change but compliation is too slow. Moreover, `printf()` consumes too much logic resources on FPGA and, needless to say, has very poor performance. Therefore, they are removed after debugging and not used while obtaining our test results on performance or resource usage.

4.4.2 Benchmarks and Profiling

Following our functional verification, all debug interfaces and auxiliary intermediate values are removed and final working codes are compiled. Performance results are

obtained from both host application and OpenCL device by running kernels for 1000 times and getting the minimum of these results.

OpenCL device can report profiling information by `clGetEventProfilingInfo`[3]. Profiling information contains times in nanoseconds when a command is *queued*, *submitted*, *started* and *ended* [37]. Subtracting start time of command from end time of command results in total execution time in nanoseconds.

Results are also collected using Windows API `QueryPerformanceCounter`[4]. This approach provides more detailed performance results such as memory transfers, enqueuing tasks, initializing OpenCL system, programming FPGA, etc.

Profiler and performance counter results are shown in Figure 4.14. Profiler results are a bit smaller than Windows API results as can be seen and verified in the image. This is because of having the overhead of profiler information collection from FPGA included in Windows API method. However, the difference is negligibly small.

[3] Details: http://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/clGetEventProfilingInfo.html

[4] Provides current value of time stamp from high resolution ($1\mu s$) performance counter, http://msdn.microsoft.com/en-us/library/windows/desktop/ms644904(v=vs.85).aspx

```
Administrator: C:\Windows\system32\cmd.exe
|                   Name                 |   Min   |
|----------------------------------------------------|
|                        |  Profiler  |    663 |
|                        |  Execution |    665 |
|            SOS2048_13  |     Memory |    421 |
|                        |    Enqueue |     37 |
|                        | KernelArgs |      0 |
------------------------------------------------------

D:\opencl\code\_ALL\MontgomeryUI\x64\Debug>hello_world_vs2010_nallatech.exe ui 8 4096 p

------------------------------------------------------
|                   Name                 |   Min   |
|----------------------------------------------------|
|                        |  Profiler  |   1190 |
|                        |  Execution |   1195 |
|             SOS4096_8  |     Memory |    507 |
|                        |    Enqueue |     38 |
|                        | KernelArgs |      0 |
------------------------------------------------------

D:\opencl\code\_ALL\MontgomeryUI\x64\Debug>hello_world_vs2010_nallatech.exe ui 9 4096 p

------------------------------------------------------
|                   Name                 |   Min   |
|----------------------------------------------------|
|                        |  Profiler  |   1791 |
|                        |  Execution |   1793 |
|             SOS4096_9  |     Memory |    405 |
|                        |    Enqueue |     32 |
|                        | KernelArgs |      0 |
------------------------------------------------------

D:\opencl\code\_ALL\MontgomeryUI\x64\Debug>hello_world_vs2010_nallatech.exe ui 10 4096 p

------------------------------------------------------
|                   Name                 |   Min   |
|----------------------------------------------------|
|                        |  Profiler  |   1536 |
|                        |  Execution |   1539 |
|            SOS4096_10  |     Memory |    496 |
|                        |    Enqueue |     37 |
|                        | KernelArgs |      0 |
------------------------------------------------------

D:\opencl\code\_ALL\MontgomeryUI\x64\Debug>hello_world_vs2010_nallatech.exe ui 11 4096 p

------------------------------------------------------
|                   Name                 |   Min   |
|----------------------------------------------------|
|                        |  Profiler  |   1533 |
|                        |  Execution |   1542 |
|            SOS4096_11  |     Memory |    506 |
|                        |    Enqueue |     37 |
|                        | KernelArgs |      0 |
------------------------------------------------------

D:\opencl\code\_ALL\MontgomeryUI\x64\Debug>hello_world_vs2010_nallatech.exe ui 12 4096 p

------------------------------------------------------
|                   Name                 |   Min   |
|----------------------------------------------------|
|                        |  Profiler  |   1910 |
|                        |  Execution |   1920 |
|            SOS4096_12  |     Memory |    515 |
|                        |    Enqueue |     38 |
|                        | KernelArgs |      0 |
------------------------------------------------------

D:\opencl\code\_ALL\MontgomeryUI\x64\Debug>hello_world_vs2010_nallatech.exe ui 13 4096 p

------------------------------------------------------
|                   Name                 |   Min   |
|----------------------------------------------------|
|                        |  Profiler  |   1700 |
|                        |  Execution |   1704 |
|            SOS4096_13  |     Memory |    499 |
|                        |    Enqueue |     37 |
|                        | KernelArgs |      0 |
------------------------------------------------------

D:\opencl\code\_ALL\MontgomeryUI\x64\Debug>hello_world_vs2010_nallatech.exe ui 8 8192 p
```

Figure 4.14: Benchmark results (in microseconds) obtained by host application for FPGA multiplier

Windows API also showed us that programming the FPGA via PCIe (CvP) takes around 3-4 seconds. For a typical application, this would only be required once per

total system shutdown if FPGA is not to be loaded from flash.

Similar benchmarking and profiling methods are used for CPU/GPU comparisons of the same code presented in the next chapter. However, memory transfers are not compared because it highly depends on system configuration, which varies a lot. A screenshot from a sample run for CPU/GPU testing is given in Figure 4.15

```
C:\Users\Nusret\Downloads\ufu\sonn\BeniCalistir.exe
    [1] Intel(R) OpenCL
Devices (1):
    [0] GeForce GTX 780 [Selected]
~~ Result OK
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~ 4096 bit on GPU (1000 iterations)
Platforms (2):
    [0] NVIDIA CUDA [Selected]
    [1] Intel(R) OpenCL
Devices (1):
    [0] GeForce GTX 780 [Selected]
~~ Result OK
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~ 8192 bit on GPU (1000 iterations)
Platforms (2):
    [0] NVIDIA CUDA [Selected]
    [1] Intel(R) OpenCL
Devices (1):
    [0] GeForce GTX 780 [Selected]
~~ Result OK
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~



----------- Unsigned int -----------
Execution Time Average (usec):
------------------------------------
|  Length  |    CPU     |    GPU     |
------------------------------------
| 256 bit  |    12.022  |    79.276  |
| 512 bit  |    28.269  |   146.333  |
| 1024 bit |    86.213  |   294.447  |
| 2048 bit |   318.339  |   524.498  |
| 4096 bit |  1234.124  |  1015.191  |
| 8192 bit |  4881.916  |  2229.018  |
------------------------------------


Execution Time Minimum (usec):
------------------------------------
|  Length  |    CPU     |    GPU     |
------------------------------------
| 256 bit  |     8.176  |    73.312  |
| 512 bit  |    23.068  |   140.320  |
| 1024 bit |    77.964  |   287.936  |
| 2048 bit |   302.512  |   497.408  |
| 4096 bit |  1212.676  |   988.608  |
| 8192 bit |  4813.620  |  2162.752  |
------------------------------------


----------- Unsigned char ----------
Execution Time Average (usec):
------------------------------------
|  Length  |    CPU     |    GPU     |
------------------------------------
| 256 bit  |     6.788  |   189.978  |
| 512 bit  |    27.685  |   387.662  |
| 1024 bit |   109.169  |   879.093  |
| 2048 bit |   395.646  |  1799.304  |
| 4096 bit |  1523.406  |  3348.312  |
| 8192 bit |  6197.041  |  7253.624  |
------------------------------------


Execution Time Minimum (usec):
------------------------------------
|  Length  |    CPU     |    GPU     |
------------------------------------
| 256 bit  |     4.964  |   183.136  |
| 512 bit  |    17.812  |   378.336  |
| 1024 bit |    95.776  |   867.808  |
| 2048 bit |   381.936  |  1783.264  |
| 4096 bit |  1488.324  |  3281.920  |
| 8192 bit |  6067.468  |  7175.232  |
------------------------------------
Press any key to continue . . .
```

Figure 4.15: Benchmark results (in microseconds) obtained by host application for CPU/GPU multiplier

CHAPTER 5

COMPARISON OF OPENCL FPGA MONTGOMERY MULTIPLIER WITH GPU AND CPU PLATFORMS

In this chapter, the Montgomery multiplier code written in OpenCL and evaluated on FPGA in the previous chapter is mapped and run on various GPU and CPU platforms. Results are compared and comments are made about the pros and cons of each platform. The OpenCL FPGA Montgomery Multiplier built in this thesis is also compared with implementations reported in the literature.

5.1 Comparison with GPU

One of the most important advantages of OpenCL over nVidia CUDA (a popular proprietary GPU programming language) is that OpenCL is portable. The same code can run on various platforms without any modification. So GPUs can be used as computation platform as well as FPGAs to offload CPU. Parallel computing capabilities and recent software improvements such as integrated development environments (IDE) made by major GPU manufacturers turned GPUs into very attractive platforms for general purpose computations also.

Modern graphics cards have very high speed PCIe interfaces to the host PC. They also have large amounts of on board DDR memory to provide high computing power. Sometimes, they even require additional power connections due to high power consumption.

We collected benchmark results by running the fastest OpenCL code for FPGA on

various GPU platforms. Table 5.1 lists the GPU platforms used in our tests and their specifications.

Table 5.1: Specifications of GPUs tested

GPU	Cores	Core Freq (MHz)	Memory BW (GB/sec)	Max Power (Watts)
nVidia Quadro FX 380	16*	450	22.4	34
nVidia Quadro 410	192*	706	14	38
nVidia GeForce GTX 780	2304*	863	288.4	250
nVidia GeForce GT 630	192*	875	28.5	50
nVidia GeForce GT435M	96*	1300	25.6	50

* CUDA Cores
Sources:
http://www.nvidia.com/object/product_quadro_fx_380_us.html
http://www.nvidia.com/object/quadro-410-graphics-card.html#pdpContent=2
http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-780/specifications
http://www.geforce.com/hardware/desktop-gpus/geforce-gt-630-oem/specifications
http://www.geforce.com/hardware/notebook-gpus/geforce-gt-435m

The ratio of GPU performance results (in number of multiplications per second) to FPGA performance results are presented in Figure 5.1, where workgoup size is also reported for each column marked as a thick yellow line. GPU to FPGA ratio is observed to be higher (GPU is better) when workgroup size is large. In other words, GPU performs better when workload is highly parallelized.

Unfortunately, nVidia Quadro FX380 failed to compute the multiplication result when size 8192 is implemented with *unsigned char* primitive data. This is probably because of having the the required workgroup size, which is 1024, being far more than the number of CUDA cores available, which is just 16 as can be seen in Table 5.1.

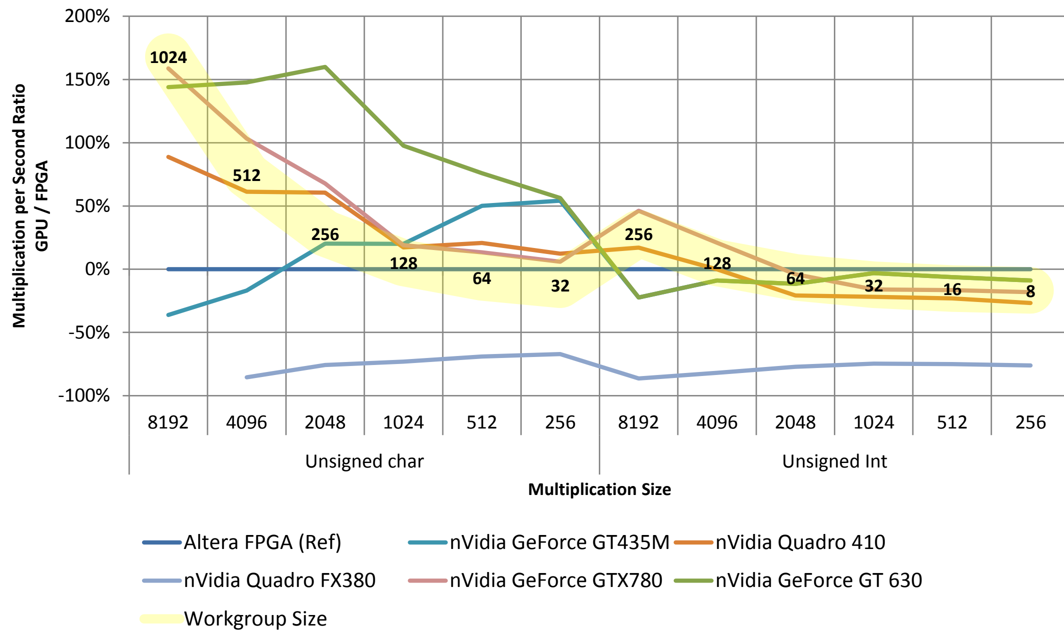


Figure 5.1: Performance comparison of FPGA implementation with GPUs

Actually, results for GPU could further be enhanced by optimizing the code for each platform depending on the number of cores and available SIMD capabilities. However, that would require too much effort and is out of the scope of this thesis. Actually, this is another advantage of using OpenCL on FPGA, which implements custom hardware so that hardware is always optimized for the developed code.

Having GPU as the compute unit has some disadvantages over FPGA. The architecture of an FPGA is fully customizable, meaning that different tasks can be implemented differently. On the other hand, the architecture of GPU is fixed. Even worse, structure of hardware differs from brand to brand, or even between models of the same brand. Therefore, it is hard to optimize a single code for all GPUs. On the other hand, all FPGA implementations can be optimized for a given specific code.

Another disadvantage of GPUs is the supply of a single chip. Major manufacturers do not sale small amounts of chips to small companies. Therefore, it is almost impossible to create custom boards with GPUs. Although there are integrated CPU/GPU (such as Intel i7) solutions, such GPUs usually have poor performance compared to other standalone GPUs. Moreover, CPU/GPU being located on the same (or very close) silicon area leads to single hot point in the design that would cause cooling problems.

Therefore, it is not a good co-processor solution for custom-made systems.

Finally, GPUs cannot be customized for a special task unlike FPGAs. FPGA can perform fully custom I/O operations unlike GPU. Those operations could be simple I/O, peripheral communication such as sensors, simple storage, other I2C/SPI/UART slave devices or even complex I/O operations such as networking, PCIe, SATA, etc. For example, an FPGA can be used as a SATA controller located on the host computers PCIe slot and perform encryption/decryption on the fly.

Moreover, modern high-end GPUs usually consume quite high power compared to FPGAs. Some high end graphics cards even require external power supply. Maximum power consumption for the GPUs tested in this thesis work are also given in Table 5.1.

5.2 Comparison with CPU

Naturally, CPUs are perfect compute units as the name central processing unit implies. The CPU is usually responsible for performing the main task and side-jobs should be taken care of by co-processors. Let us imagine a system performing a real-time image processing task while storing encrypted data to SATA device or sending/receiving data over a custom network with CRC-checksum, etc. In such a system, main task would be image processing, which could be performed by the CPU while the other task of encrypting and storing data on SATA device could be performed by FPGA. Actually, image processing could also be performed on FPGA but this is a topic, which is out of the scope of this thesis. Therefore, comparing a processor (CPU) with a co-processor (FPGA) is not so fair.

Another aspect is that modern CPUs are usually more power hungry due to having multiple cores, high clock frequencies, etc. The specifications of the CPUs compared in this thesis are given in Table 5.2.

Table 5.2: Specifications of CPUs tested

CPU	Cores	Core Freq (MHz)	Memory BW (GB/sec)	Max Power (Watts)
Intel i7 4770K	8*	3900	25.6	84
Intel Xeon X5650	12*	2666	32	95
Intel Xeon E5-2650	16*	2000	51.2	95
Intel i7 2620M	4*	2700	21.3	35

* including virtual hyper-threading cores
Sources:
http://ark.intel.com/products/75123/Intel-Core-i7-4770K-Processor-8M-Cache-up-to-3_90-GHz
http://ark.intel.com/products/47922/Intel-Xeon-Processor-X5650-12M-Cache-2_66-GHz-6_40-GTs-Intel-QPI
http://ark.intel.com/products/64590/
http://ark.intel.com/products/52231/Intel-Core-i7-2620M-Processor-4M-Cache-up-to-3_40-GHz

Due to the portability of OpenCL code, it can also run on a variety of CPUs without any modification. So we performed benchmarks of our Montgomery multiplier code on the CPUs listed in Table 5.2. Results are presented in Figure 5.2. Actually, these results could further be enhanced also by optimizing the code for each CPU platform depending on the number of cores and available SIMD capabilities similar to the argument made GPU comparison section. However, that would also require too much effort and is out of the scope of this thesis. As was mentioned before, this is actually an advantage of OpenCL on FPGA over CPU and GPU counterparts. Because OpenCL implements a custom hardware on FPGA. In other words, FPGA hardware is always optimized for the code and there is no need to optimize the code for hardware.

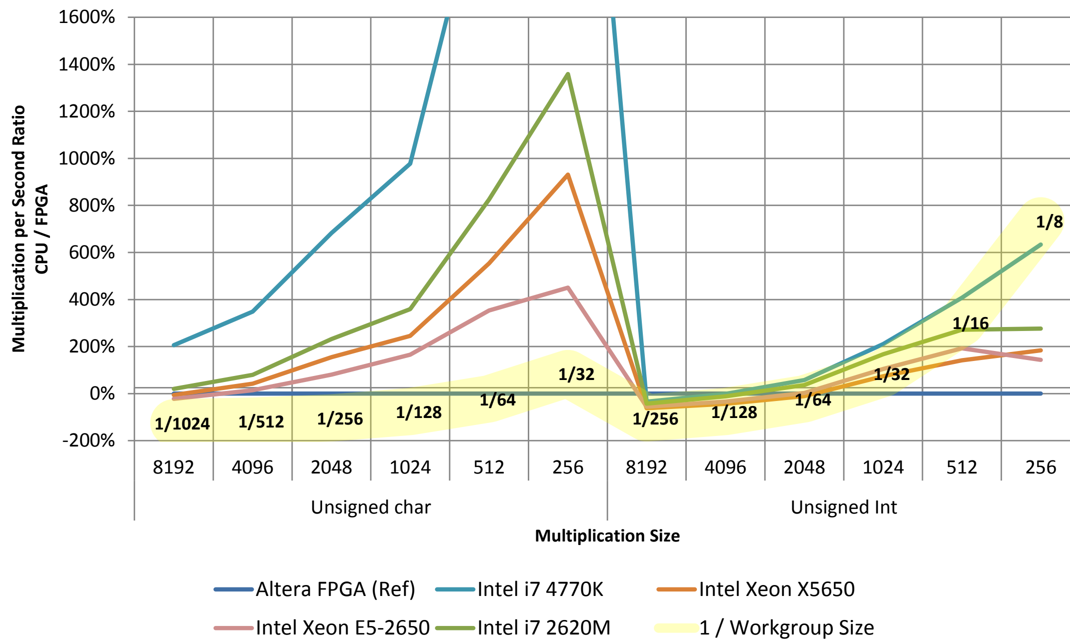


Figure 5.2: Performance comparison of FPGA implementation with CPUs

$1/WorkgroupSize$s are reported for each column and marked as a thick yellow line in Figure 5.2. It is observed that the smaller the number of workgroups is the better the performance is for CPUs. This is actually expected due to small number of cores.

5.3 Comparison with other Implementations in the Literature

In [33], the authors used Cell processors and 6 Synergistic Processor Units (Sony Play Station 3 allows 6 of 8 SPUs to be used for general purpose computations) to perform 256 bit multiplication. The authors compared their results with a software solution by using IBM multi-precision math (MPM) library running on Intel Core 2 Quad Q9300 processor. Results are given in Table 5.3

Soft processor designs such as the one in [4] have a similar approach along with this thesis. The design presented in [4] is implemented using multi MicroBlaze soft processor cores to perform parallel computing. Their results for 1024 bits operand size shows that computation takes 19334 cycles at 250 TTU (transfer time unit - transfer latency). That implementation is on Xilinx Virtex 5 while the clock speed

is 100MHz, which means 193.34 μsec per multiplication hence 5498 multiplications per second. Results of [4] are included in Table 5.3

Fastest solutions are from [38]. The authors in [38] use multiple Application Specific Instruction Processors (ASIP) together with multiple dedicated multiplier units (MMUs) running at 250 MHz. Their results are also included in Table 5.3.

Table 5.3: Comparison of performance of multiplier implementations in the literature.

Reference	Computation Unit	Operand Size	Operations / sec
This thesis	OpenCL on Altera FPGA	256	16667
This thesis	OpenCL on Altera FPGA	1024	4132
[33]	Play Station 3[1]	256	27474
[34]	2.6 GHz AMD Opreton 252	256	19048
[30]	nVidia GeForce 8800 GTS	224	3138
[30]	nVidia GeForce 285 GTX	224	9827
[4]	4x MicroBlaze[2]	1024	5498
[4]	8x MicroBlaze[2]	1024	7435
[38]	16x ASIP 16x 32bit MMU[3]	1024	65359
[38]	8x ASIP 8x 64bit MMU[3]	1024	70423
[38]	4x ASIP 4x 128bit MMU[3]	1024	72464
[38]	4x ASIP 16x 32bit Multiplier[4]	1024	16155

[1] 3.2 GHz Cell and 6x SPUs
[2] Softprocessor in Xilinx FPGA running at 100MHz
[3] ASIP: Application Specific Instruction Processor running at 250 MHz, MMU: Dedicated Montgomery multiplication Unit
[4] Utilizes pSHS method described in [4]

Implemented design is not the best nor the worst among implementations in the literature. However, due to flexible structure of FPGA and easy to use interface, it is very promising for many applications. Design is highly customizable and PCIe interface to host does not require much extra effort to implement high level applications.

CHAPTER 6

CONCLUSION

Electronic devices are very important in today's modern life. Especially mobile devices are already indispensable for everyone. Battery consumption, security, responsiveness, reliability, etc. are all very important topics for handheld devices. Not only handheld devices but other stationary electronic equipment are also required to be efficient in terms of power, security, resistance to errors, speed, responsiveness, etc.

Speed, security and reliability are fatal issues both in military and civil applications. The most common error correction techniques and many cryptographic algorithms have mathematical bases on Galois field. Hence arithmetic operations in Galois field should be performed very efficiently.

One of the most time consuming operations in Galois field is the modular exponentiation, which is basically repeated modular multiplication. Montgomery multiplication is a clever and fast technique that is very suitable for repeated multiplications and hence for the computation of modular exponentiation.

There is a general trend of performing parallel processing in modern computing devices. Major electronics manufacturers are now producing devices with many-core processing units. Even mobile phones now have multi-core processors. Therefore parallel programming stays to be an important topic.

Gate-level design of an FPGA project provides a highly parallel solution. However, such a design may have been very complicated and may require additional implementations such as a custom interface to host device.

In order to eliminate design complexity, a designer could chose software solutions to run on CPU or GPU. Unfortunately, using CPU for such task may not be desired due to the high workload required. Moreover, using GPU may limit optimization possibilities since GPU hardware cannot be modified.

On the other hand, OpenCL offers a parallel, heterogeneous and portable programming framework for software developers. Its parallel structure is in favor of many-core trend. Being a heterogeneous and portable framework makes OpenCL a very attractive environment for developers. Because a single piece of code can run on a variety of hardware even on customized hardware such as FPGA.

A designer can benefit from many aspects when OpenCL is used together with FPGA programming. Parallel, portable and fully customized designs may be created easily. Multiple tasks can be performed on a single custom hardware. Better yet, the same code would run on a different platform when custom hardware does not exist.

In the present thesis work, an OpenCL implementation of Montgomery multiplication is done and evaluated on FPGA. To summarize our findings, performance figures are in favor of GPU when workgroup size is high, CPU when workgroup size is small whereas FPGA is in the middle. Generated hardware on FPGA can easily be optimized to fit the specific problem requirements.

As a consequence, recently supported OpenCL on FPGA seems to be a very promising platform for systems developers. Especially those projects that require FPGAs for specialized tasks would benefit greatly by utilizing OpenCL on FPGA.

OpenCL code can be further optimized for performance. Since memory bandwidth is the most common bottleneck of parallel architectures, optimization possibilities can be investigated for memory transfers. Additionally, compilation process might be improved by manually performing intermediate steps of Altera offline compiler. AOC first generates VHDL code of the design using OpenCL code, then compiles it to generate programming file. AOC also creates some additional interfaces (i.e. temperature monitor) which are not mandatory for Montgomery multiplication. User can further decrease logic utilization by removing potentially unnecessary components. However, this leads manual compilation and requires too much effort which contradicts

with ease of programming advantage of OpenCL on FPGA.

REFERENCES

[1] A. Munshi, “The opencl specification,” November 2012. https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf accessed on 20 November 2013.

[2] Acceleware Corp., *OpenCL on FPGAs for GPU Programmers*, June 2014. http://design.altera.com/openclforward?elq=fb69b4f2a9dd4b77b9ed1253df464bc5 accessed on 12 July 2014.

[3] G. Zhou, H. Michalik, and L. Hinsenkamp, “Complexity analysis and efficient implementations of bit parallel finite field multipliers based on karatsuba-ofman algorithm on fpgas,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 18, no. 7, pp. 1057–1066, 2010.

[4] Z. Chen and P. Schaumont, “A parallel implementation of montgomery multiplication on multicore systems: Algorithm, analysis, and prototype,” *Computers, IEEE Transactions on*, vol. 60, no. 12, pp. 1692–1703, 2011.

[5] V. Guruswami, “List decoding of error-correcting codes,” 2001. http://hdl.handle.net/1721.1/8700 accessed on 15 December 2013.

[6] nVidia Corp., *CUDA C Programming Guide*, July 2013. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf accessed on 15 December 2013.

[7] Altera Corp., “Transceiver protocols,” 2014. http://www.altera.com/technology/high_speed/protocols/all-protocols/hs-all-protocols.html accessed on 12 July 2014.

[8] Xilinx Corp., “Vivado design suite user guide,” May 2014. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ accessed on 12 July 2014.

[9] H. Patil and S. Szygenda, *Security for Wireless Sensor Networks using Identity-Based Cryptography*. Taylor & Francis, 2012.

[10] E. W. Weisstein, “Finite field..” http://mathworld.wolfram.com/FiniteField.html accessed on 20 December 2013.

[11] I. Karonen, “Galois fields in cryptography,” May 2012.

[12] NIST, *Recommended Elliptic Curves for Federal Government Use*, July 1999. http://csrc.nist.gov/groups/ST/toolkit/documents/dss/NISTReCur.pdf accessed on 11 November 2013.

[13] Altera Corp., *Stratix V Device Overview*, May 2013. http://www.altera.com/literature/hb/stratix-v/stx5_51001.pdf accessed on 20 November 2013.

[14] Xilinx Inc., *7 Series FPGAs Overview*, July 2013. http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf accessed on 20 November 2013.

[15] Altera Corp., *Configuration via Protocol (CvP) Implementation in Altera FPGAs User Guide*, November 2013. http://www.altera.com/literature/ug/ug_cvp.pdf accessed on 20 November 2013.

[16] Xilinx Inc., *Zynq-7000 All Programmable SoC Overview*, September 2013. http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf accessed on 20 November 2013.

[17] Altera Corp., *Altera's User-Customizable ARM-Based SoC*, 2013. http://www.altera.com/literature/br/br-soc-fpga.pdf accessed on 20 November 2013.

[18] D. Chen and D. Singh, "Invited paper: Using opencl to evaluate the efficiency of cpus, gpus and fpgas for information filtering," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pp. 5–12, 2012.

[19] O. Rosenberg, *OpenCL Overview*, November 2011. http://www.khronos.org/assets/uploads/developers/library/overview/opencl-overview.pdf accessed on 19 November 2013.

[20] *AMD Accelerated Parallel Processing OpenCL*, July 2012. http://developer.amd.com/wordpress/media/2012/10/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide4.pdf accessed on 20 November 2013.

[21] Altera Corp., *Altera SDK for OpenCL Programming Guide*, November 2013. http://www.altera.com/literature/hb/opencl-sdk/aocl_programming_guide.pdf accessed on 20 November 2013.

[22] Altera Corp., *Altera SDK for OpenCL Optimization Guide*, November 2013. http://www.altera.com/literature/hb/opencl-sdk/aocl_optimization_guide.pdf accessed on 28 November 2013.

[23] A. Karatsuba and Y. Ofman, "Multiplication of many-digital numbers by automatic computers," *Proceedings of the USSR Academy of Sciences*, vol. 145, pp. 292–293, 1962. Translation in Physics-Doklady, 7 (1963), pp. 595–596.

[24] K. Jankowski, P. Laurent, and A. O'Mahony, "Intel polynomial multiplication instruction and its usage for elliptic curve cryptography," white paper, Intel Corp., April 2012.

[25] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, pp. 519–521, 1985.

[26] J. Henry S. Warren, *Montgomery Multiplication*, July 2012. `http://www.hackersdelight.org/MontgomeryMultiplication.pdf` accessed on 4 September 2014.

[27] C. Koç, T. Acar, and J. Kaliski, B.S., "Analyzing and comparing montgomery multiplication algorithms," *Micro, IEEE*, vol. 16, no. 3, pp. 26–33, 1996.

[28] K. Leboeuf, R. Muscedere, and M. Ahmadi, "A gpu implementation of the montgomery multiplication algorithm for elliptic curve cryptography," in *Circuits and Systems (ISCAS), 2013 IEEE International Symposium on*, pp. 2593–2596, 2013.

[29] B. Phillips, "Modular multiplication in the montgomery residue number system," in *Signals, Systems and Computers, 2001. Conference Record of the Thirty-Fifth Asilomar Conference on*, vol. 2, pp. 1637–1640 vol.2, 2001.

[30] S. Antão, J.-C. Bajard, and L. Sousa, "Rns-based elliptic curve point multiplication for massive parallel architectures," *The Computer Journal*, vol. 55, no. 5, pp. 629–647, 2012.

[31] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs, Second Edition*. Oxford University Press, Incorporated, 2010.

[32] J. Bajard, L. Imbert, and G. Jullien, "Parallel montgomery multiplication in gf(2k) using trinomial residue arithmetic," in *Computer Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symposium on*, pp. 164–171, 2005.

[33] N. Costigan and P. Schwabe, "Fast elliptic-curve cryptography on the cell broadband engine," 2009. `https://eprint.iacr.org/2009/016.pdf`.

[34] P. Longa and C. Gebotys, "Analysis of efficient techniques for fast elliptic curve cryptography on x86-64 based processors," 2010. `http://eprint.iacr.org/2010/335.pdf`.

[35] E. Bach and J. Shallit, *Algorithmic Number Theory, Volume 1 Efficient Algorithms*. MIT Press, 1996.

[36] G. Seroussi, “Table of low-weight binary irreducible polynomials.” http://www.hpl.hp.com/techreports/98/HPL-98-135.pdf accessed on 20 November 2013.

[37] K. Group, “Opencl 1.0 reference pages.” http://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/ accessed on 10 May 2014.

[38] J. Han, S. Wang, W. Huang, Z. Yu, and X. Zeng, “Parallelization of radix-2 montgomery multiplication on multicore platform,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 21, pp. 2325–2330, Dec 2013.

APPENDIX A

CONSTANT IRREDUCIBLE POLYNOMIALS (N) USED IN REDUCTION

A.1 Multiplication size = 256

$N = 2^{255} + 2^{82} + 1$

57 896 044 618 658 097 711 785 492 504 343 953 926 634 992 332 820 286 855 432 070 462 473 263 644 673

A.2 Multiplication size = 512

$N = 2^{511} + 2^{216} + 1$

6 703 903 964 971 298 549 787 012 499 102 923 063 739 682 910 296 196 688 861 780 721 860 882 015 036 773 488 400 937 254 395 743 382 402 202 627 011 270 709 097 309 260 301 068 685 522 328 078 814 019 585

A.3 Multiplication size = 1024

$N = 2^{1014} + 2^{385} + 1$

175 555 970 201 398 037 864 189 960 037 990 696 642 380 564 349 834 626 243 584 063 630 598 316 216 309 534 309 285 622 385 163 609 395 625 111 210 811 907 575 838 661 883 607 828 732 903 171 318 983 861 449 587 663 958 422 720 200 465 138 886 329 341 967 592 540 794 109 353 938 004 211 206 812 952 671

567 167 909 202 905 862 495 379 142 974 533 959 301 296 171 310 894 306 367 785 222 390 921 629 270 017

A.4 Multiplication size = 2048

$N = 2^{2044} + 2^{45} + 1$

2 019 812 879 456 937 956 294 679 793 041 871 997 527 756 416 857 217 752 008 146 589 220 290 946 179 243 180 824 825 088 220 182 091 480 544 872 557 618 626 218 382 472 446 905 682 568 443 009 524 153 017 695 039 429 835 456 312 255 734 387 359 399 353 256 674 753 602 399 004 223 017 299 513 665 163 734 760 114 880 896 154 760 654 411 352 865 752 269 065 180 473 493 221 316 613 037 972 024 945 245 649 095 119 645 836 854 271 401 292 810 924 160 285 593 428 511 002 207 895 128 629 862 853 708 189 137 044 278 769 634 391 162 054 011 069 795 371 475 232 403 866 084 849 896 947 018 852 869 025 231 100 827 080 451 951 695 355 294 426 263 107 822 318 857 933 207 716 854 908 911 291 043 620 940 374 829 272 062 414 888 470 322 899 339 833 471 475 133 464 576 850 290 332 404 912 809 509 262 097 240 885 875 596 853 249

A.5 Multiplication size = 4096

$N = 2^{4074} + 2^{595} + 1$

249 001 713 135 994 078 324 258 973 769 336 791 653 624 593 984 456 963 630 731 936 284 054 257 386 102 070 057 112 410 529 851 331 315 252 806 357 011 619 290 825 742 110 773 514 177 146 729 490 244 714 087 950 219 387 184 073 924 622 621 915 372 463 159 159 285 562 041 248 308 712 572 173 795 556 476 770 799 716 635 786 196 379 064 752 556 969 119 001 405 545 246 300 725 605 276 696 199 842 357 165 576 621 756 309 719 511 934 744 573 981 807 294 116 163 153 518 064 695 231 748 348 878 966 107 136 014 931 774 107 864 908 109 115 084 543 200 800 773 219 642 141 922 592 338 106 110 542 449 831 122 183 844 814 622 165 779 366 892 598 549 922 430 646 665 804 884 021 175 091 262 725 918 152 793 172 648 858 267 268 464 110 334 791 362 576 799 033 366 767

913 234 287 738 394 835 554 310 050 629 538 176 582 960 372 878 672 102 271
464 067 872 008 175 870 802 073 980 053 489 255 356 360 033 313 993 854 047
459 410 544 652 920 159 439 890 452 712 804 027 560 927 190 574 517 245 957
973 813 918 006 847 923 065 860 541 842 123 096 676 893 184 555 733 152 140
217 670 087 114 625 756 516 789 175 507 472 277 193 775 986 597 009 391 533
956 674 577 573 048 037 271 457 445 757 780 641 831 115 799 279 363 404 727
677 377 921 750 150 407 896 216 022 467 606 521 077 505 532 139 116 251 532
376 542 067 479 132 673 930 995 274 013 191 418 103 802 172 531 046 688 599
188 106 172 828 539 172 542 417 395 955 679 641 069 276 675 035 577 427 719
760 646 904 705 383 135 382 438 885 188 649 095 827 893 547 903 718 886 684
714 053 070 790 428 255 848 605 753 637 462 595 694 747 859 069 574 208 030
396 337 944 644 323 620 832 204 029 953

A.6 Multiplication size = 8192

$N = 2^{8145} + 2^{728} + 1$

7 750 231 643 082 485 742 460 962 148 454 817 554 808 927 312 110 943 521 132
588 686 766 392 043 019 772 694 494 408 217 204 309 517 171 319 696 133 174
838 856 731 062 862 885 892 624 909 301 425 458 031 069 687 381 622 956 884
467 176 122 834 274 907 839 071 721 949 250 045 030 032 452 235 549 622 262
738 873 889 659 188 068 186 906 408 048 285 754 242 101 646 693 186 200 819
786 797 249 106 810 745 561 450 215 960 487 324 579 098 393 772 117 674 521
719 451 059 408 010 322 072 818 487 646 150 460 643 526 173 893 871 456 459
336 467 096 064 105 965 175 632 321 304 421 569 426 101 101 024 972 404 941
849 271 455 164 557 969 029 069 763 548 687 313 664 936 669 722 349 353 380
347 509 379 274 327 180 874 639 266 454 278 903 109 547 858 443 717 982 509
125 882 373 246 703 317 594 463 947 060 856 137 280 577 318 302 776 076 201
372 955 744 039 830 403 165 544 427 894 116 559 019 184 337 406 011 676 040
474 726 562 517 756 345 272 319 925 653 131 961 817 531 748 659 680 100 423
129 114 764 953 424 044 104 277 579 741 289 456 682 951 879 218 919 920 488
585 381 331 309 214 444 452 537 858 950 923 824 717 507 365 751 424 240 450
454 464 557 080 064 752 460 367 291 468 502 803 528 018 669 713 395 318 303

862 232 642 552 390 611 196 129 485 964 323 450 407 157 009 784 534 357 632
506 245 996 430 233 868 608 126 793 832 317 541 519 210 503 694 101 223 599
686 223 867 362 936 080 799 531 050 677 026 357 175 506 780 653 063 824 682
197 874 805 170 801 223 792 880 670 303 071 706 935 826 452 922 185 610 324
316 199 882 647 690 626 341 080 645 126 744 589 039 297 131 932 267 978 840
335 840 271 514 845 737 820 372 691 231 603 763 241 349 211 473 334 493 434
482 502 962 657 532 554 599 710 677 319 354 302 316 738 352 364 582 536 366
983 648 835 613 549 161 320 989 624 845 809 512 134 300 175 060 656 217 587
906 090 375 419 647 994 013 312 512 782 586 476 711 678 445 504 661 352 524
802 974 335 248 779 681 927 748 467 435 722 289 631 107 669 594 160 983 842
446 253 582 917 985 535 869 077 770 656 963 150 932 595 696 595 792 796 153
241 565 437 076 315 977 420 077 011 593 906 652 754 206 412 661 295 774 373
329 547 648 077 892 881 408 714 615 329 566 414 609 384 160 827 121 065 690
959 320 505 282 032 818 012 740 186 387 849 657 380 059 165 005 992 797 170
262 732 030 783 020 756 432 483 041 706 595 522 434 205 882 342 418 811 081
658 607 682 216 871 640 801 661 815 411 812 163 077 511 952 888 724 044 727
614 717 790 158 721 060 793 630 594 109 978 698 015 944 177 872 722 550 666
711 996 224 744 690 034 205 762 907 967 870 457 105 904 857 939 497 702 187
205 356 840 262 058 198 154 837 004 341 125 490 953 072 419 173 118 405 504
899 241 392 580 378 666 042 303 156 202 166 970 644 754 683 324 172 390 077
362 876 638 553 402 777 138 479 701 767 206 577 506 914 403 828 273 705 441
308 731 404 289 942 287 447 227 108 332 789 699 488 908 193 990 316 874 267
927 954 122 180 500 166 556 335 696 655 244 218 656 989 840 071 948 206 538
272 466 283 209 009 709 728 804 319 138 320 958 804 949 898 176 916 206 200
138 621 555 322 735 185 204 019 655 349 631 313 609 923 028 442 787 489 582
025 199 663 919 653 337 825 957 758 500 524 441 788 335 083 079 315 305 932
938 037 586 087 289 110 553 955 948 231 988 842 656 274 011 986 264 588 289

APPENDIX B

OPENCL CODE

```
1   #define SIZE    32
2
3   ulong mult(uint A, uint B)
4   {
5       ulong _A, _B, _C;
6       _A = A; _B = B; _C = 0;
7
8       #pragma unroll
9       for (int i = 0; i < 32; i++)
10      {
11          if ((_B & 0x1) == 0x1) _C ^= _A;
12          _A <<= 1;
13          _B >>= 1;
14      }
15      return _C;
16  }
17
18  __kernel
19  __attribute__((reqd_work_group_size(SIZE, 1, 1)))
20  __attribute__((num_simd_work_items(4)))
21  void Montgomery (
22      __global const uint * restrict A,
23      __global const uint * restrict B,
24      __global const uint * N,
25      __global const uint * N_,
26      __global uint * tl,
27      __global uint * m,
28      __global uint * C)
29  {
30      const int sizeR = SIZE + 1;
31
32      size_t i = get_global_id(0);
33      size_t j = 0;
34
35      ulong tmp;
36
37      barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE);
38      // ****************** [ t = A * B ] ****************** //
39      for(j = 0; j<SIZE; j++)
40      {
41          tmp = mult(A[i], B[j]);
42          tl[i+j] ^= (tmp & 0xFFFFFFFF);
43          barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE);
44          tl[i+j+1] ^= (tmp >> 32);
45          barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE);
46      }
47
48      // ****************** [ m = tN_ ] ****************** //
49      for(j = 0; j<SIZE*2; j++)
50      {
51          tmp = mult(tl[j], N_[i]);
52          m[i+j] ^= (tmp & 0xFFFFFFFF);
53          barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE);
54          m[i+j+1] ^= (tmp >> 32);
55          barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE);
56      }
57
58      // *************** [ m = tN_ mod R ] *************** //
59      //if(i + sizeR < 2 * SIZE) m[i+sizeR] = 0;
60      //barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE);
61
62      // ***************** [ u = t ^ mN ] ***************** //
63      for(j = 0; j<sizeR; j++)
64      {
65          tmp = mult(m[j], N[i]);
66          tl[i+j] ^= (tmp & 0xFFFFFFFF);
67          barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE);
68          tl[i+j+1] ^= (tmp >> 32);
69          barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE);
70      }
71
72      C[i] =tl[i+sizeR];
73      barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE);
74  }
```