Accelerating Homomorphic Encryption in the Cloud Environment through High-Level Synthesis and Reconfigurable Resources

Michael J. Foster
mf1656@rit.edu

Follow this and additional works at: https://scholarworks.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
Accelerating Homomorphic Encryption in the Cloud Environment through High-Level Synthesis and Reconfigurable Resources

by

Michael J. Foster

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Engineering

Supervised by
Dr. Marcin Lukowiak
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
November 2017

R·I·T | KATE GLEASON
College of ENGINEERING

Approved by:

Dr. Marcin Lukowiak Advisor
Department of Computer Engineering

Dr. Stanislaw Radziszowski Advisor
Department of Computer Science

Dr. Dhireesha Kudithipudi
Department of Computer Engineering
To my parents, Bill and Debby, and my brother, Tim. Thank you for believing in me, supporting me, and praying for me.
Acknowledgments

First of all, I would like to thank my Lord and Savior, Jesus Christ, for all His blessings. I have amazing family and friends. I have been blessed with excellent educational and career opportunities. Most importantly, I have been blessed with the hope that whether I live or die, God is with me, He has forgiven me, He loves me, and, by His grace alone, I will spend eternity with Him (Hebrews 6:19).

I owe a huge debt of gratitude to my parents, Bill and Debby Foster. Thank you for supporting me in every way possible through all these years. You have done everything you possibly could do to help me succeed, and I am eternally grateful. I would also like to thank my brother Tim and his family: Lizzie, Caleb, Sadie, and Emarie. Thank you for all the joy you bring us. I am very proud to be your brother, brother-in-law, and uncle.

I would like to thank the professors and students that I worked with at the Nanocomputing Lab: Dr. Dhireesha Kudithipudi, Dr. Ernest Fokoue, James Mnatzagianian, Nick Soures, Anvesh Polepalli, Sahuvi Praveen, Qutaiba Saleh, Amanda Hartung, Luke Boudreau, Dr. Cory Merkel, and others. Thank you for helping and encouraging me through my first steps into graduate research.

Thank you to all my colleagues and professors working in the Applied Cryptography and Information Security (ACIS) lab: Dr. Michael Kurdziel, Scott Thompson, Daniel Stafford, Chun-Yi Chu (“Ryan”), Cody Tinker, Prathibha Rama, and others. I have enjoyed working alongside you all as we pursue our love of security and computers.

I would like to thank Rick Tolleson for giving me access to the computers I needed and for help with posters and employment. I am truly grateful for your help.

I am also extremely grateful to the staff and engineers at Xelic Inc. for giving me the opportunity to sharpen my skills among so many experienced and kind people. Kenny, thank you for being such a great mentor and for encouraging me to pursue my Master’s. I hope our paths will cross again.

I would like to thank my mathematics professors: Dr. David Barth-Hart and Dr. Anurag Agarwal. Thank you for sharing the beauty of mathematics with me.

Thank you to all the researchers I have met through the Homomorphic Encryption (SIRA) research group: Dr. Peizhao Hu, Dr. Sumita Mishra, Dr. Matthew Wright, Asma Aloufi, Andrew Dunn, and others. Thank you for sharing with me more of the computer science side of cryptography.

Finally, I would like to thank my advisors and committee members. Dr. Radziszowski, thank you for sharing your love of cryptography and mathematics with me. Dr. Kudithipudi, thank you for supporting me through my growth as a graduate student and for encouraging me to achieve my highest potential. Dr. Lukowiak, thank you for introducing me to cryptography and for supporting me throughout the research process.
Abstract

The recent surge in cloud services is revolutionizing the way that data is stored and processed. Everyone with an internet connection, from large corporations to small companies and private individuals, now have access to cutting-edge processing power and vast amounts of data storage. This rise in cloud computing and storage, however, has brought with it a need for a new type of security. In order to have access to cloud services, users must allow the service provider to have full access to their private, unencrypted data. Users are required to trust the integrity of the service provider and the security of its data centers. The recent development of fully homomorphic encryption schemes can offer a solution to this dilemma. These algorithms allow encrypted data to be used in computations without ever stripping the data of the protection of encryption. Unfortunately, the demanding memory requirements and computational complexity of the proposed schemes has hindered their wide-scale use. Custom hardware accelerators for homomorphic encryption could be implemented on the increasing number of reconfigurable hardware resources in the cloud, but the long development time required for these processors would lead to high production costs. This research seeks to develop a strategy for faster development of homomorphic encryption hardware accelerators using the process of High-Level Synthesis. Insights from existing number theory software libraries and custom hardware accelerators are used to develop a scalable, proof-of-concept software implementation of Karatsuba modular polynomial multiplication. This implementation was designed to be used with High-Level Synthesis to accelerate the large modular polynomial multiplication operations required by homomorphic encryption. The accelerator generated from this implementation by the High-Level Synthesis tool Vivado HLS achieved significant speedup over the implementations available in the highly-optimized FLINT software library.
## Contents

Dedication ii  

Acknowledgments iii  

Abstract iv  

1 Introduction 4  

1.1 The Problem ................................. 4  
1.2 Homomorphic Encryption .......................... 4  
1.3 High-Level Synthesis ............................... 6  
1.4 This Work ................................. 8  

2 Mathematical Software Libraries and Algorithms 9  

2.1 Software Libraries ................................. 9  
2.1.1 GMP ........................................ 9  
2.1.2 MPIR ....................................... 10  
2.1.3 MPFR ....................................... 11  
2.1.4 LibTom .................................... 11  
2.1.5 PARI ....................................... 12  
2.1.6 LiDIA ....................................... 12  
2.1.7 NTL ....................................... 13  
2.1.8 FLINT ..................................... 13  
2.1.9 MIRACL .................................... 14  
2.2 Software Multiplication Algorithms .................. 14  
2.2.1 Integer or Polynomial Multiplication? ............... 14  
2.2.2 Algorithm Selection Process ......................... 16  
2.2.3 Schoolbook Multiplication ......................... 17
## CONTENTS

2.2.4 Karatsuba Multiplication Algorithm .......................... 18  
2.2.5 Toom Multiplication Algorithm .............................. 19  
2.2.6 FFT/NTT Multiplication Algorithm ............................ 19  
2.2.7 Comba Scheduling ........................................... 20  

2.3 Software Modular Reduction ..................................... 21  
2.3.1 Barrett Reduction ........................................... 22  
2.3.2 Montgomery Reduction ...................................... 22  

### 3 High-Level Synthesis of Karatsuba Multiplication 24  
3.1 Karatsuba Multiplication ....................................... 25  
3.2 Recursion and High-Level Synthesis ............................. 26  
3.3 Leveraging the Karatsuba Tree Structure ....................... 27  
  3.3.1 Representing the Tree .................................... 27  
  3.3.2 Calculating the Leaf Product Operands .................... 28  
3.4 Vivado HLS Synthesis Directives and Data Types ............. 30  
  3.4.1 UNROLL Directive ....................................... 30  
  3.4.2 PIPELINE Directive ..................................... 30  
  3.4.3 DATAFLOW Directive ................................... 31  
  3.4.4 ARRAY_PARTITION Directive ............................. 32  
  3.4.5 CLOCK Directive ....................................... 33  
  3.4.6 DEPENDENCE Directive .................................. 34  
  3.4.7 Arbitrary Precision Data Types ........................... 34  
3.5 Version 1: A Single Memory Structure .......................... 34  
  3.5.1 Version 1: File Structure, Control Constants, and Verification 34  
  3.5.2 Version 1: Algorithm .................................... 36  
  3.5.3 Version 1: Data Flow .................................... 40  
  3.5.4 Version 1: High-Level Synthesis Results ................ 41  
3.6 Version 2: Separate Leaf Multiplication and Subtraction .... 48
3.6.1 Version 2: Data Flow ........................................ 48
3.6.2 Version 2: High-Level Synthesis ............................ 49

3.7 Version 3: Independent Final Product Assignment .......... 53
3.7.1 Version 3: Data Flow ...................................... 53
3.7.2 Version 3: High-Level Synthesis ............................ 53

3.8 Version 4: Pipelined Internal Memory ........................ 56
3.8.1 Version 4: Algorithm .................................... 56
3.8.2 Version 4: Data Flow ...................................... 58
3.8.3 Version 4: High-Level Synthesis ............................ 59

3.9 Version 5: Experimenting with FIFOs ......................... 61
3.9.1 Version 5: Data Flow ...................................... 62
3.9.2 Version 5: High-Level Synthesis ............................ 62

3.10 Version 6: Switch to Two-dimensional Arrays .............. 64
3.10.1 Version 6: Algorithm .................................... 64
3.10.2 Version 6: Data Flow ...................................... 64
3.10.3 Version 6: High-Level Synthesis ............................ 65

3.11 Version 7: Optimized Section Loop ......................... 67
3.11.1 Version 7: Algorithm .................................... 67
3.11.2 Version 7: High-Level Synthesis ............................ 69

3.12 Addition of Modular Reduction ................................ 70
3.13 Parallelization .............................................. 70
3.14 Hardware Accelerator vs. FLINT .............................. 75

4 Conclusions and Future Work .................................. 77

A karatsuba_combine_2d_parallel9_modCoeff Synthesis Directives 79

Bibliography .................................................. 81
Chapter 1

Introduction

1.1 The Problem

The availability of cloud computing services is continuing to expand. Companies such as Nimbix[1], IBM[2], and Amazon[3] are now giving anyone with an internet connection access to massive computing power. Of course, this access comes at a cost. Although some cloud storage services offer end-to-end data encryption[4], cloud computing services require that the user’s data be given to the service provider’s computers in an unencrypted form. This may not be a barrier for some users, but for researchers who wish to work with personal data such as medical records or companies who wish to run proprietary algorithms, the risk is significant. These users are at the mercy of the cloud service providers. They must rely upon the integrity of the service providers and the security of the provider’s systems. In 2009, a crucial piece of the solution to this problem arose in the form of homomorphic encryption [5].

1.2 Homomorphic Encryption

In September of 2009, Craig Gentry, who was a Ph.D. student at Stanford at the time, submitted his dissertation. This document described his work on the first fully homomorphic encryption scheme [5]. Since the publishing of this groundbreaking work, mathematicians have continued to propose, refine, and evaluate Homomorphic
CHAPTER 1. INTRODUCTION

Figure 1.1: The process flow for using homomorphic encryption with a cloud service. “PT” represents plaintext, and “CT” represents ciphertext. The “Encrypt,” “Evaluation,” and “Decrypt” operations are all defined by the HE scheme being used.

Encryption (HE) schemes [6][7][8][9][10].

Homomorphic encryption allows operations to be performed on encrypted data. For example, as shown in Figure 1.1, if a user wants to perform an operation with two plaintext operands, the operands can be encrypted, and then the homomorphic version of the operation can be performed using only the encrypted versions of the operands. The homomorphic operation will return an encrypted output. The decrypted version of this output is equivalent to the output which would have been received if the normal operation was performed on the unencrypted operands. When applied to cloud computing, this type of encryption would allow a user to utilize a cloud computing service without ever giving the service provider access to the unencrypted inputs, intermediate values, or outputs for any operation that it performs for the user. The user’s data is never stripped of the protection of encryption.

HE schemes are often classified into three major groups: partially homomorphic, somewhat homomorphic, and fully homomorphic [11]. Partially homomorphic schemes can perform either addition or multiplication homomorphically but not both. Examples of partially homomorphic schemes include El Gamal[12] and Paillier[13]. Somewhat homomorphic schemes may support one or both of these fundamental homomorphic operations, but these operations can only be performed consecutively a limited number of times. A fully homomorphic scheme can perform an unlimited number of consecutive homomorphic addition and multiplication operations. The
CHAPTER 1. INTRODUCTION

Figure 1.2: FPGAs become a part of the cloud environment. The unique capabilities of these new devices complement the strengths of the CPUs and GPUs already in the cloud.

development of a fully homomorphic scheme, therefore, greatly expanded the number of possible homomorphic operations which could be performed. If the benefits of HE are to be fully leveraged in the cloud, a fully homomorphic scheme must be used.

1.3 High-Level Synthesis

Presently, all HE schemes have high computing resource requirements, and fully homomorphic schemes are especially demanding. This computational complexity has hindered the widespread use of HE. These demands may lessen as HE schemes continue to be developed, refined, and tested, but, in the meantime, the issues posed by the computational complexity of current HE schemes is being addressed using a special family of powerful computing devices, Field-programmable Gate Arrays (FPGAs). FPGAs offer both a high degree of programmability and rapid execution of highly-parallel computational loads. These devices have been used to accelerate HE [14][15][16][17], and, conveniently, as shown in Figure 1.2, they are starting to show up in the cloud [2][18].

If hardware acceleration of HE is to be widely used in the cloud, the hardware implementation process must be cost-effective and accessible to both software programmers and hardware engineers. As illustrated in Figure 1.3, before the development of High-Level Synthesis (HLS), there were two primary methods for executing algorithms on computers. The first was to describe the algorithm in a software pro-
CHAPTER 1. INTRODUCTION

Figure 1.3: Design flow enabled by High-Level Synthesis.

programming language and compile the description for execution on a CPU. The second was to model the digital circuit which would execute the algorithm. This second option required the use of hardware description languages such as Verilog or VHDL. These hardware descriptions were compiled into hardware accelerators which could be implemented as Application-Specific Integrated Circuits (ASICs) or programmed onto FPGAs. Traditionally, only hardware engineers had the expertise to develop such accelerators. Also, since traditional accelerator descriptions require a high level of detail and customization for each task, the development time and cost for these accelerators can be very high compared to their software counterparts. These issues could hinder the use of evolving HE algorithms if it were not for another evolving technology called High-Level Synthesis (HLS).

HLS tools take software descriptions of algorithms (usually written in C or C++) and convert them into hardware accelerators. This is not a new idea. These types of tools have been around for the last 30 years, but previous generations of the tools were difficult to use, and the results were often poor [19]. As the third generation of HLS tools evolves, their results are becoming much more promising.

Although it may seem improbable that an accelerator generated by a tool could
compete with one designed by an experienced hardware engineer, recent case studies have shown that currently available tools are getting close to being competitive with handwritten RTL [20][21]. These competitive HLS implementations, however, must be written with an understanding of the desired final circuit and be carefully optimized with synthesis directives. This notable difference between software written for a CPU and software written for HLS requires the creation of HLS-optimized versions of existing software algorithms.

1.4 This Work

This work seeks to show that a high level of privacy and security is possible in the cloud if HE, FPGAs, and HLS are used together. This is done by taking the first steps toward the development of a proof-of-concept software library which contains fundamental mathematical algorithms required by HE schemes. The uniqueness of this library lies in the fact that its software implementations are specifically tuned for use with a state-of-the-art HLS tool, Xilinx’s Vivado HLS.

This work studies the design process of the first entry into the HLS software library for HE, large polynomial multiplication. The computational complexity of the huge polynomial multiplications is a significant bottleneck in most HE schemes. Any implementation of HE in the cloud can benefit from the acceleration of this fundamental operation.
Before a software library for HLS can be developed, the currently available mathematical software libraries must be studied in order to gain insights about the algorithms commonly used for certain operations and the rational behind the selection of an algorithm given a specific set of operands. This chapter will give a brief summary of some of the most well-known, currently-available mathematical software libraries, in-depth analysis of their algorithm selection processes for multiplication operations, and an overview of commonly used multiplication and modular reduction algorithms.

### 2.1 Software Libraries

This section will briefly introduce a collection of some of the more well-known arithmetic software libraries. Although this is not a complete list of all the currently available libraries, those mentioned are a representative subset of a vast and ever expanding collection of mathematical software libraries.

#### 2.1.1 GMP

The GNU Multiple Precision Arithmetic (GMP) library [22] is one of the most well-known mathematical software libraries. It is an open-source project which continues to be regularly updated at the time of writing. It is written in C, but it includes C++ interfaces. The project’s website (gmplib.org) states that its main target applications
are “cryptography applications and research, Internet security applications, algebra systems, computational algebra research, etc.” [22]. The website also has the following to say about the development methodology of the library: “GMP is carefully designed to be as fast as possible, both for small operands and for huge operands. The speed is achieved by using fullwords as the basic arithmetic type, by using fast algorithms, with highly optimised assembly code for the most common inner loops for a lot of Central Processing Units (CPUs), and by a general emphasis on speed”. The GMP library is divided into five functional groups: “High-level signed integer arithmetic functions (mpz)”, “High-level rational arithmetic functions (mpq)”, “High-level floating-point arithmetic functions” (mpf or newer mpfr), “C++ class based interface to [the first three]”, and “Low-level positive-integer, hard-to-use, very low overhead functions (mpn)”.

2.1.2 MPIR

The Multiple Precision Integers and Rationals (MPIR) library [23] is a fork of GMP. As such, its library structure and documentation are almost identical to those of GMP. Like GMP, it is written in ANSI C and assembly with available C++ Application Programming Interfaces (APIs). Although MPIR’s main website (mpir.org) does not give an reason for the fork from GMP, there are some web articles which shed some light on its development. According to one web article, MPIR is “optimized for different operating systems and C compilers” [24]. Another web article states that the reason for the fork was the “active refusal of the GMP team to support Microsoft platforms” [25]. This article also states that MPIR is designed “to be the fastest for those who are limited by basic types of C” [25].
2.1.3 MPFR

The GNU Multiple Precision Floating-point Rounding (MPFR) library [26] is an opensource, C-language library based on GMP. It has third-party C++ interfaces as well as interfaces for other programming languages such as Perl, Python, LISP, Java, and Ada. According to its website (mpfr.org), MPFR is “a C library for multiple-precision floating-point computations with correct rounding. The main goal of MPFR is to provide a library for multiple-precision floating-point computation which is both efficient and has a well-defined semantics. It copies the good ideas from the ANSI/IEEE-754 standard for double-precision floating-point arithmetic (53-bit significand)”. MPFR has been used in MATLAB and Octave packages as well as in Maple, Sage, and the FLINT number theory software library.

2.1.4 LibTom

The LibTom collection of mathematical software libraries is the personal project of educator Thomas St. Denis [27]. It is written in “portable C” [27]. The libraries started as a personal project and then later became a well-documented set of libraries which support “a variety of cryptographic and algebraic primitives designed to enable developers and students to pursue the field of cryptography much more efficiently”. LibTom is composed of five separate libraries: LibTomCrypt, LibTomMath, TomsFastMath, LibTomPoly, and LibTomFloat. LibTomCrypt is “a fairly comprehensive, modular and portable cryptographic toolkit that provides developers with a vast array of well known published block ciphers, one-way hash functions, chaining modes, pseudo-random number generators, public key cryptography and a plethora of other routines”. LibTomMath is “a free open source portable number theoretic multiple-precision integer library written entirely in C” which can ideally “serve as an educational tool in the future for CS students studying number theory”. TomsFastMath is “a fast public domain, open source, large integer arithmetic library
written in portable ISO C. It is a port of LibTomMath with optional support for inline assembler multipliers”. LibTomPoly “provides access to polynomials of finite characteristic of varying degrees (it will grow to accommodate)” and a small set of algebraic operations. Finally, LibTomFloat provides functions for floating-point numbers. The documentation for the LibTom libraries seems to have been written around 2010. The libraries do not appear to have received any significant updates since this main submission.

2.1.5 PARI

PARI/GP is “a specialized computer algebra system, primarily aimed at number theorists, but has been put to good use in many other different fields, from topology or numerical analysis to physics” [28]. PARI, by itself, is a C-language mathematical software library. Similar to the libraries already mentioned, PARI is also C++ compatible. Development of this library started in the mid-80s and continues to present day.

2.1.6 LiDIA

Developed by the LiDIA Group at the Darmstadt University of Technology, the LiDIA library is “a C++ library for computational number theory which provides a collection of highly optimized implementations of various multiprecision data types and time-intensive algorithms” [29]. It is one of the few libraries which is primarily written in C++. Although the original project website is no longer active, recovered versions of this library can be found on GitHub[30] and the website for the Department of Computer Science at Stony Brook University[29].
CHAPTER 2. MATHEMATICAL SOFTWARE LIBRARIES AND ALGORITHMS

2.1.7 NTL

Number Theory Library (NTL) is an award-winning C++ number theory software library developed by Victor Shoup[31]. It started as a modification of the Long Integer Package (LIP) created by Arjen K. Lenstra[32]. The project website (shoup.net/ntl) describes it as “a high-performance, portable C++ library providing data structures and algorithms for arbitrary length integers; for vectors, matrices, and polynomials over the integers and over finite fields; and for arbitrary precision floating point arithmetic” which can be used in conjunction with GMP. It includes functions for fast and robust polynomial factorization, order determination for elliptic curves, and lattice reduction. This library is still regularly updated and forms the bottom layer of the homomorphic encryption library HElib.

2.1.8 FLINT

Fast Library for Number Theory (FLINT) is a C-language number theory software library which depends upon the GMP, MPIR, and MPFR libraries[33]. It is used by the mathematical software Sage for “for polynomial arithmetic over \( \mathbb{Z} \), \( \mathbb{Q} \) and \( \mathbb{Z}/n\mathbb{Z} \) for small \( n \)”[33] and is the primary library used by the Scarab homomorphic encryption project[34]. The library’s website (flintlib.org) states that “FLINT supports arithmetic with numbers, polynomials, power series and matrices over many base rings, including: Multiprecision integers and rationals, Integers modulo \( n \), p-adic numbers, Finite fields (prime and non-prime order), Real and complex numbers (via the Arb extension library)”[33]. Supported operations include “conversions, arithmetic, computing GCDs, factoring, solving linear systems, and evaluating special functions” along with “various low-level routines for fast arithmetic”.

13
MIRACL is a software library which, according to its GitHub page, “is widely regarded by developers as the gold standard open source SDK for elliptic curve cryptography (ECC)”\(^{[35]}\). Its main purpose is to enable “developers to build security into highly constrained environments, including embedded, mobile apps and SCADA”. It is a C-language library with available C++ interfaces.

### 2.2 Software Multiplication Algorithms

One of the most computationally expensive operations required in homomorphic encryption algorithms is the modular multiplication of large polynomials. For example, in the YASHE homomorphic scheme, there can be polynomials with 1,024 to 16,384 coefficients where coefficient sizes can range from 20 to 1,550 bits \(^{[36]}\). Because of this, an important part of any HE software library is its modular polynomial multiplication implementation. Before this implementation can be created, the best multiplication algorithm for the task must be selected. This work began with an in-depth study of the source code and documentation of the multiplication implementations of several of the well-known mathematical software libraries.

#### 2.2.1 Integer or Polynomial Multiplication?

Software libraries use many of the same algorithms for both polynomial and integer multiplication. This may seem confusing at first, but a simple example can clear up any confusion.

Suppose the integers 12 and 34 are being multiplied by hand. Using the method taught in grade school, the numbers would be written with one on top of the other,
and the following partial products would be calculated:

\[ 4 \times 12 = 48 \] \hspace{1cm} (2.1a)
\[ 3 \times 12 = 36 \] \hspace{1cm} (2.1b)

The second partial product is scaled up by the base (10 in this example), and the partial products are added together to get the final product.

\[ 48 + 360 = 408 \] \hspace{1cm} (2.2)

Suppose the digits in the operands of this example were coefficients in polynomials. In other words, instead of calculating \((12 \times 34)\), calculate \((x+2)(3x+4)\). The process is very similar to that shown above. The partial products are calculated and then summed.

\[ 4(x + 2) = 4x + 8 \] \hspace{1cm} (2.3a)
\[ 3(x + 2) = 3x + 6 \] \hspace{1cm} (2.3b)
\[ (4x + 8) + (3x + 6)x = 3x^2 + 10x + 8 \] \hspace{1cm} (2.3c)

Notice that this calculation is identical to the integer example except for carry rule applied during the summation of the partial products. In the integer example, the sum of 4 and 6 in the tens’ place creates a carry which is added into the hundreds’ place. In the polynomial example, the coefficients are independent, allowing the \(x\) coefficient to expand to two digits without affecting the \(x^2\) coefficient. Just like in the schoolbook multiplication algorithm shown in this example, other multiplication algorithms can easily be transformed from integer multiplication to polynomial mul-
CHAPTER 2. MATHEMATICAL SOFTWARE LIBRARIES AND ALGORITHMS

tiplication algorithms by simply changing the carry rules used during the summation of partial products.

Some multiplication algorithms, such as Karatsuba, are best explained using examples with polynomial operands. In practice, however, they can be used with both integer and polynomial operands. Please note that, in the algorithm descriptions given in this chapter, all of the algorithms described can be used for both integer and polynomial multiplication.

2.2.2 Algorithm Selection Process

Mathematical software libraries often contain implementations of multiple multiplication algorithms under-the-hood. When their top-level multiplication function is called, a highly optimized selection process is run to determine the best algorithm for the given task. In this section, the selection process and multiplication algorithms used by GMP and MPIR libraries will be explained. This selection process is very similar to those used in other mathematical libraries.

The algorithm used for integer multiplication in the GMP and MPIR libraries is selected based upon the size of the operands. For example, when the top-level integer multiply function, mpz_mul(), is called, it begins comparing the size of the operands to some predefined thresholds. These thresholds represent the operand size at which the algorithm given in its name should start being used. For example, the threshold MUL_KARATSUBA_THRESHOLD represents the minimum operand size at which the Karatsuba multiplication algorithm should be used. These thresholds are defined in header files which are specific to the CPU architecture being used to run the algorithm.

In GMP and MPIR, operand sizes and thresholds are expressed in units of limbs. A limb is defined as “the part of a multi-precision number that fits in a single word”[37]. This term was chosen “because a limb of the human body is analogous to
## 2.2.3 Schoolbook Multiplication

The most well-known algorithm for multiplication is the method taught in grade school. The two operands are written with one above the other, and each digit in the bottom number is successively multiplied by all the digits in the top number (with carries applied). This algorithm requires no pre- or post-computations and has a complexity of $O(n^2)\textsuperscript{[39]}$. Despite the existence of much more complex algorithms, this algorithm is still regarded by many software library developers as the optimal algorithm for the multiplication of “small” numbers (less than about 1,000 bits).

### Table 2.1: Operand size thresholds for multiplication using MPIR\textsuperscript{[37]}.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Threshold for a Generic Architecture with 32-bit Limbs (bits)</th>
<th>Threshold for x86_64 Haswell with 64-bit Limbs (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basecase (Schoolbook)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Karatsuba</td>
<td>1,024</td>
<td>1,024</td>
</tr>
<tr>
<td>Toom-3</td>
<td>4,096</td>
<td>6,720</td>
</tr>
<tr>
<td>Toom-4</td>
<td>9,600</td>
<td>15,744</td>
</tr>
<tr>
<td>Toom-8.5</td>
<td>12,832</td>
<td>19,392</td>
</tr>
<tr>
<td>FFT</td>
<td>128,320</td>
<td>249,856</td>
</tr>
</tbody>
</table>

A digit, only larger, and containing several digits”\textsuperscript{[37]}. A limb is further defined as “an unsigned machine-integer type”\textsuperscript{[38]} which normally contains 32 or 64 bits\textsuperscript{[37]}. Table 2.1 shows the operand size thresholds for multiplication using MPIR. The best performance is achieved when MPIR is configured for the target processor’s specific architecture, but if the architecture is not selected or detected, generic values are used. The generic value can therefore give an idea of the magnitude of the best value, but it is not exact.
2.2.4 Karatsuba Multiplication Algorithm

The Karatsuba algorithm offers an optimization for polynomial multiplication that allows it to out-perform the schoolbook method for sufficiently large operands. In order to clearly explain this optimization, an example of algorithm execution will be given.

Let $A$ and $B$ be two polynomials of degree $n - 1$ with integer coefficients. To simplify this example, suppose $n$ is even. The first step in the algorithm is to divide $A$ and $B$ into polynomials half their size referred to as $A_H$, $A_L$, $B_H$, and $B_L$, respectively. An example of this is shown below:

\begin{align*}
A &= x^3 + 2x^2 + 3x + 4 \quad (2.4a) \\
A_L &= 3x + 4 \quad (2.4b) \\
A_H &= x + 2 \quad (2.4c)
\end{align*}

$A_H$ takes the half of the coefficients of $A$ which are multiplied by the higher powers of $x$, while $A_L$ takes the half of the coefficients which are multiplied by the lower powers of $x$. The operands can therefore be written as the following:

\begin{align*}
A &= A_L + A_H x^{n/2} \quad (2.5a) \\
B &= B_L + B_H x^{n/2} \quad (2.5b)
\end{align*}

When these two terms are multiplied out, they produce the following:

\begin{equation}
A \times B = A_L B_L + (A_L B_H + A_H B_L) x^{n/2} + A_H B_H x^n \quad (2.6)
\end{equation}

Notice that the term $(A_L B_H + A_H B_L)$ requires two multiplications and one addition.
CHAPTER 2. MATHEMATICAL SOFTWARE LIBRARIES AND ALGORITHMS

One multiplication can be removed, however, with some rearranging:

\[(A_LB_H + A_HB_L) = (A_L + A_H)(B_H + B_L) - A_LB_L - A_HB_H\]  \hspace{1cm} (2.7)

One multiplication has been removed, but one addition and two subtractions have been added. The two additions \((A_L + A_H \text{ and } B_H + B_L)\) are considered to be the pre-computations required by the algorithm, and the two subtractions \((-A_LB_L - A_HB_H)\) are the post-computations[39]. When the operands get large enough that one addition and two subtractions are faster to execute than one multiplication, Karatsuba becomes a better choice than the schoolbook method. In terms of complexity, Karatsuba is \(O(n^{1.58})\)[39].

2.2.5 Toom Multiplication Algorithm

The Toom algorithm is a more generalized version of the Karatsuba algorithm which was formulated by A. L. Toom [40][41]. In some libraries, a version of Toom optimized by S. L. Cook, the Toom-Cook algorithm [42], is used. The Toom-\(n\) algorithm breaks each operand down into \(n\) pieces. Toom-2 is essentially the Karatsuba algorithm. For \(n\) greater than 2, interpolation operations are required to calculate partial products [37]. The overhead associated with these operations will only result in a speedup over Karatsuba for sufficiently large operands. For example, MPIR starts using Toom-Cook for integer multiplication when operands are around 4,000 to 6,000 bits long, depending upon the CPU architecture and limb size [37].

2.2.6 FFT/NTT Multiplication Algorithm

The most common algorithm used for the multiplication of very large integers is the Schönhage-Strassen Fast Fourier Transform (FFT) algorithm [43]. This algorithm starts by breaking the operands into an equal number of pieces. These pieces are
then put into the FFT domain “by treating each word as an sample in the time domain”[44] and calculating their FFT. Each converted piece is then multiplied by its counterpart from the other operand, and the resulting partial products are taken out of the FFT domain using an inverse FFT. In the final step, the partial products are combined to form the final product. The overhead associated with transforming values into and out of the FFT domain outweighs the speedup associated with the smaller, parallel multiplications in the FFT domain unless the operands are very large. For example, MPIR does not use the FFT multiplication algorithm for integer multiplication until the size of each operand is well over 100,000 bits [37].

In the literature related to homomorphic encryption, the FFT multiplication algorithm is sometimes referred to as the Number Theoretic Transform (NTT). When the FFT algorithm is used with elements of fields or rings, such as in homomorphic encryption, it is called the NTT [41][15].

2.2.7 Comba Scheduling

Comba scheduling is an optimization technique for the schoolbook multiplication algorithm. It is described by its namesake, Paul G. Comba, in [45] where it is referred to as “Optimization in the small”[45]. It has been used in several hardware accelerators for homomorphic encryption ([46],[47]) as well as in TomsFastMath[48], a sub-library of the LibTom mathematical software library collection.

Figure 2.1 illustrates this algorithm. When the schoolbook algorithm is used, the
CHAPTER 2. MATHEMATICAL SOFTWARE LIBRARIES AND ALGORITHMS

partial products are calculated one row at a time. For example, in the figure, the first partial products to be calculated would be $a_0b_0$, $a_1b_0$, and $a_2b_0$. In the Comba algorithm, the partial products are calculated and accumulated by column. In the figure, the columns are numbered in the order that they would be processed, from #1 to #5. This process minimizes memory accesses when the algorithm is executed on a CPU[45]. After all the partial products in a column are calculated and summed, the carry terms are saved in registers and the sum is written to memory. If the schoolbook algorithm was used, partial products from previous rows would need to be read from memory during the final accumulation step. The Comba algorithm allows the accumulation step to occur when the partial products to be summed (the products that share the same column) are still in quickly-accessible registers instead of memory.

2.3 Software Modular Reduction

The two most popular modular reduction algorithms discussed in the literature and used in software libraries are the Barrett and Montgomery algorithms. Some implementations of modular polynomial multiplication embed modular reduction in the multiplication implementation as an optimization; however, some software implementations, such as the one in the FLINT library, reduce the polynomial coefficients in a separate step after the full multiplication has been completed [33]. In the Karatsuba polynomial multiplication implementation presented in this work, a Barrett modular reduction step was incorporated into the multiplication implementation in order to limit coefficient growth and thereby minimize the memory requirements of the implementation.
2.3.1 Barrett Reduction

Barrett reduction is the simpler of the two reduction algorithms. In order to calculate $t \mod M$, the following two pre-computations are required [44]:

\[
q = 2^\lceil \log_2(M) \rceil \quad \text{(2.8a)}
\]

\[
\mu = \left\lfloor \frac{2^q}{M} \right\rfloor \quad \text{(2.8b)}
\]

Once these pre-computed values are calculated, input $t$ can be transformed to the reduced value $r$ using Equation 2.9. Note that division by a power of 2 inside the floor function translates to a right bit-shift, which can be easily performed in a computer.

\[
r = t - M \left\lfloor \frac{t\mu}{2^q} \right\rfloor \quad \text{(2.9)}
\]

After using Equation 2.9, $r$ may still be larger than $M$. If this is the case, Equation 2.10 must be executed once [49][15][50].

\[
r = r - M \quad \text{(2.10)}
\]

2.3.2 Montgomery Reduction

Montgomery reduction [51] has a structure similar to the FFT multiplication algorithm. Before an input can be reduced, it must be transformed into the Montgomery domain. This transformation can be seen on lines 3 and 4 in the algorithm description shown in Algorithm 1. In line 14, the result is transformed out of the Montgomery domain. Note that Algorithm 1 presents the algorithm as a modular product. Montgomery reduction is traditionally presented in this manner in the literature, but it can be easily converted into only a modular reduction algorithm by replacing inputs
a and b with a single input to be reduced.

Algorithm 1: Montgomery Reduction Algorithm [52].

Input : $a, b$
Output: $c$ equal to $(ab \mod n)$

1. Choose $r \in \mathbb{N}$ such that $r > n$ and $\gcd(r, n) = 1$
2. Pre-compute $k$ such that $k = \frac{r(r^{-1} \mod n) - 1}{n}$
3. $\bar{a} = (ar \mod n)$
4. $\bar{b} = (br \mod n)$
5. $x = \bar{a} \bar{b}$
6. $s = (xk \mod r)$
7. $t = x + sn$
8. $u = \frac{t}{r}$
9. if $u < n$ then
10. \hspace{1em} $\bar{c} = u$
11. else
12. \hspace{1em} $\bar{c} = u - n$
13. end
14. $c = (\bar{cr}^{-1} \mod n)$
Chapter 3

High-Level Synthesis of Karatsuba Multiplication

The algorithm which was chosen to be a target for High-Level Synthesis was Karatsuba multiplication. This algorithm is used by multiple software libraries, including FLINT and GMP, for multiplication of polynomials and integers which are too large to be efficiently computed with the schoolbook multiplication algorithm but are not large enough to justify the overhead associated with algorithms based off of the FFT.

This algorithm was chosen based on two of its features. First, the Karatsuba algorithm can be highly parallelized and therefore should map well to the highly parallel structure of an FPGA. Second, the inherent recursive nature of the Karatsuba algorithm brings some interesting complications to the process of High-Level Synthesis, and solutions to these complications can be applied to other recursive algorithms used in homomorphic encryption.

After an in-depth introduction to the Karatsuba algorithm, this chapter will describe the development of a C-language software implementation of the Karatsuba algorithm for HLS. Each version of the implementation will be described along with its synthesis results. Lessons learned from each set of synthesis results are applied to subsequent versions of the algorithm until a satisfactory performance is acquired.
3.1 Karatsuba Multiplication

The notation given in the Karatsuba example in Chapter 2 will be used extensively throughout this chapter. In addition, in order to make the following discussion easier to follow, the products $A_H B_H$, $A_L B_L$, and $[(A_H + A_L)(B_H + B_L) - A_L B_L - A_H B_H]$ will be called the “high product,” “low product,” and “middle product,” respectively.

For operands with a degree greater than one, the Karatsuba algorithm is executed recursively. Each product is broken up into its high, middle, and low products recursively until the operands of the products are constants. This process creates a ternary tree as shown in Figure 3.1. In the figure, the low products branch to the right, and the high products branch to the left. Notice that all the operations shown in the leaf nodes have integer (not polynomial) operands. Once the value of these leaf nodes is calculated, the algorithm works up the tree combining these values together to build the polynomial products.

In order for the algorithm to execute recursively, each operand must be able to be split into two halves of equal size until each half has only one term. This means that
n (the number of terms in an operand) must be a power of 2. If the number of terms in an operand is not a power of 2, zeros must be inserted as coefficients for higher degree x’s until the length of the operand is a power of 2.

3.2 Recursion and High-Level Synthesis

The Karatsuba algorithm is naturally recursive. The software function used in the FLINT library (_fmpz_poly_mul_karatsuba [33]) starts at the root of the Karatsuba tree and starts iterating down the “low” branch (\(A_LB_L\)). Once it reaches a leaf node, it computes the product of the associated coefficients and then returns. The recursive function call on the “low” branch is followed by a recursive function call on the “high” branch (\(A_HB_H\)). Once this call returns, the function finally moves down the “middle” branch ((\(A_H + A_L\))(\(B_H + B_L\))). This process is illustrated by the step numbers shown next to each leaf node on the Karatsuba tree shown in Figure 3.2. The “#” labels under each leaf node denote the order in which each leaf product is evaluated.

Although recursion brings a level of simplicity to software algorithms, it causes
the process of preparing code for High-Level Synthesis more complicated than it is for non-recursive implementations. Currently available High-Level Synthesis tools do not support recursive algorithms. Theoretically, the High-Level Synthesis tool should be able to convert the Karatsuba algorithm to hardware if the size of the Karatsuba tree (i.e. the number of recursive calls) were set to a constant value, but currently available tools do not support this type of configuration. Given these constraints, the Karatsuba function needed to be implemented in a non-recursive manner.

The process of converting a recursive function into a non-recursive function is straightforward in software when high-level data structures, such as stacks, are readily available, but this process is more complex when the software is being prepared for High-Level Synthesis. The frequent use of high-level memory structures may not be the fastest or most resource efficient solution for a hardware accelerator. In [53], a stack data structure is explicitly created and maintained in the software description in order to mimic the use of a stack in recursion. This work takes a different approach. The overhead associated with maintaining a stack data structure is removed by creating a custom data storage scheme which takes advantage of optimizations enabled by the unique qualities of the Karatsuba algorithm.

3.3 Leveraging the Karatsuba Tree Structure

3.3.1 Representing the Tree

The high-level view of the Karatsuba algorithm given by the tree in Figure 3.1 shows three parallel branches. This representation emphasizes the fact that the calculations required for the leaf nodes can be performed in parallel. In this work, a C-language version of the Karatsuba algorithm was designed to model this parallelism. The size of the Karatsuba tree, and thus the size of the operands, would be static. The tree structure would be represented in hardware with as much parallelism as possible.
CHAPTER 3. HIGH-LEVEL SYNTHESIS OF KARATSUBA MULTIPLICATION

Note that this design could be extended to support very large Karatsuba trees. Even if the available hardware does not have enough resources to accommodate the calculations for the full Karatsuba tree, the hardware could be used iteratively to calculate sub-trees. The algorithm running in software would then be responsible for performing the remaining product combination steps close to the root of the tree. For example, if this was done for the tree shown in Figure 3.1, the high, middle, and low products of the root node could be computed in separate calls to the hardware accelerator, and the algorithm running on the CPU could combine these three products to get the final result. Obviously, this would likely not be necessary for the small tree shown in the figure, but it could become important for trees containing hundreds of coefficients.

3.3.2 Calculating the Leaf Product Operands

With the size of the tree statically set, all of leaf node product operands can be pre-calculated. For example, before any actual numbers are given, it is known that the leftmost leaf node in the tree shown in Figure 3.1 contains the product of $a_3$ and $b_3$, and the next leaf node contains the product of the sum ($a_3 + a_2$) and the sum ($b_3 + b_2$). The process of writing clean, non-recursive code which could correctly index all the coefficients for these operands based on a parameterizable tree size proved to be difficult. In order to try to bypass this problem, a set of recursive and non-recursive MATLAB functions were written which generated the C code based on a given operand size parameter. The result was essentially an implementation of Karatsuba with all the recursion “unrolled.” This solution, however, was not ideal. Although High-Level Synthesis requires that loops be carefully constructed, the presence of loops in the code gives the designer a degree of control over the resource use of the resulting hardware accelerator generated by High-Level Synthesis. Code which starts in a fully unrolled state cannot be adjusted to use fewer resources.
such as it would in a rolled or partially unrolled state. A more direct solution to the coding problem was needed.

The indexing problem was solved by breaking up the Karatsuba algorithm into two pieces: a recursive piece and a non-recursive piece. The recursive piece, \textit{karatsuba\_calc\_leaves}, is executed in software and is tasked with calculating the operands required for the leaf node products. The non-recursive piece, \textit{karatsuba\_combine}, takes the operands from the recursive piece, calculates the leaf products (i.e. the value of $a_3b_3$, $(a_3 + a_2)(b_3 + b_2)$, etc.), and works up the Karatsuba tree combining products until it reaches the root node and calculates the final product. This non-recursive piece can be given to an HLS tool and converted into a hardware accelerator.

In the original design, the \textit{karatsuba\_calc\_leaves} function maps the coefficients and performs any necessary addition operations. For example, the sum $a_3 + a_2$ in the second leaf is calculated by \textit{karatsuba\_calc\_leaves} and passed to \textit{karatsuba\_combine} as a single value. In a future version, \textit{karatsuba\_calc\_leaves} could be changed to simply generate a coefficient mapping. This mapping could then be passed to the hardware accelerator, and the accelerator could perform all operations required for a Karatsuba multiplication. In this configuration, \textit{karatsuba\_calc\_leaves} would only need to be executed upon setup of the system, not for every multiplication. Alternatively, the mapping could simply be generated by a script and linked to the \textit{karatsuba\_combine} source file. For the initial version, however, \textit{karatsuba\_calc\_leaves} is executed for every multiplication, and \textit{karatsuba\_combine} is given the more computationally expensive tasks of integer coefficient multiplication and product combination.

The \textit{karatsuba\_combine} algorithm went through several versions as it evolved to work better with the HLS tool. After a brief introduction to the concept of synthesis directives, the following sections will walk through the algorithmic structure of each version and the corresponding HLS results.
3.4 Vivado HLS Synthesis Directives and Data Types

The HLS tool used to synthesize the Karatsuba implementation was Vivado HLS. Vivado HLS has a catalog of synthesis directives which can be associated with sections of a software source code. These directives specify which optimization methods the tool should use on a given section of code. In this section, several of the most important directives will be introduced. For a complete listing of available directives, see [54].

3.4.1 UNROLL Directive

The UNROLL directive is self-explanatory. It is applied to loops, and it has a factor parameter which specifies the number of times the loop should be unrolled. A factor of 0 or 1 leaves the loop unchanged. If no factor is specified, the loop will be fully unrolled.

When applied to hardware synthesis, the UNROLL directive is used to specify how many instances of the loop logic should be present in the resulting circuit. This directive must be used carefully. If the unroll factor is set to even one level higher than required, a significant amount of unnecessary hardware could be generated.

3.4.2 PIPELINE Directive

The PIPELINE directive enables hardware-level pipelining similar to the technique implemented in the instruction pipeline of a CPU. When applied to loops, this directive allows consecutive loop iterations to be executed in parallel. A loop iteration does not need to wait for the previous iteration to fully complete before it begins execution. This is illustrated in Figure 3.3.

Figure 3.3 shows two examples of how a loop containing four consecutive operations (OP1, OP2, OP3, and OP4) could be pipelined. In the ideal case, the next
CHAPTER 3. HIGH-LEVEL SYNTHESIS OF KARATSUBA MULTIPLICATION

Figure 3.3: Example of pipelining.

loop iteration could begin one cycle after the previous iteration starts. This would mean that the hardware used to perform OP1 would never be idle. If, however, OP1 depends upon the output of OP2 from the previous iteration, it may need to wait two cycles before it can begin. The ideal case is said to have an Initiation Interval (II) of 1, while the other case is said to have an II of 2. The PIPELINE directive allows the desired II to be specified. If the desired II cannot be accomplished, the tool will increment the II until it finds one that works. By default, the II is set to 1.

When the PIPELINE directive is applied to a loop or function, all of the sub-loops will be fully unrolled. This is not ideal if the sub-loops are large. In these situations, the DATAFLOW directive should be considered. For a more detailed description of the available options, please see [54].

3.4.3 DATAFLOW Directive

The DATAFLOW directive is similar to the PIPELINE directive except it works at the level of “dataflow processes” instead of clock cycles. The level of loop or function hierarchy at which the DATAFLOW directive is placed is designated as a “dataflow region.” Inside a dataflow region, consecutive loops and function calls which access the same memory structures are grouped together into “dataflow processes.” These dataflow processes are linked together by FIFO channels. All the dataflow
processes operate independently of one another. Each FIFO channel queues the output data from one dataflow process and feeds it to the next dataflow process when the next process is ready for input. This allows all the dataflow processes to execute simultaneously.

When the DATAFLOW directive is not used, loop logic will become idle in the hardware circuit when it has finished its task and is waiting for the loops that follow to finish. In this situation, the execution time of the full function is equal to the sum of the execution times of each loop processor. When the DATAFLOW directive is used in a streaming data situation, however, the execution time of the function becomes equal to the execution time of the slowest loop processor.

### 3.4.4 ARRAY_PARTITION Directive

The ARRAY_PARTITION directive is used to divide arrays into multiple, smaller arrays. As will be shown in this chapter, this is useful when parallelism is needed.

There are three types of array partitioning available: complete, block, and cyclic. Complete partitioning gives every array element its own memory structure. In hardware, this means that instead of storing the array in a single memory structure, such as a block RAM, each element is stored in a flip-flop. Although this makes all the
data readily accessible, it is very inefficient when the array is large.

Block partitioning is illustrated in Figure 3.4. In the figure, the original array
has four elements. If a block partition with a factor of 2 is applied to this array, the
array is essentially cut in half. This can be useful if two processes which need to
execute in parallel need access to different halves of the array. If the array were not
partitioned, the processes would not be able to operate in parallel due to the conflict
caused by the data dependency. Note that, in hardware, the two smaller arrays will
be implemented as two separate memory structures.

Cyclic partitioning is useful if an operation requires several consecutive array
elements. If a cyclic partition with a factor of 2 is applied to an array, two new
arrays are created which contain every other element from the original array. This is
illustrated at the bottom of Figure 3.4. Cyclic partitioning is useful if, for example,
the array contained pairs of values which needed to be multiplied by each other.
Instead of waiting for two values to be loaded consecutively, the two operands could
be loaded in parallel from different memory structures. Again, in hardware, the two
smaller arrays will be implemented as two separate memory structures.

3.4.5 CLOCK Directive

The CLOCK directive can be used to set the target clock speed for the hardware
accelerator generated by Vivado HLS. By default, the clock period is set to 10 ns
(100 MHz). The synthesis report for a generated accelerator will give the estimated
shortest clock period which can be used with the accelerator. Since this is an estimate,
the accelerator must be generated with the clock period adjusted to this estimated
period in order to confirm that it can be attained.
3.4.6 DEPENDENCE Directive

The DEPENDENCE directive is used to specify the existence or non-existence of a data dependency. Vivado HLS will sometimes miss existing data dependencies or flag dependencies that do not actually exist. For example, if the tool incorrectly flags a dependency in a pipelined loop, it could result in a longer interval than is necessary. This directive can be used to minimize pipeline intervals in these situations.

3.4.7 Arbitrary Precision Data Types

When Vivado HLS evaluates C-language functions which use standard C data types, it assumes that all of the bits in the defined data type are being used. For example, if a variable of type uint32_t is used, all the hardware that Vivado HLS generates to evaluate the variable will have a width of 32 bits. This can be inefficient if it is known that the variable’s value never exceeds 10 bits, for example.

Vivado HLS addresses this problem by providing support for their own arbitrary precision data types. This support allows variables to be declared as type “intn” or “uintn” where n is any number of bits. These data types were used in this work to allow the coefficient sizes used in the Karatsuba hardware accelerator to be 64, 128, and 192 bits.

3.5 Version 1: A Single Memory Structure

3.5.1 Version 1: File Structure, Control Constants, and Verification

The Karatsuba software implementation was originally intended to be written as a direct extension of the FLINT library, but because of the use of recursion and the numerous files required to use the library’s integer and polynomial objects (which would not work directly with HLS without some editing), the software implementation was constructed as a stand-alone C-language algorithm. This proof-of-concept, stand-
alone implementation can be easily modified to use integer data types of various classes and sizes by editing some `#define`'s.

The implementation is made up of three source files and one header file. The source file `karatsuba_calc_leaves.c` contains the recursive `karatsuba_calc_leaves` function, while `karatsuba_combine.c` contains the non-recursive function (`karatsuba_combine`) which will be the focus of the rest of this chapter. The header file `karatsuba_mul.h` contains the definitions for the data type of the coefficients (`coeff_t`), the data type for counter variables (`index_t`), the operand size, and constant parameters related to the operand size.

A constant flag `HLS_EN` is defined in the file containing the `karatsuba_combine` function. This flag enables special memory declarations and loop labels which are required for High-Level Synthesis. When `HLS_EN` is zero, `malloc` is used to allocate memory. When `HLS_EN` is non-zero, arrays are only declared. The loop labels included for High-Level Synthesis are used by Vivado HLS to identify the loop to which a synthesis directive should be applied.

The source file `test_karatsuba.c` contains a main() function which will run the `karatsuba_mul` function against test cases. The test cases are generated by a MATLAB script (`gen_poly_mult_test_data.m`) which uses MATLAB’s pseudo-random number generator and convolution functions to generate pseudo-random sets of operands and calculate the expected products. These test cases are saved to header files and referenced via `#include` statements in `test_karatsuba.c`. `test_karatsuba.c` has also been configured to work as a test bench for Vivado HLS’s C simulation and C/RTL Cosimulation. The accelerators presented in this work were verified using `test_karatsuba.c` with C/RTL Cosimulation. Finally, `test_karatsuba.c` can also be configured to calculate the execution time for a full set of test cases or for each test case individually.
3.5.2 Version 1: Algorithm

Before any parts of the structure of the karatsuba_combine algorithm were written, the Karatsuba algorithm was carefully parsed in order to try to find patterns or other qualities which could be used to optimize the software implementation. One distinct pattern related to the shifting portion of Karatsuba became a major cornerstone of the resulting implementation.

In the Karatsuba algorithm, once the low, middle, and high products are calculated, the middle and high products are multiplied by powers of \( x \) (i.e. “shifted”) before all the products are summed. It was found that the amount that the high product is shifted causes it to never “overlap” with the low product when they are summed. In other words, none of the terms in the high product need to be summed with terms in the low product in the final step since the low product will never have a coefficient for \( x^n \). In fact, the highest possible power of \( x \) present in the low product is \( x^{n-2} \). This means that there is always a one-coefficient (power of \( x \)) gap between where the low product ends and the high product begins. The amount of overlap between the middle product and the low product on one side and the high product on the other side can also be easily calculated based upon the length of the products.

Figure 3.5 gives an example in which \( n \) from Equation 2.6 is 4. In the example, the middle product is shifted by 2 (multiplied by \( x^2 \)), and the high product is shifted by 4 (multiplied by \( x^4 \)). These patterns were used to maximize potential parallelism in memory accesses in the software implementation.

Since dynamic memory allocation is not supported by High-Level Synthesis tools, all of the required memory must be declared with a constant size at the beginning of each function. Any function which dynamically allocates sections of memory as needed must be revised in order to support indexing over a limited number of large arrays. In the case of the Karatsuba algorithm, the length of the arrays required to hold products varies as the tree is traversed. The first version of karatsuba_combine
CHAPTER 3. HIGH-LEVEL SYNTHESIS OF KARATSUBA MULTIPLICATION

Figure 3.5: Combining low, middle, and high products with 3 terms into a product with 7 terms.

Figure 3.6: Storage of Karatsuba “leaf products” in a C-language array “mem.”

sought to use a single array of the minimum size required for the operation instead of multiple arrays which are each only used for a single stage of the operation. This was done in order to allow the algorithm to be an efficient software solution while also being compatible with HLS. Figure 3.6 gives an example of the product storage strategy used in karatsuba_combine. The products shown are taken from the example Karatsuba tree shown in Figure 3.1.

As shown in Figure 3.6, the Karatsuba tree leaf products are stored in a one-dimensional array called “mem.” The products are stored in the following order: low, middle, high. This ordering changes for the branches: The branch ordering is low, high, middle. Since the loop which combined the intermediate products added in the middle product last, this ordering simplified the array indexing logic. Note that each
Figure 3.7: The combination step used by the Karatsuba function `karatsuba_combine`. The labels under the array indicate the term of the final product for which the array element is a coefficient (e.g. $c$ is the constant term, $x$ is the coefficient of the $x$ term, etc.).

leaf product will be a single value of a pre-set coefficient data type of constant size.

The `mem` array is used as a “scratchpad” for the full Karatsuba calculation and is the only array required for the calculation. `mem` is double the size required to hold the leaf products. For example, the array in Figure 3.6 holds 18 elements. After the leaf products have been calculated and the algorithm starts working up the Karatsuba tree, one side of the `mem` array is used to store the results from the previous iteration, and the other side is used to write the results for the current iteration. This process enables the algorithm to use only one array in the calculation.

In the first combination step, the leaf products are made to represent the coefficients of the second-degree products. For example, $a_0b_0$, $a_1b_1$, and $(a_1 + a_0)(b_1 + b_0)$ from Figure 3.1 become the result of the product $(a_1x + a_0)(b_1x + b_0)$. Because of the “shifting” (multiplying by powers of $x$) described by the Karatsuba equation, this can be done easily. For example, $a_0b_0$ is already equal to the constant term of the product, and $a_1b_1$ is already equal to the coefficient of $x^2$ in the product. The only operations which need to be done are to subtract $a_0b_0$ and $a_1b_1$ from the product $(a_1 + a_0)(b_1 + b_0)$ in order to get the middle product, the coefficient of $x$. This can be done directly and does not require the use of the other side of the `mem` array.

After the first combination step, the second-degree products are combined to form the sixth-degree final product as shown in Figure 3.7. The coefficients of the low,
middle, and high second-degree products are represented in the figure by the letters \( l \), \( m \), and \( h \), respectively. For example, \( l_0 \) is the constant term of the low second-degree product. The Karatsuba equation shows that the middle second-degree product must be multiplied by \( x^2 \), and the high second degree product must be multiplied by \( x^4 \). As was mentioned earlier, notice that this “shifting” allows the product to be constructed with only two additions. While these addition operations are occurring, all the other values can be written in parallel directly to the storage location of the new product. The number of additions will double on each successive combination step.

Please note that the middle term in Figure 3.7 is not taken directly from the second-degree middle product. Before the step shown in the figure, the low and high second-degree products must be subtracted from the original middle second-degree product. The middle product is equal to \((A_H + A_L)(B_H + B_L) - A_LB_L - A_HB_H\) not just \((A_H + A_L)(B_H + B_L)\).

For larger Karatsuba trees, the operation shown in Figure 3.7 is performed continuously until the final product has been constructed. Each half of \( \text{mem} \) changes from old product storage to new product construction target every iteration. In the code, consecutive memory locations holding the low, high, and middle products used to build the same larger product are referred to as “sections.” The size of each section starts at 3 and triples in size as the algorithm moves up each level of the tree (In the code, “level” is used to refer to the vertical dimension. For example, the height of the tree could be expressed by a number of levels.). One downside of this method is the expanding sections of unused memory. The overlap of the second-degree products shown in Figure 3.7 shows that as products are combined to build larger products, the memory space required to store the smaller products is larger than the space required to store the larger products. This unused space is illustrated in Figure 3.7 by memory locations 16 and 17. Since there is no equivalent to memory deallocation in digital circuits, the full \( \text{mem} \) array must remain allocated for the full execution of
CHAPTER 3. HIGH-LEVEL SYNTHESIS OF KARATSUBA MULTIPLICATION

the function in order for the function to be accepted by a High-Level Synthesis tool.

3.5.3 Version 1: Data Flow

As the karatsuba_combine algorithm evolved, it became important to think about the algorithm in terms similar to those used by the HLS tool. As illustrated in Figure 3.8, Vivado HLS treats loops like stand-alone data processors. It reports the latency and interval of each loop individually. In its default configuration, the total latency and interval of the function is equal to the sum of the latencies and intervals of the loops the function contains. This type of treatment can be aptly described using data flow diagrams similar to those used for high-level RTL diagrams. Figure 3.9 shows the data flow diagram for version 1 of karatsuba_combine. The pseudocode for this version is given in Algorithm 2.

Algorithm 2: Pseudocode for Version 1 of karatsuba_combine.

1. leaf_products : foreach leaf of the Karatsuba tree do
   2. Calculate leaf product.
   3. Subtract $A_L B_L$ and $A_H B_H$ from the leaf product $(A_L + A_H)(B_L + B_H)$ to get middle product.
4. end
5. level : foreach level of the Karatsuba tree do
6. section : foreach intermediate product in a level do
7.   low : foreach coefficient in the low product do
8.     Write the low product into the next intermediate product’s memory location.
9.   end
10. high : foreach coefficient in the high product do
11.     Write the high product into the next intermediate product’s memory location.
12. end
13. middle : foreach coefficient in the middle product do
14.     Add the middle product (with $A_L B_L$ and $A_H B_H$ subtractions) into the next intermediate product’s memory location.
15. end
16. end
17. end
In version 1, the leaf_products loop iterates over all the leaf product operands calculating the leaf products and applying the subtractions required for the middle products. During the multiplication step, the integer values are multiplied and stored in \textit{mem}. For the subtraction step, the products are read out of \textit{mem}, the subtraction operation is performed, and the result is written back into \textit{mem}.

The level loop iterates through each level of the tree from the bottom up. The section loop, which is nested inside the level loop, iterates over each set of low, high, and middle products in a level and combines them to form the next intermediate product. The low, high, and middle products are written to \textit{mem} using smaller loops nested inside the section loop. The data flow for these is shown in Figure 3.10. Note that all of the intermediate product data is read from and written to \textit{mem}. When the level loop is on its final iteration, the final product is written to the output array, \textit{product}.

### 3.5.4 Version 1: High-Level Synthesis Results

Table 3.1 compares the execution time of the \textit{karatsuba_combine} algorithm running on an AMD CPU and an FPGA running at different clock speeds. In the case of
Figure 3.9: Data flow diagram for version 1 of karatsuba_combine.

Table 3.1: Execution time comparison between an AMD CPU and the HLS result for karatsuba_combine version 1 with no directives. CPU specifications: 3.7 GHz clock speed, 14.5 GB RAM, Red Hat Enterprise Linux (Release 6.8) OS. FPGA model: xcvu9p-fsgd2104-3-e-es1.
Figure 3.10: Data flow diagram for the section loop in version 1 of karatsuba\_combine. This is a sub-diagram of Figure 3.9.
the FPGA running at the default clock speed (100 MHz), the accelerator running on the FPGA was generated by Vivado HLS without the use of any directives. The data plainly shows that this accelerator is significantly slower than the algorithm running on the CPU. Even if a CLOCK directive is added and the FPGA clock speed is brought close to the maximum possible clock speed for the design (around 350 MHz), the accelerator is still significantly slower. The benefit of hardware acceleration through HLS cannot be gained by simply dropping the software into the HLS tool and synthesizing it. A hardware-centered coding style must be combined with a wise use of synthesis directives in order to generate an accelerator which makes use of the strengths of an FPGA and thereby achieves a positive speedup over a CPU.

When \texttt{karatsuba\_combine} was first synthesized with no directives, the Analysis view in Vivado HLS showed that the divider opcode \texttt{udiv} was being called to calculate the number of sections (i.e. products to be built) in each level. This was a problem since the divider had a very high latency. The number of sections was set equal to the number of leaf products at the bottom level and was subsequently divided by 3 as the algorithm moved up each level. The number of sections for each level could therefore be precalculated since it is initially based off of a constant and was iteratively divided by a constant. These kinds of patterns were not automatically detected by Vivado HLS. The realization of this proved to be an introduction to the level of responsibility that Vivado HLS leaves to the user. It does perform some hardware optimizations automatically, but when it comes to interpreting the software algorithm, it is very careful not to overrule any decisions that the programmer has made. In this case, a division symbol was used; therefore, a divider was instantiated with no questions asked.

An UNROLL directive was used to unroll the level loop completely. The number of levels in a Karatsuba tree is very small compared to the size of the operands; therefore, this application of the UNROLL directive was scalable. For example, if
each operand has 512 coefficients, the level loop only has 8 iterations \((\log_2(512) - 1)\). This unrolling was done for all subsequent versions of \textit{karatsuba\_combine}. This action not only removed the divider, but it also allowed for more efficient pipelining in the future. In version 1, this action decreased the overall latency by about 160 cycles. This improved latency, however, came at the expense of about 400 flip-flops and 1300 LUTs. Unrolling removed the need for the divider, but it also duplicated the level loop hardware. Since PIPELINE directives had not yet been added, the duplicated hardware was forced to share the same memory structures. Since the duplicated hardware was forced to execute sequentially, the latency benefit was small compared to the increase in resource usage. This was remedied in later versions.

The loop with the highest latency in version 1 was the leaf\_products loop. This loop, therefore, became the target for experiments with the UNROLL, PIPELINE, and ARRAY\_PARTITION directives. As was just mentioned, unrolling a loop which uses memory structures (arrays) without increasing the memory bandwidth (number of partitions) increases the hardware usage without improving the latency. Unrolling the loop too much can even increase the latency due to increased hardware complexity. A good indicator that the maximum unroll factor has been passed is an increase in reported loop iteration latency which balances out the decrease in loop trip count caused by unrolling.

The array partition type selection became an important part of the initial experiments. The “complete” partition type usually had the best performance, but it did not scale well to larger operands due to its high resource demands. The “block” partition type did not help the latency of the loops since the data was accessed sequentially. The loop unroll directive allows consecutive iterations of the loop to be executed in parallel; therefore, in order to the make unrolling worthwhile, consecutive data values must be available in parallel. This can be done with the “cyclic” partition type.
One thing to watch when experimenting with the ARRAY_PARTITION directive is the number of memory ports being used in the synthesized circuit. By default, arrays used internal to the top-level function or as inputs and outputs are synthesized to dual-port block RAMs. Depending on the partition factor and the usage of the memory, Vivado HLS will decide whether or not to use both ports on each block RAM. If the array is used for inputs or outputs (like leaf_ops in karatsuba_combine), the number of ports used is shown in the Interface Summary section of the synthesis report. In some cases, if only one port is being used, the partition factor (and thereby the number of block RAMs) can be cut in half without any resulting performance loss.

Vivado HLS will sometimes give a warning when an array partition factor is not high enough to fulfill the needs of an unrolled or pipelined loop. This synthesis warning will state that a “store” or “load” operation could not be scheduled “due to limited memory ports.” It is usually best to slowly increase the partition factor until this warning is no longer generated.

In the case of the leaf products loop, each loop iteration requires two consecutive values from the input, leaf_ops. The best partition type, therefore, was cyclic. In order to make unrolling the leaf products loop worthwhile, the leaf_ops input array had to be partitioned at a factor at least equal to the unrolling factor. If the partition factor is too large, the resource usage will spike. If it is too small, some of the duplicated hardware will have to wait for input values to become available. Eventually, the array to which the resulting products were being written (mem) became the bottleneck.

Table 3.2 shows some examples of how the resource usage changed for version 1 with the adjustment of the leaf products loop unroll factor and the leaf_ops cyclic array partition factor. Note that when the partition factor changes from 8 to 16, the LUT usage drops and the DSP48E usage increases. When the partition factor was 8, 3 of the 8 multipliers need for the 8 parallel instances of the leaf products loop were
Table 3.2: Resource usage for a few sets of synthesis directives applied to version 1 of karatsuba_combine. FPGA model: xcvu9p-fsgd2104-3-e-es1.

Built using LUTs. When the partition factor was increased, the HLS tool was able to route all the inputs to DSPs causing the LUT usage to drop. When the partition factor was raised to 128, the LUT usage spiked due to the need to route such a large input array to a much smaller set of multipliers. In the end, all of these configurations had exactly the same estimated execution time (2.061090 ms). Note that these results are for operands with 512 32-bit coefficients with the level loop fully unrolled, the leaf_products loop pipelined, and a clock frequency of 100 MHz.

One notable discovery during experimentation with the ARRAY_PARTITION directive regarded the use of the opcode urem by the Vivado HLS scheduler. Like the udiv opcode, this operation, which calculates the remainder for a division, has a very long latency which can drastically affect the overall latency of a loop. With the help of some experimental results and a Xilinx forums post [55], it was found that this opcode is often used when the array partition factor is not a power of 2. Although this is a good rule of thumb, there were a few cases in later versions when urem was not used when the partition factor was a multiple of 3. This may have been due to the fact that this particular algorithm works in groups of 3’s (low, middle, and high products). These cases were rare, however, and there was never a case when urem was used with a power of 2 partition factor.

In the end, the most useful directive to be applied to the leaf products loop was PIPELINE. Before a loop is pipelined, its total latency is equal to the loop’s iteration latency multiplied by its trip count. Unrolling the loop can bring the trip count
Table 3.3: Estimated execution time for a few sets of synthesis directives applied to version 1 of \textit{karatsuba\_combine}. FPGA model: xcvu9p-fsgd2104-3-e-es1.

down by the unrolling factor, but, as was discussed earlier, if unrolling is not done in combination with the \textsc{ARRAY\_PARTITION} directive, it can cause the loop’s iteration latency to increase and thereby have no effect or a negative effect on the loop’s total latency. When a loop is pipelined, the loop’s total latency is approximately equal to its interval ($II$) multiplied by its trip count plus its latency. Since the leaf products loop iterations were relatively independent, pipelining this loop cut the total loop latency by a factor of up to 6. Despite this drop in latency, version 1 was still not competitive with its corresponding software implementation. This is shown in Table 3.3 for operands with 512 32-bit coefficients at the default 100 MHz clock frequency.

In all cases, the level loop is fully unrolled, and the leaf products loop is pipelined.

### 3.6 Version 2: Separate Leaf Multiplication and Subtraction

Version 2 was created in order to improve the interval ($II$) of the leaf products loop. This was done by removing a modulus operation and a memory bottleneck.

#### 3.6.1 Version 2: Data Flow

After the initial multiplication for the leaf products, the subtractions need to be applied to the middle product. This subtraction operation is dependent on every consecutive set of three leaf products. In version 1, this operation was done every 3 iterations of the leaf products loop using an if statement and a modulus operator.
CHAPTER 3. HIGH-LEVEL SYNTHESIS OF KARATSUBA MULTIPLICATION

The subtraction operation would be performed every time the loop iterator mod 3 was equal to 2 (every third iteration). It was thought that Vivado HLS would detect that this operation was performed every third cycle and factor this into its state machine design. In the end, this proved to be true when the loop was left rolled or when its unroll factor was a multiple of 3. If, however, the unroll factor was not a multiple of 3, Vivado HLS interpreted the modulus operator literally and calculated the state of the if statement condition using the high latency opcode urem. This constraint made it difficult to match an unroll factor to the leaf_ops array partition factor, which was constrained to a power of 2.

In addition to the urem opcode issue, the leaf products loop in version 1 also had an unintended memory bottleneck. The middle product subtraction operation can be performed independent of the leaf product multiplications, but its presence in the leaf products loop was preventing the interval of the loop from getting better than 2. According to Vivado HLS’s Analysis view, the memory reads required for the subtractions were interfering with the memory write of the product.

In version 2, the middle leaf product subtraction step was pulled out of the leaf products loop and put in its own loop, the “mid_sub” loop. This can be seen in the updated pseudocode and data flow diagram shown Algorithm 3 and Figure 3.11, respectively.

3.6.2 Version 2: High-Level Synthesis

Given a sufficient array partition factor for both leaf_ops and mem, both the leaf products and the mid_sub (middle leaf product subtractions) loops could be pipelined with an interval of 1. This pipelining was combined with cyclic array partitioning of the internal array mem. As shown in Table 3.4, this resulted in a significant increase in resource usage. This jump in usage, however, was not in vain. As shown in Table 3.5, these new accelerators were able to achieve a speedup of around 2 compared to

1 leaf_products : foreach leaf of the Karatsuba tree do
2     Calculate leaf product.
3 end
4 mid_sub : foreach group of a low, middle, and high leaf product do
5     Subtract $A_L B_L$ and $A_H B_H$ from the leaf product $(A_L + A_H)(B_L + B_H)$ to get the middle product.
6 end
7 level : foreach level of the Karatsuba tree do
8     section : foreach intermediate product in a level do
9         low : foreach coefficient in the low product do
10            Write the low product into the next intermediate product’s memory location.
11         end
12     end
13     high : foreach coefficient in the high product do
14         Write the high product into the next intermediate product’s memory location.
15     end
16     middle : foreach coefficient in the middle product do
17         Add the middle product (with $A_L B_L$ and $A_H B_H$ subtractions) into the next intermediate product’s memory location.
18     end
19 end
Figure 3.11: Data flow diagram for version 2 of karatsuba_combine.
TABLE 3.4: Resource usage for a few sets of synthesis directives applied to version 2 of karatsuba combine. FPGA model: xcvu9p-fsgd2104-3-e-es1.

<table>
<thead>
<tr>
<th>Leaf Products</th>
<th>leaf_ops</th>
<th>Cyclic Partition Factor</th>
<th>Resource Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6</td>
<td>75</td>
<td>128,410, 154,560</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>75</td>
<td>143,115, 169,283</td>
</tr>
<tr>
<td>12</td>
<td>24</td>
<td>75</td>
<td>143,115, 169,283</td>
</tr>
</tbody>
</table>

Table 3.5: Estimated execution time for a few sets of synthesis directives applied to version 2 of karatsuba combine. FPGA model: xcvu9p-fsgd2104-3-e-es1.

<table>
<thead>
<tr>
<th>Leaf Products</th>
<th>leaf_ops</th>
<th>Cyclic Partition Factor</th>
<th>Estimated Execution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6</td>
<td></td>
<td>0.400930</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td></td>
<td>0.467340</td>
</tr>
<tr>
<td>12</td>
<td>24</td>
<td></td>
<td>0.467340</td>
</tr>
</tbody>
</table>

Version 2 Software Execution Time = 0.8479 ms

Their software counterparts running on a CPU. Note that the software being used as a benchmark is the software used to generate the hardware accelerator. The CPU specifications are the same as those used to evaluate version 1, and the synthesis directives used in version 1 were rolled over into version 2. The only exception is the use of cyclic partition with a factor of 3 for the mem array.

This increase in performance was a step in the right direction, but the jump in usage was not ideal. In the end, the sum of all the loop latencies was not going to be able to be brought low enough to get the desired performance. The full function needed to be a pipeline. In other words, a loop should be able to start on the next set of inputs immediately without waiting for later loops to finish processing its most recent output. When this is done, the interval and latency of the function would be different. The interval would be the actual execution time (i.e. the number of cycles between each new output). This interval would be equal to the largest interval of all the consecutive loops in the function (i.e. the slowest pipeline stage).

Vivado HLS allows the pipeline directive to be applied to the top-level function.
When this was done, the tool was able to generate highly optimized accelerator with an interval of 1 cycle when the operand size was 8 32-bit coefficients. This strategy, however, is not scalable. The pipeline directive fully unrolls every loop underneath the level of hierarchy at which it is set. This means that when a function is pipelined, every loop in the function is fully unrolled. This would not scale well to operands of 512 32-bit coefficients.

3.7 Version 3: Independent Final Product Assignment

Version 3 was created to remove some conditional memory write operations and to start giving the algorithm a more pipelined structure.

3.7.1 Version 3: Data Flow

In version 3, the memory writes to the final product array were removed from the section loop and placed in their own loop at the end of the karatsuba_combine function. This was done for two reasons. First, the removal of conditional memory writes allowed the inside of the section loop to be a dataflow region. Second, this was done to try to remove any of the latency incurred by the write condition checks and product memory writes from the section loop. These changes are reflected in the updated pseudocode and data flow diagram shown in Algorithm 4 and Figure 3.12, respectively.

3.7.2 Version 3: High-Level Synthesis

The changes made in version 3 did not have any noticeable impact on the HLS results. The removal of conditional writes from the section loop did not affect the loop’s latency. In the end, the memory bottleneck which was limiting the improvement of performance was caused by the way in which the mem array was being used.
Algorithm 4: Pseudocode for Version 3 of \texttt{karatsuba\_combine}.

1. leaf\_products : \textbf{foreach} leaf of the Karatsuba tree \textbf{do}
2. \hspace{1em} Calculate leaf product.
3. \textbf{end}

4. mid\_sub : \textbf{foreach} group of a low, middle, and high leaf product \textbf{do}
5. \hspace{1em} Subtract $A_L B_L$ and $A_H B_H$ from the leaf product $(A_L + A_H)(B_L + B_H)$ to get the middle product.
6. \textbf{end}

7. level : \textbf{foreach} level of the Karatsuba tree \textbf{do}
8. \hspace{1em} \textbf{section} : \textbf{foreach} intermediate product in a level \textbf{do}
9. \hspace{2em} \textbf{low} : \textbf{foreach} coefficient in the low product \textbf{do}
10. \hspace{3em} Write the low product into the next intermediate product’s memory location.
11. \hspace{2em} \textbf{end}
12. \hspace{2em} \textbf{high} : \textbf{foreach} coefficient in the high product \textbf{do}
13. \hspace{3em} Write the high product into the next intermediate product’s memory location.
14. \hspace{2em} \textbf{end}
15. \hspace{2em} \textbf{middle} : \textbf{foreach} coefficient in the middle product \textbf{do}
16. \hspace{3em} Add the middle product (with $A_L B_L$ and $A_H B_H$ subtractions) into the next intermediate product’s memory location.
17. \hspace{2em} \textbf{end}
18. \hspace{1em} \textbf{end}
19. \textbf{end}

20. product\_assign : \textbf{foreach} coefficient in the final product \textbf{do}
21. \hspace{1em} Copy the final product from the internal array to the \textit{product} output.
22. \textbf{end}
Figure 3.12: Data flow diagram for version 3 of \textit{karatsuba\_combine}.
From a digital design perspective, the maximum throughput could be achieved if a physically different hardware component is used to store the *mem* array at each stage in the algorithm. This would allow the function to be pipelined at the outer loops’ level of hierarchy. When a single array like *mem* is used in the software description, however, Vivado HLS uses the same set of memories to supply all the loops. This forces the first loop in the function to have to wait until the last loop in the function is complete before it can begin using the *mem* array again. This bottleneck could be removed by creating multiple instances of the *mem* array. From a software perspective, creating multiple instances of the same array can be a waste of memory. This is especially true when the memory cannot be de-allocated when it is no longer being used. In addition, the use of multiple arrays requires the use of multiple slow memory allocation operations. In the end, however, this had to be done in order to create an accelerator with an acceptable speedup over the fastest version of the software.

### 3.8 Version 4: Pipelined Internal Memory

The internal memory storage structure of the *karatsuba_combine* function was completely redesigned in version 4. The new structure was a closer imitation of an RTL design.

#### 3.8.1 Version 4: Algorithm

In version 4, the *mem* array is replaced with three separate arrays used to store the low, middle, and high products. Therefore, instead of storing each set of low, high, and middle products next to each other in a section of the *mem* array, these products are stored in the same position in their own array. This is illustrated in Figure 3.13. The products shown are taken directly from the Karatsuba tree example illustrated by Figure 3.1. If the Karatsuba tree is larger than the one shown in the figure, the next set of products for the low, high, and middle branches would be stored at array
Algorithm 5: Pseudocode for Version 4 of karatsuba_combine.

1. leaf_products: *foreach* leaf of the Karatsuba tree *do*
   2. Calculate leaf product.
   3. Select Stage 1 destination array and store leaf product.
   4. *end*

5. mid_sub: *foreach* group of a low, middle, and high leaf product *do*
   6. Subtract $A_L B_L$ and $A_H B_H$ from the leaf product $(A_L + A_H)(B_L + B_H)$ to get the middle product.
   7. Select Stage 2 destination array and store low, middle, and high products.
   8. *end*

9. level: *foreach* level of the Karatsuba tree *do*
   10. section_mid_sub: *foreach* group of a low, middle, and high product in the current level *do*
       11. Subtract $A_L B_L$ and $A_H B_H$ from the leaf product $(A_L + A_H)(B_L + B_H)$ to get middle product.
       12. Store low, middle, and high products in Stage 3 arrays.
       *end*

13. section: *foreach* intermediate product in a level *do*

14. section_build: *foreach* coefficient in the next intermediate product *do*
       15. if Writing lower half of intermediate product then
           16. Load low product coefficient.
       else if Writing upper half of intermediate product then
           17. Load high product coefficient.
       *end*

18. if Writing middle portion of intermediate product then
       19. Load middle product coefficient.
       *end*

20. if Writing middle portion of intermediate product then
       21. Sum loaded middle product with loaded low or high product and assign the result to temporary variable.
       *end*

22. if Building final product then
    23. Write temporary variable to output product array.
    else
       24. Select destination Stage 2 array.
       25. Write temporary variable to Stage 2 array.
       *end*

26. *end*

27. *end*
indices 3, 4, and 5 in the arrays low_products, high_products, and mid_products, respectively.

These three arrays were further broken up into three stages denoted in the source code by the prefix “s#.” where “#” is the stage number. For example, the first stage’s low products array is called $s1_{\text{low}_\text{products}}$. These stages were created to ensure that each section of the algorithm was dependent on a different memory structure thereby eliminating the bottleneck present in previous versions.

### 3.8.2 Version 4: Data Flow

Each stage of low, middle, and high product arrays was created to act as channels between loops. Stage 1 stored the initial leaf products and acted as the data source for the $\text{mid}_\text{sub}$ loop. Stage 2 stored the output of the $\text{mid}_\text{sub}$ loop and acted as the data source for the middle product subtraction operations required in the level loop. Stage 3 stored the results of the middle product subtractions and acted as the data
source for intermediate product combination step. Once inside the level loop, Stage 2 also stored the results from each level loop iteration and acted as the data source for the next iteration. The updated pseudocode and data flow diagram are shown in Algorithm 5 and Figure 3.14, respectively.

The memory read and write procedures in version 4 were carefully constructed in order to allow the arrays to be implemented as FIFOs either through the RESOURCE directive or a combination of the STREAM and DATAFLOW directives. In order for an array to be synthesized as a FIFO, Vivado HLS must be convinced that the array is accessed sequentially [54]. Because of this, all the loops in version 4 iterated sequentially over all the elements in each array even if not all the elements were being used in the current operation. In the section loop, each new section was constructed by a sub-loop, called the section_build loop in the source code, which sequentially multiplexed each value from the three input arrays into a single variable and de-multiplexed the value into the next stage of arrays. The data flow of this inner loop is shown in Figure 3.15.

3.8.3 Version 4: High-Level Synthesis

The accelerators generated from version 4 had significantly longer execution times than those generated from version 2. The best accelerator was only able to achieve an execution time of around 1.9 ms. This degradation in performance was caused by the sequential-access coding style described earlier. In version 4, the loops iterated through every element in each array in order to ensure that they would act like FIFOs. Since not all elements in a section are used in a product combination operation, this strategy of iterating through every element added some unnecessary memory accesses and loop iterations. In the end, these extra operations had a significant affect on the overall performance of the accelerator.

Although the performance did not improve, a step was taken toward obtaining
Figure 3.14: Data flow diagram for version 4 of karatsuba_combine.
Figure 3.15: Data flow diagram for the section loop in version 4 of karatsuba_combine. This is a sub-diagram of Figure 3.14.

A more pipelined overall function. Since the arrays were divided into stages, the DATAFLOW directive was used to successfully divide the function into two independent dataflow processes. The first process contained the leaf products loop, and the second process contained the mid_sub loop and the level loop. These two processes acted like two stages in a pipeline. The interval of the entire function was now equal to the interval of the slowest of the two dataflow processes. In this case, the process with the mid_sub and level loops was the slowest, and its execution time (about 1.9 ms) was equal to the execution time of the entire function.

3.9 Version 5: Experimenting with FIFOs

Version 5 was created to determine if the use of FIFOs would be beneficial to the design. If they do not add any benefits, the array access logic could be optimized to use the minimum number of loop iterations instead using the logic introduced in version 4 which accessed every element in an array.
3.9.1 Version 5: Data Flow

The data flow in version 5 was identical to the data flow in version 4. The implementation change was the manual unrolling of the leaf products loop by a factor of 3. In the leaf products loop, the destination of a calculated product cycles through the stage 1 low, middle, and high product arrays. In version 4, the value of a counter which would wrap around at 3 was used to determine which array should be the destination of each iteration’s calculated product. This modulus 3 counter logic was interfering with the implementation of the stage 1 arrays as FIFOs. In general, this type of logic also limited the degree to which the loop logic could be optimized by Vivado HLS. Since the code size would not increase much if the loop was manually unrolled by a factor of 3, it was determined to be an optimization worthy of being tested. This change is reflected in the updated pseudocode shown in Algorithm 6.

3.9.2 Version 5: High-Level Synthesis

The stage 1 arrays were successfully synthesized as FIFOs using a combination of the STREAM and DATAFLOW directives. In the end, the FIFOs had no noticeable performance gain over memories for this function. There was, however, a resource usage benefit. When the multiplication operand size was set to only 8 32-bit coefficients, the default implementations of the stage 1 arrays were arrays of flip-flops. In this case, the utilization of a FIFO saved a few hundred flip-flops at the expense of about a 100 LUTs for each array. When the operand size was increased to 512 32-bit coefficients, the stage 1 arrays were implemented using block RAMs by default. When FIFOs were used, half of the block RAMs required for each array were removed at the expense of 131 addition flip-flops and 481 addition LUTs per array.

1 leaf_products : foreach group of three leaves in the Karatsuba tree do
2     Calculate low leaf product and store in Stage 1 low product array.
3     Calculate middle leaf product (before subtractions) and store in Stage 1 middle product array.
4     Calculate high leaf product and store in Stage 1 high product array.
5 end
6 mid_sub : foreach group of a low, middle, and high leaf product do
7     Subtract $A_L B_L$ and $A_H B_H$ from the leaf product $(A_L + A_H)(B_L + B_H)$ to get middle product.
8     Select Stage 2 destination array and store low, middle, and high products.
9 end
10 level : foreach level of the Karatsuba tree do
11     section_mid_sub : foreach group of a low, middle, and high product in the current level do
12         Subtract $A_L B_L$ and $A_H B_H$ from the leaf product
13         $(A_L + A_H)(B_L + B_H)$ to get middle product.
14         Store low, middle, and high products in Stage 3 arrays.
15     end
16     section : foreach intermediate product in a level do
17         section_build : foreach coefficient in the next intermediate product do
18             if Writing lower half of intermediate product then
19                 Load low product coefficient.
20             else if Writing upper half of intermediate product then
21                 Load high product coefficient.
22             end
23             if Writing middle portion of intermediate product then
24                 Load middle product coefficient.
25             end
26             if Writing middle portion of intermediate product then
27                 Sum loaded middle product with loaded low or high product and assign the result to temporary variable.
28             end
29             if Building final product then
30                 Write temporary variable to output product array.
31             else
32                 Select destination Stage 2 array.
33                 Write temporary variable to Stage 2 array.
34             end
35         end
36     end
37 end
3.10 Version 6: Switch to Two-dimensional Arrays

3.10.1 Version 6: Algorithm

The HLS results from version 4 showed that if arrays are divided up into stages, the interval of the entire function can become equal to the largest interval among the dataflow processes instead of the sum of the intervals of all the consecutive loops in the function. Since the mid_sub and level loop process still had a very high interval, the next step was to break up this single process into a unique process for every iteration of the level loop. This would bring the interval of the function down to the interval of one iteration of the level loop instead of the sum of all the level loop iterations’ intervals.

In version 4, new stages had been created by declaring multiple arrays instead of a single array. This could not be done for arrays used in the level loop since the number of loop iterations was a variable dependent upon the operand size. New stages could only be added by changing the arrays from one-dimensional arrays to two-dimensional arrays. The second dimension would be identical in size and use to the original one-dimensional array, and the first dimension would represent the level loop iteration at which it should be used. This new implementation was placed in a new source file called \textit{karatsuba_combine_2d.c}.

3.10.2 Version 6: Data Flow

The data flow of version 6 is the same as that of version 4 and 5 except for every stage 2 and stage 3 array shown in Figure 3.14, a new, independent array is used for every iteration of the level loop.
3.10.3 Version 6: High-Level Synthesis

In order for Vivado HLS to recognize that each loop iteration uses a different array, the stage 2 and 3 arrays must be partitioned. The ARRAY_PARTITION directive must be applied to dimension 1 of all stage 2 and 3 arrays with a factor equal to the number of iterations in the level loop ($\log_2(\text{OP_LEN}) - 1$ where \text{OP_LEN} is the number of coefficients in an operand).

The version 6 synthesis results were exactly as was expected. With the DATAFLOW directive applied to the level loop and the level loop fully unrolled, every iteration of the section_mid_sub loop and section loop was made into its own dataflow process. The interval of the entire \texttt{karatsuba_combine_2d} function was now equal to the interval of the section loop with the largest number of iterations. The pipeline of the function is illustrated in Figure 3.16. In the figure, the variable $n$ is equal to $\log_2(\text{OP_LEN}) - 1$ where \text{OP_LEN} is the number of coefficients in an operand.

Table 3.6 gives the resource usage for version 6 for operands with 512 32-bit coeffi-
CHAPTER 3. HIGH-LEVEL SYNTHESIS OF KARATSUBA MULTIPLICATION

<table>
<thead>
<tr>
<th>Vivado HLS Version</th>
<th>Clock Frequency (MHz)</th>
<th>Resource Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>BRAM</td>
</tr>
<tr>
<td>2016.4</td>
<td>100</td>
<td>1,386</td>
</tr>
<tr>
<td>2017.2</td>
<td>200</td>
<td>1,630</td>
</tr>
<tr>
<td>2017.2</td>
<td>222</td>
<td>1,630</td>
</tr>
</tbody>
</table>

Table 3.6: Resource usage for version 6 of karatsuba_combine. FPGA model: xcvu9p-fsgd2104-3-e-es1.

<table>
<thead>
<tr>
<th>Vivado HLS Version</th>
<th>Clock Frequency (MHz)</th>
<th>Function Interval (cycles)</th>
<th>Estimated Execution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016.4</td>
<td>100</td>
<td>19,683</td>
<td>0.196830</td>
</tr>
<tr>
<td>2017.2</td>
<td>200</td>
<td>21,871</td>
<td>0.109355</td>
</tr>
<tr>
<td>2017.2</td>
<td>222</td>
<td>21,871</td>
<td>0.098419</td>
</tr>
</tbody>
</table>

Table 3.7: Execution times for version 6 of karatsuba_combine. FPGA model: xcvu9p-fsgd2104-3-e-es1.

coefficients. In all cases, the level loop is fully unrolled, and the leaf_products, mid_sub, section_mid_sub, and section_build loops are pipelined. The input leaf_ops array, the stage 1 arrays, an the output product array also had 4 cyclic partitions. As should be expected, the introduction of a new array for each level loop iteration caused a significant increase in BRAM usage compared to the usage reported for previous versions. The flip-flop and LUT usage, however, was less than the usage given earlier for version 4 since the section_build loop in this version was pipelined instead of fully unrolled like the low, high, and middle product loops in version 4.

Before version 6, Vivado HLS version 2016.4 had been the only tool used for synthesis. During the development of version 6, comparisons were done between the results given by Vivado HLS 2016.4 and the newly available Vivado HLS 2017.2. Version 2017.2 often reported a slightly higher interval, but it created circuits which could run at more than double the clock frequency of those generated by version 2016.4. As shown in Tables 3.6 and 3.7, the circuits generated by 2017.2 could run at a clock frequency of up to 222 MHz (4.5 ns period), while 2016.4 kept the clock frequency closer to the default (100 MHz).

Table 3.7 shows the execution times for version 6. For the first time since version
2, the execution time is under 1 ms. It is also about four times less than that of version 2.

3.11 Version 7: Optimized Section Loop

3.11.1 Version 7: Algorithm

Now that the function had a full pipeline of dataflow process functions, the basis of the slowest process function, the section loop, could be optimized. The version 2 accelerators attained high throughput in the section loop by unrolling all of the subloops. This removed much of the control logic and allowed Vivado HLS to optimize at the expense of a high resource usage. The section_build loop introduced in version 4 went to the other extreme by utilizing control logic to direct one value at a time. Since this loop could not be efficiently unrolled, the high loop iteration count required by the section_build loop translated to a highest interval of the functions dataflow process pipeline. Version 7 sought to lower this interval by replacing the section_build loop with a solution with simpler control logic.

The new section loop logic is essentially the section_build loop unrolled by a factor of 3 and simplified. One iteration of the section loop now constructs three sections each iteration. This unrolling allowed all the multiplexing logic from the section_build loop to be removed. Also, and more importantly, it brought the number of loop iterations of the section loop down by a factor of 3. The new code writes all the low and high products together in one loop. This was done to emphasize to the synthesis tool that these operations are not dependent and can be performed in parallel. If necessary, this could be further emphasized with the use of a DEPENDENCE directive.

The section_mid_sub loop was also refactored. When it was created in version 4, this loop was specifically built to iterate over all of the values in each array in order to mimic the behavior of a FIFO. This led to a loop with numerous useless loop
Algorithm 7: Pseudocode for Version 7 of `karatsuba_combine`.

```
leaf_products : foreach group of three leaves in the Karatsuba tree do
  Calculate low leaf product and store in Stage 1 low product array.
  Calculate middle leaf product (before subtractions) and store in Stage 1 middle product array.
  Calculate high leaf product and store in Stage 1 high product array.
end

mid_sub : foreach group of a low, middle, and high leaf product do
  Subtract $A_L B_L$ and $A_H B_H$ from the leaf product $(A_L + A_H)(B_L + B_H)$ to get middle product.
  Select Stage 2 destination array and store low, middle, and high products.
end

level : foreach level of the Karatsuba tree do
  section_mid_sub : foreach group of a low, middle, and high product in the current level do
    Subtract $A_L B_L$ and $A_H B_H$ from the leaf product $(A_L + A_H)(B_L + B_H)$ to get middle product.
    Store low, middle, and high products in Stage 3 arrays.
  end
  section : foreach intermediate product in a level do
    low_high : foreach coefficient in the low and high product do
      Write low and high products into next intermediate product destination in Stage 2 arrays.
    end
    mid : foreach coefficient in the middle product do
      Add middle product into next intermediate product destination in Stage 2 arrays.
    end
  end
end

final_product_mid_sub : foreach coefficient in the Stage 2 middle products array do
  Subtract the low and high products from the middle product (middle subtractions).
end

final_product_low : foreach coefficient in the final low product do
  Write the low product into the output product array.
end

final_product_high : foreach coefficient in the final high product do
  Write the high product into the output product array.
end

final_product_mid : foreach coefficient in the final middle product do
  Add the middle product into the output product array.
end
```
iterations due to unused space in the arrays. The refactored loop contains address offset logic which causes the loop to skip all unnecessary calculations and thereby achieve the minimum possible number of loop iterations.

### 3.11.2 Version 7: High-Level Synthesis

Table 3.8 presents the resource usage for the version 7 hardware accelerators. The manual unrolling of the section_build loop led to a significant increase in flip-flop and LUT usage. This additional hardware, however, enabled the version 7 accelerator to achieve about half the execution time of the version 6 accelerator. Table 3.9 lists these times. These results are for operands with 512 32-bit coefficients. In all cases, the level loop is fully unrolled, and the leaf products, mid_sub, and section_mid_sub loops are pipelined. Both accelerators could run at a maximum clock frequency of 222 MHz.
3.12 Addition of Modular Reduction

In order for the karatsuba_combine algorithm to be usable in a homomorphic encryption algorithm, it must support modular reduction of its product coefficients. This feature was implemented in-line with all the karatsuba_combine_2d logic discussed in this chapter and saved under the function name karatsuba_combine_2d_modCoeff. The addition of this functionality had the added benefit of limiting the size of the coefficients of the intermediate products. This helped to minimize the size of the memory structures used to store the intermediate data in the hardware accelerator.

Barrett reduction was used to reduce the leaf products. It was implemented in a loop between the leaf_products loop and the mid_sub loop. For all other cases, reduction was performed with a simple if statement that checked if a value was greater than the modulus. If the if statement returned true, the modulus was subtracted from the value.

If statement checks were used in most cases instead of full Barrett reduction in order to minimize hardware usage. Barrett reduction required a large multiplication. This was practical when the values being reduced could have two times the number of bits in the modulus, but when the values could only be one or two bits larger than the modulus, using if statements with subtractions was more hardware efficient.

3.13 Parallelization

After version 7, the interval of the accelerator was constrained by the section and section_mid_sub loops. These loops were given their own dataflow processes and were pipelined to the lowest possible interval. Their final interval was determined by the width of the Karatsuba tree. These loops were required to iterate across all the intermediate products in each level. The only way to decrease the overall function interval was to decrease the trip count of these loops. This was done by creating
CHAPTER 3. HIGH-LEVEL SYNTHESIS OF KARATSUBA MULTIPLICATION

Figure 3.17: Karatsuba tree for 4-term operands with the three smaller parallel sub-trees in dotted boxes.

Algorithm 8: Pseudocode for karatsuba_combine_2d_parallel.

1. Call karatsuba_combine_2d to calculate the root node low product.
2. Call karatsuba_combine_2d to calculate the root node high product.
3. Call karatsuba_combine_2d to calculate the root node middle product.
4. final_product_mid_sub : foreach coefficient in the middle product array do
   5. Subtract the low and high products from the middle product (middle subtractions).
   end
5. final_product_low : foreach coefficient in the final low product do
   6. Write the low product into the output product array.
   end
6. final_product_high : foreach coefficient in the final high product do
   7. Write the high product into the output product array.
   end
7. final_product_mid : foreach coefficient in the final middle product do
   8. Add the middle product into the output product array.
   end

a wrapper function, karatsuba_combine_2d_parallel, which used karatsuba_combine_2d three times to evaluate the smaller internal trees which calculate the low, high, and middle products for the root node of the Karatsuba tree. These smaller sub-trees are shown in boxes in Figure 3.17. The pseudocode for karatsuba_combine_2d_parallel is shown in Algorithm 8.

Vivado HLS implemented the three function calls as three parallel instances of karatsuba_combine_2d. Now, instead of the section and section_mid_sub loop having to iterate across the entire width of the large tree, three smaller instances of these
loops iterated in parallel across smaller trees. The wrapper function took the output of these three function calls and performed the last combination step required to calculate the final product. This parallelization resulted in a significant drop in the function’s overall interval.

Given the performance benefits of using three internal function calls, another version of the wrapper function, `karatsuba_combine_2d_parallel9`, was implemented which moved down the tree one more level and called nine instances of the `karatsuba_combine_2d` function. These nine function calls evaluated nine sub-trees as denoted in Figure 3.18 with dotted boxes. The wrapper function took the results from the nine function calls and performed the two remaining combination operations at the top of the tree. This increase in parallelization once again significantly boosted the performance of the Karatsuba hardware accelerator.

Modular reduction was added to `karatsuba_combine_2d_parallel9` and saved as `karatsuba_combine_2d_parallel9_modCoeff`. This new function was synthesized for operand sizes ranging from 32 to 2,048 coefficients with coefficient sizes ranging from 1 to 3 64-bit limbs (i.e. 64 to 192 bits). The resource usage data from these synthesis runs is shown in Table 3.10. For all cases shown, the achieved clock frequency was 200 MHz. An example synthesis directives file for these runs is shown in Appendix A.

The target FPGA for the synthesis runs shown in Table 3.10 was a Xilinx Virtex UltraScale+ (Model Number: xcvu9p-fsgd2104-3-e-es1). This FPGA was selected

---

**Figure 3.18:** Ternary tree resembling a Karatsuba tree for 8-term operands with the nine smaller parallel sub-trees in dotted boxes.
### Table 3.10: Resource usage for karatsuba_combine_2d_parallel9_modCoeff.

<table>
<thead>
<tr>
<th>Number of Operand Coefficients</th>
<th>Coefficient Bit Width (bits)</th>
<th>Resource Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>BRAM</td>
</tr>
<tr>
<td>32</td>
<td>64</td>
<td>648</td>
</tr>
<tr>
<td>32</td>
<td>128</td>
<td>1,752</td>
</tr>
<tr>
<td>32</td>
<td>192</td>
<td>2,415</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>1,416</td>
</tr>
<tr>
<td>64</td>
<td>128</td>
<td>2,832</td>
</tr>
<tr>
<td>64</td>
<td>192</td>
<td>3,900</td>
</tr>
<tr>
<td>128</td>
<td>64</td>
<td>1,848</td>
</tr>
<tr>
<td>128</td>
<td>128</td>
<td>3,696</td>
</tr>
<tr>
<td>128</td>
<td>192</td>
<td>5,088</td>
</tr>
<tr>
<td>256</td>
<td>64</td>
<td>2,280</td>
</tr>
<tr>
<td>256</td>
<td>128</td>
<td>4,560</td>
</tr>
<tr>
<td>256</td>
<td>192</td>
<td>6,276</td>
</tr>
<tr>
<td>512</td>
<td>64</td>
<td>2,712</td>
</tr>
<tr>
<td>512</td>
<td>128</td>
<td>5,424</td>
</tr>
<tr>
<td>512</td>
<td>192</td>
<td>7,464</td>
</tr>
<tr>
<td>1,024</td>
<td>64</td>
<td>9,192</td>
</tr>
<tr>
<td>1,024</td>
<td>128</td>
<td>16,872</td>
</tr>
<tr>
<td>1,024</td>
<td>192</td>
<td>26,040</td>
</tr>
<tr>
<td>2,048</td>
<td>64</td>
<td>22,632</td>
</tr>
<tr>
<td>2,048</td>
<td>128</td>
<td>43,530</td>
</tr>
<tr>
<td>2,048</td>
<td>192</td>
<td>66,126</td>
</tr>
</tbody>
</table>

FPGA model: xcvu9p-fsgd2104-3-e-es1.
Table 3.11: Available resources for the FPGA targeted in this work and the Xilinx FPGA with the most available BRAMs.

<table>
<thead>
<tr>
<th>Model Number</th>
<th>Available Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>xcvu9p-fsgd2104-3-e-es1 (UltraScale+)</td>
<td>BRAM: 4,320, DSP48E: 6,840, FF: 2,364,480, LUT: 1,182,240</td>
</tr>
<tr>
<td>xcvu190-flgb2104-3-e (UltraScale)</td>
<td>BRAM: 7,560, DSP48E: 1,800, FF: 2,148,480, LUT: 1,074,240</td>
</tr>
</tbody>
</table>

Most of the accelerator configurations given in Table 3.10 could fit on one of the two devices described in Table 3.11. The only exceptions are the accelerators which support 1,024 and 2,048 coefficient operands. The high BRAM usage stems from the parallelization of the algorithm. The nine separate function calls used to evaluate the nine sub-trees all require their own sets of BRAMs. Also, even if the algorithm contained fewer parallel operations, as the operands and coefficients get larger, the total memory size required to store all the intermediate product data on the FPGA fabric becomes difficult to handle for even the largest devices.

Given the resource usage constraints and increased code size due to parallelization, it was determined that the number of parallel function calls should not be increased beyond nine. The `karatsuba_combine_2d_parallel9_modCoeff` was determined to be the best compromise between performance and resource usage. It was the performance of this design which was compared against the FLINT library in order to determine if High-Level Synthesis could produce an accelerator with a performance speedup over
### Chapter 3. High-Level Synthesis of Karatsuba Multiplication

#### Table 3.12: Speedup (CPU Execution Time/FPGA Execution Time) of FPGA accelerator over FLINT for modular polynomial multiplication. *More than 4,320 FPGA BRAMs required. **More than 7,560 FPGA BRAMs required.

<table>
<thead>
<tr>
<th>Coefficient Size (bits)</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>6.31</td>
<td>11.44</td>
<td>8.08</td>
<td>9.16</td>
<td>8.50</td>
<td>6.35**</td>
<td>4.78**</td>
</tr>
<tr>
<td>128</td>
<td>48.68</td>
<td>54.52</td>
<td>69.13</td>
<td>74.16*</td>
<td>71.42*</td>
<td>50.88**</td>
<td>51.67**</td>
</tr>
<tr>
<td>192</td>
<td>82.45</td>
<td>110.00</td>
<td>118.90*</td>
<td>136.71*</td>
<td>127.35*</td>
<td>96.41**</td>
<td>81.08**</td>
</tr>
</tbody>
</table>

The performance of the `karatsuba_combine_2d_parallel9_modCoeff` hardware accelerator was compared to the modular polynomial multiplication function `fmpz_mod_poly_mul` from the FLINT library. The FLINT function’s execution time was captured on a computer with a quad-core AMD A10-7850K running at a 3.7 GHz clock speed. The computer was running a Red Hat Enterprise Linux operating system with 14.5 GB of RAM. Table 3.12 shows the hardware accelerator’s speedup over FLINT’s function for a range of operand and coefficient sizes. Note that speedup values marked with a single asterisk (*) would fit on the Xilinx Virtex UltraScale FPGA with the highest available number of BRAMs. Speedup values marked with two asterisks (**) would not fit on FPGAs which can currently be targeted by Vivado HLS 2017.2. Finally, the speedup values without any asterisks would fit on the device targeted by Vivado HLS for all the synthesis runs, the Virtex UltraScale+ FPGA described in Table 3.11.

For all the cases tested, the hardware accelerator was able to achieve significant speedup over FLINT’s highly-optimized software implementation. The magnitude of achieved acceleration can be clearly seen in the bar graph presented in Figure 3.19. The accelerator’s ability to maintain the same performance as the coefficient size increases gives it a noticeable advantage over FLINT’s software implementation. As FPGA size increases, this advantage can be leveraged for larger and larger operations.
Figure 3.19: Final speedup versus FLINT.
Chapter 4
Conclusions and Future Work

Even with the latest versions of HLS tools, the process of converting software into hardware must be approached with care. It may be best to think of the combination of software source code and synthesis directives as its own high-level hardware description language. For those used to the high degree of control given by traditional hardware description languages, the HLS process may feel very constraining. On the other side of the spectrum, engineers who are only familiar with software development will need to learn about the basic constructs and optimizations used in hardware design in order to fully leverage the features of the HLS tool and avoid generating an accelerator which has inferior performance to the software implementation.

Despite the learning curve, this work has shown that High-Level Synthesis is becoming a powerful tool in the world of hardware acceleration. The speedup results achieved by the proof-of-concept accelerator presented in this work shows definitively that this high-level design technique can generate high performance hardware. The benefits of this design strategy will only broaden as High-Level Synthesis tools are further developed and improved.

The next step is to expand this work to a full library of accelerators for homomorphic encryption. Future work will need to focus on implementing the remaining operations required for a specific homomorphic encryption scheme. The resource constraints of FPGAs will need to be balanced with the high memory requirements of
homomorphic schemes. Acceleration of HE will likely require the use of multiple FP-GAs working alongside CPUs and GPUs. Once these systems have been constructed, the final step is to deploy accelerated HE to the cloud.
Appendix A:  \textit{karatsuba\_combine\_2d\_parallel9\_modCoeff}  

Synthesis Directives

########################################################################
## This file is generated automatically by Vivado HLS.
## Please DO NOT edit it.
## Copyright (C) 1986−2017 Xilinx, Inc. All Rights Reserved.
########################################################################

set directive pipeline − rewind "karatsuba\_combine\_2d\_modCoeff/leaf\_products\_loop"
set directive pipeline − rewind "karatsuba\_combine\_2d\_modCoeff/mod\_leaf\_products\_loop"
set directive pipeline − rewind "karatsuba\_combine\_2d\_modCoeff/mod\_products\_loop"
set directive pipeline − rewind "karatsuba\_combine\_2d\_modCoeff/mid\_sub\_loop"
set directive unroll "karatsuba\_combine\_2d\_modCoeff/level\_loop"
set directive pipeline − rewind "karatsuba\_combine\_2d\_modCoeff/section\_mid\_sub\_loop"
set directive pipeline − rewind "karatsuba\_combine\_2d\_modCoeff/final\_product\_mid\_sub\_loop"
set directive pipeline − rewind "karatsuba\_combine\_2d\_modCoeff/final\_product\_mid\_loop"
set directive pipeline − rewind "karatsuba\_combine\_2d\_modCoeff/final\_product\_high\_loop"
set directive pipeline − rewind "karatsuba\_combine\_2d\_modCoeff/final\_product\_low\_loop"
set directive pipeline − rewind "karatsuba\_combine\_2d\_modCoeff/final\_product\_mid\_loop"
set directive pipeline − rewind "karatsuba\_combine\_2d\_modCoeff/product\_mod\_loop"
set directive array partition − type complete − dim 1 "karatsuba\_combine\_2d\_modCoeff"
    s2\_low\_products\_mod
set directive array partition − type complete − dim 1 "karatsuba\_combine\_2d\_modCoeff"
    s2\_mid\_products\_mod
set directive array partition − type complete − dim 1 "karatsuba\_combine\_2d\_modCoeff"
    s2\_high\_products\_mod
set directive array partition − type complete − dim 1 "karatsuba\_combine\_2d\_modCoeff"
    s3\_low\_products
set directive array partition − type complete − dim 1 "karatsuba\_combine\_2d\_modCoeff"
    s3\_mid\_products
set directive array partition − type complete − dim 1 "karatsuba\_combine\_2d\_modCoeff"
    s3\_high\_products
set directive array partition − type block − factor 4 − dim 1 "karatsuba\_combine\_2d\_parallel9\_modCoeff" product
set directive array partition − type cyclic − factor 4 − dim 1 "karatsuba\_combine\_2d\_parallel9\_modCoeff" leaf\_ops0
set directive array partition − type cyclic − factor 4 − dim 1 "karatsuba\_combine\_2d\_parallel9\_modCoeff" leaf\_ops1
set directive array partition − type cyclic − factor 4 − dim 1 "karatsuba\_combine\_2d\_parallel9\_modCoeff" leaf\_ops2
set directive array partition − type cyclic − factor 4 − dim 1 "karatsuba\_combine\_2d\_parallel9\_modCoeff" leaf\_ops3
set directive array partition − type cyclic − factor 4 − dim 1 "karatsuba\_combine\_2d\_parallel9\_modCoeff" leaf\_ops4
set directive array partition − type cyclic − factor 4 − dim 1 "karatsuba\_combine\_2d\_parallel9\_modCoeff" leaf\_ops5
set directive array partition − type cyclic − factor 4 − dim 1 "karatsuba\_combine\_2d\_parallel9\_modCoeff" leaf\_ops6
set directive array partition − type cyclic − factor 4 − dim 1 "karatsuba\_combine\_2d\_parallel9\_modCoeff" leaf\_ops7
set directive array partition − type cyclic − factor 4 − dim 1 "karatsuba\_combine\_2d\_parallel9\_modCoeff" leaf\_ops8
set directive array partition − type block − factor 2 − dim 1 "karatsuba\_combine\_2d\_parallel9\_modCoeff" s3\_low\_products

79
APPENDIX A.  

KARATSUBA_COMBINE_2D_PARALLEL9_MODCOEFF
SYNTHESIS DIRECTIVES

set_directive_array_partition -type block -factor 2 -dim 1
  karatsuba_combine_2d_parallel9_modCoeff s3 mid_products
set_directive_pipeline -rewind "karatsuba_combine_2d_parallel9_modCoeff/mid_sub9_loop"
set_directive_pipeline -rewind "karatsuba_combine_2d_parallel9_modCoeff/
  product_low_high_loop"
set_directive_pipeline -rewind "karatsuba_combine_2d_parallel9_modCoeff/
  product_mid_loop"
set_directive_pipeline -rewind "karatsuba_combine_2d_parallel9_modCoeff/
  product_mod_loop"
set_directive_pipeline -rewind "karatsuba_combine_2d_parallel9_modCoeff/
  final_product_mid_sub_loop"
set_directive_pipeline -rewind "karatsuba_combine_2d_parallel9_modCoeff/
  final_product_low_high_loop"
set_directive_pipeline -rewind "karatsuba_combine_2d_parallel9_modCoeff/
  final_product_low_loop"
set_directive_pipeline -rewind "karatsuba_combine_2d_parallel9_modCoeff/
  final_product_mid_loop"
set_directive_pipeline -rewind "karatsuba_combine_2d_parallel9_modCoeff/
  final_product_mod_loop"
set_directive_dataflow "karatsuba_combine_2d_modCoeff"
set_directive_dataflow "karatsuba_combine_2d_modCoeff/level_loop"
set_directive_dataflow "karatsuba_combine_2d_parallel9_modCoeff"
set_directive_array_partition -type block -factor 2 -dim 1
  karatsuba_combine_2d_parallel9_modCoeff s3 high_products
set_directive_pipeline -rewind "karatsuba_combine_2d_parallel9_modCoeff/low_high_product_loop"
set_directive_pipeline -rewind "karatsuba_combine_2d_parallel9_modCoeff/mid_product_loop"
set_directive_pipeline -rewind "karatsuba_combine_2d_parallel9_modCoeff/
  final_product_low_loop"
set_directive_clock "karatsuba_combine_2d_parallel9_modCoeff" 5nsPeriod
Bibliography


