University of Windsor
Scholarship at UWindsor

Electronic Theses and Dissertations

10-19-2015

High Level Synthesis and Evaluation of the Secure Hash Standard for FPGAs

Ian Spencer Janik
University of Windsor

Follow this and additional works at: http://scholar.uwindsor.ca/etd

Recommended Citation
Janik, Ian Spencer, "High Level Synthesis and Evaluation of the Secure Hash Standard for FPGAs" (2015). *Electronic Theses and Dissertations.* Paper 5470.

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.

High Level Synthesis and Evaluation of the Secure Hash Standard for FPGAs

By

Ian Janik

A Thesis
Submitted to the Faculty of Graduate Studies
through the Department of Electrical and Computer Engineering
in Partial Fulfillment of the Requirements for
the Degree of Master of Applied Science
at the University of Windsor

Windsor, Ontario, Canada

2015

© 2015 Ian Janik

High Level Synthesis and Evaluation of the Secure Hash Standard for FPGAs

By

Ian Janik

APPROVED BY:

R. Seth
Department of Civil and Environmental Engineering

E. Abdel-Raheem
Department of Electrical and Computer Engineering

M. Khalid, Advisor
Department of Electrical and Computer Engineering

September 16, 2015

Declaration of Co-Authorship

I. Co-Authorship Declaration

This thesis also incorporates the outcome of a joint research undertaken in collaboration with Qing Tang under the supervision of Dr. Mohammed Khalid. The collaboration is covered in Chapter 2 of the thesis. In all cases, the key ideas, primary contributions, experimental designs, data analysis and interpretation, were performed by the author, and the contribution of coauthor was primarily through the help in evaluation of the Altera SDK for OpenCL that comprises the paper "An overview of Altera SDK for OpenCL: A user perspective" that was published at the Electrical and Computer Engineering (CCECE), 2015 conference. Ideas of that paper are used in this thesis.

I am aware of the University of Windsor Senate Policy on Authorship and I certify that I have properly acknowledged the contribution of other researchers to my thesis, and have obtained written permission from each of the co-author(s) to include the above material(s) in my thesis.

I certify that, with the above qualification, this thesis, and the research to which it refers, is the product of my own work.

II. Declaration of Previous Publication

This thesis includes 2 original papers that have been previously published/submitted for publication in peer reviewed journals, as follows:

Thesis Chapter	Publication title/full citation	Publication status*
Chapter 2	I. Janik, Q. Tang, and M. Khalid, "An overview of Altera SDK for OpenCL: A user perspective," *Electrical and Computer Engineering (CCECE), 2015 IEEE 28th Canadian Conference on*. pp. 559–564, 2015.	*published*
Chapters 2 and 3	I. Janik and M. Khalid "FPGA Synthesis and Evaluation of SHA-2 and SHA-3 Algorithms Using the Altera SDK for OpenCL"	*In preparation*

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

I declare that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

Abstract

Secure hash algorithms (SHAs) are important components of cryptographic applications. SHA performance on central processing units (CPUs) is slow, therefore, acceleration must be done using hardware such as Field Programmable Gate Arrays (FPGAs). Considerable work has been done in academia using FPGAs to accelerate SHAs. These designs were implemented using Hardware Description Language (HDL) based design methodologies, which are tedious and time consuming. High Level Synthesis (HLS) enables designers to synthesize optimized FPGA hardware from algorithm specifications in programming languages such as C/C++. This substantially reduces the design cost and time. In this thesis, the Altera SDK for OpenCL (AOCL) HLS tool was used to synthesize the SHAs on FPGAs and to explore the design space of the algorithms. The results were evaluated against the previous HDL based designs. Synthesized FPGA hardware performance was comparable to the HDL based designs despite the simpler and faster design process.

Acknowledgements

First and foremost I would like to thank my supervisor Dr. Khalid for giving me the opportunity to work with him to obtain my degree. I need to thank him for all of the guidance, advice, support, and encouragement he has given over the past two years.

I would like to thank Dr. Abdel-Raheem and Dr. Seth for agreeing to be a part of my committee and contributing their time from their busy schedules to provide insight on this project.

I'd also like to thank Ryan Tang for helping me when I needed it. Without his help learning and problem solving, this work would have not gone as smoothly.

I need to thank my mom for her continued support and encouragement throughout my university career that has allowed me to pursue my interests.

Lastly, I would like to thank my girlfriend, Jen. She motivates me to try my hardest and is always there to listen and help whenever I need it. I would not be where I am today without her.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

AOCL	Altera SDK for OpenCL
API	Application Program Interface
ASIC	Application Specific Integrated Circuit
BSP	Board Support Package
CAD	Computer Aided Design
CPU	Central Processing Unit
DDR3	Double Date Rate 3
DSE	Design Space Exploration
DSP	Digital Signal Processor
FIFO	First-In First-Out
FPGA	Field Programmable Gate Array
GPU	Graphical Processing Unit
HDL	Hardware Description Language
HLL	High Level Language
HLS	High Level Synthesis
IO	Input-Output
LE	Logic Element
LUT	Look Up Table
NDRange	N-Dimensional Range
NIST	National Institute of Standards and Technology

OpenCL	Open Computing Language
PRU	Percentage Resource Utilization
PCIe	Peripheral Component Interconnect Express
ROTL	Rotate Left
ROTR	Rotate Right
SDK	Software Development Kit
SHA	Secure Hash Algorithm
SHR	Shift Right
SoC	System on Chip
XOF	Extendable-Output Function

Chapter 1 Introduction

In today's world, electronic communication has become a necessity in both work and personal lives, especially since the advent of the Internet. There are online communication applications focusing on everything from commerce to the military that must be secure to ensure privacy is achieved.

Cryptography deals with securing electronic communication. A major part of cryptography is hash functions. They take an input message of any length and produce an output of a fixed length. Hash functions can only be used one way, meaning an input message cannot be derived from an output. Furthermore, the design of these functions cause a drastic change in the output when an input is even slightly changed. Applications of hash functions include file integrity, password verification, file identification, pseudorandom number generation, as well as key derivation [1], [2].

Due to the algorithms used in their computation, hash functions have poor speed performance on general purpose processors, such as central processing units (CPUs) in computers [3]. In order to achieve high speed and secure communication, hash functions must be accelerated using application specific processors. These processors can be implemented in dedicated hardware, such as application specific integrated circuits (ASICs), or they can be designed with reconfigurable hardware, with field programmable gate arrays (FPGAs). ASICs require the longest design time, but provide the best in performance and power efficiency. The development cost can only be overcome with the production of a high volume of circuits. If the time to market must be short, or the volume needed is too small, ASIC design can be unacceptable for a certain application.

FPGAs can be reconfigured to implement any hardware design. This makes them useful in prototyping of systems, or even in final products when the cost of using ASICs is too high. Another advantage of FPGAs is that their reprogrammable hardware can be changed to suit different applications at runtime.

Traditional FPGA design is done using hardware description languages (HDLs). This process is very tedious and time consuming compared to software development programming languages. HDL programming in FPGA designs can be compared to using assembly code for CPU programs. The ability to specify hardware designs in a high level and abstract way is needed to substantially expand the user base for FPGA technologies. The process of designing FPGA hardware using high level languages (HLLs) is called high level synthesis (HLS). This takes abstract designs and synthesizes them down to an HDL model. Existing tools can then compile it into FPGA hardware. HLS allows programmers that do not have the strong background needed in hardware design to easily utilize FPGAs in their applications. There have been many attempts to produce HLS tools, such as Xilinx Vivado, University of Toronto's LegUp, and the Altera Software Development Kit (SDK) for OpenCL (AOCL) [4]–[6]. These tools take different approaches to implementing HLS designs. The Altera SDK for OpenCL uses a heterogeneous computing approach to HLS.

Heterogeneous computing systems contain additional computational hardware besides CPUs. These systems are used for demanding applications that have poor performance on CPUs. The most popular heterogeneous computing components are graphical processing units (GPUs), also known as video cards. They are used to enable very fast rendering of graphics that will be displayed to the user and allow for greater complexity in user interfaces. A separate GPU is a necessity to play modern games on computers due to the amount of computation needed to simulate the physics and image aspects of the display. The clock frequency of a GPU is much slower than that of the CPU, but it contains multiple computational units that can all work in parallel, allowing a much higher throughput than a CPU. GPUs can be used for more than just graphics. By taking control of the processing power of the hardware, it is possible to use them in high throughput and parallel computational tasks. To allow programs to utilize GPUs and other hardware, the Open Computing Language (OpenCL) was developed [7]. This has become a popular standard for heterogeneous computing using CPUs, GPUs, digital signal processors (DSPs), and even FPGAs.

The AOCL uses the heterogeneous computing standard to allow programs to take advantage and utilize FGPA hardware installed in a computer. Rather than just converting high level designs to FPGA implementation, it focuses on utilizing FPGA hardware to accelerate computationally intensive parts of programs.

1.1 Thesis Goals

Cryptographic hash functions have many applications on computers and servers, but their speed performance is poor on CPU hardware. In a heterogeneous computing system with reconfigurable FPGA hardware available, the computation of the hash functions can be accelerated to increase speed performance of these applications.

1.1.1 General Objectives

- Develop an understanding of the OpenCL programming ecosystem and its relationship to HLS for FPGAs

- Explore the Secure Hash Algorithms (SHAs) defined by the National Institute of Standards and Technology (NIST).
- Develop C++ implementations of the SHAs to run on CPU.
- Use the AOCL to accelerate the OpenCL versions of the SHAs (functionally equivalent to C++ implementations) using an FPGA.
- Evaluate the speedup of the HLS model compared to CPU speed and other FPGA implementations in published literature.
- Test the viability of the Altera SDK for OpenCL platform in the acceleration with this common and computationally intensive task.

1.2 Thesis Outline

The goal of this thesis is to accelerate the secure cryptographic functions specified in the secure hash standard using the AOCL HLS tool for FPGAs. In Chapter 2, background information is presented on FPGAs, HLS, heterogeneous computing, OpenCL, SHAs, as well as a review of relevant work in literature. Chapter 3 presents the synthesis and evaluation of the SHAs. This includes a detailed description of the design process and final outcomes. Chapter 3 also includes the experimental results and discussion comparing them to previous related research work. Finally, in Chapter 4, the thesis concludes with a summary and discussion of possible future work in this area.

Chapter 2 Background and Related Work

There has been much research done in the area of accelerating cryptographic hash functions using FPGAs. This chapter focuses on the giving an overview of FPGAs, as well as HLS. It then goes on to describe heterogeneous computing systems, OpenCL, and the AOCL CAD tool that is used in this work. The secure hash standard and its functions are outlined in detail. The chapter ends with a literature review of published research using FPGAs to accelerate the functions of the secure hash standard.

2.1 Field Programmable Gate Arrays

FPGAs are prefabricated chips that contain reconfigurable hardware. They can be programmed and reprogrammed to implement different applications. FPGAs consist of memory blocks, input-output (IO) blocks, logic elements (LEs), as well as embedded hardware such as DSP blocks. All of these elements are attached with programmable interconnects [8]. Configuring these interconnects and LEs will implement a hardware design. Each LE is composed of a look-up table and a flip-flop. The architecture is shown in Figure 2.1. The function of the 4-input look-up table (4-LUT) is to implement any binary function of 4 inputs. For more complex functions, such as multiplication or division, the integrated DSP blocks are used to reduce the number of LEs needed, and increase speed of computation.

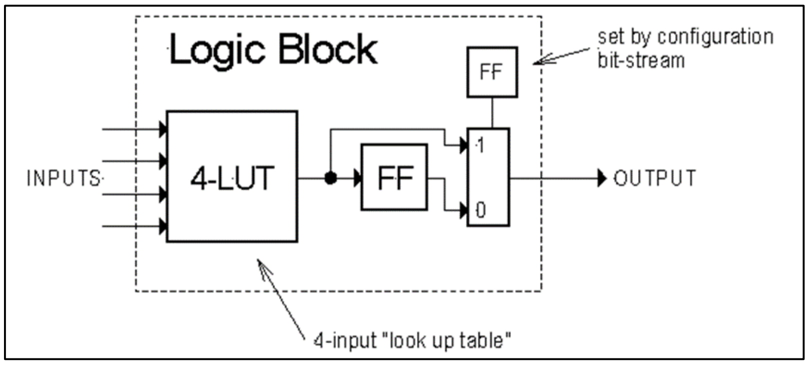

Figure 2.1: Logic Element Architecture [9]

Traditional FPGA design is done with hardware description languages (HDLs), such as Verilog or VHDL. Using mature computer aided design (CAD) tools, very fast and efficient FPGA hardware is generated from the HDL specifications.

2.2 High Level Synthesis

Design using HDLs for FPGAs is a tedious and time consuming process. The complexity of the programming limits the utilization of FPGA technology to developers with a good understanding of hardware design. The time needed to complete HDL designs is much longer than comparable applications in software. This is due to the high level and abstract specification possible in software development. To advance adoption and utilization, FPGA design needs to become as simple as software development. The process of high level synthesis (HLS) aims to allow FPGA models be specified in high level software languages such as C or C++. The way that HLS CAD tools work is to take algorithms written in high level languages (HLLs) and generate optimized HDL models that can then be synthesized to FPGA hardware using existing tools. These HLS tools allow people without the complex hardware design knowledge to use FPGAs in their applications. HLS tools can also speed up FPGA design to similar timeframes as software development.

Some currently available HLS tools include Xilinx Vivado, University of Toronto's Legup, and the Altera SDK for OpenCL [4]–[6]. These tools take different approaches in imagining the future of HLS. Vivado and Legup both take an algorithm that is specified in a HLL and create a functionally equivalent HDL model. The Altera SDK for OpenCL targets the acceleration of computationally intensive programs by utilizing an FPGA board installed in a computer. The FPGA is used to offload high intensive calculation from the CPU.

2.3 Heterogeneous Computing

Traditional computing systems contain only one type of processor, the CPU. When applications exist that have inherently poor performance on CPUs, there is no way to accelerate them on the available hardware. To increase computation speed, specialized hardware is added to the system. These systems that contain multiple types of processors are referred to as heterogeneous computing systems [10]. These specialized processors can be multicore CPUs, GPUs, DSPs, FPGAs, or other devices. The most common, GPUs, were designed to process complex graphics for displays, however, their hardware can be exploited to do other computational tasks. The different architecture of the GPU allows for high throughput parallel processing. Other types of heterogeneous components can run certain applications faster or more efficiently.

2.3.1 Open Computing Language

The most challenging aspect of using heterogeneous computing systems is creating programs that can take advantage of the installed hardware. Open computing language (OpenCL) was developed to be the standard programming language for heterogeneous systems [7]. Having one language that can accommodate all types of added devices allows for maximum portability of programs.

OpenCL was first developed by Apple Inc. in 2008, but has since been taken over by the Khronos Group [7]. This is a group of companies that work together to create royalty-free open standards for various computational applications. Some other popular standards they are responsible for maintaining, besides OpenCL, are OpenGL, WebGL, and Vulkan [11]. With multiple companies working together on the application program interface (API) of the OpenCL standard, compatibility between vendors is guaranteed. That doesn't mean that all the hardware is identical, but each implementation must provide the specified characteristics to be OpenCL certified.

The OpenCL model operates with two required components. The first is the host processor that is used to execute standard sequential program code. This host processor is usually the CPU in a workstation or server. The second component is the OpenCL device, which is used to run the parallel component of the code. There can be multiple OpenCL devices in a single system used together to achieve the highest possible throughput. The program code is broken into two parts as well, the host program and the OpenCL kernel. The host program, unsurprisingly, runs on the host processor, and is used to initialize the OpenCL device and data. The OpenCL kernel is the code that runs on the OpenCL device. Multiple instances of the kernel can be executed simultaneously depending on the capabilities of the OpenCL device targeted.

2.3.1.1 Host Program

The host program is no different from any other traditional CPU program. The OpenCL API is used to find, setup, and initialize the OpenCL devices. The API exists for C and C++ programming languages. The host program is needed to organize and send the proper data to the OpenCL implementation. The API includes many functions that make the application very portable between different systems. An example of this is the querying functions that are available. The host program can scan for installed OpenCL compatible hardware. Then, it can be programmed to use any configuration of hardware available. This allows the creation of OpenCL applications that can adapt to any system with the best possible results. For this to be practical, the kernel code must be compiled at runtime, which means a compiler must be included in the OpenCL software development kit (SDK) for that device. The kernel code can be compiled pre-runtime as well, but that limits the OpenCL devices that the program can utilize.

Another aspect of the host program is initialization and setup of the kernel. The OpenCL kernel can be thought of as a regular function in programming. The kernel has arguments that need to be set and it must be launched by the host. When an argument to a kernel contains a large amount of data it is best to use an OpenCL buffer object. The API contains functions for creating, reading, and writing to buffers.

Kernels can be launched in multiple instances. Each instance is defined as a work-item. These work-items can be organized into multiple work-groups, and can be identified in one, two, or three dimensions, called the N-Dimensional Range (NDRange).

The NDRange is specified depending on the application to make indexing more intuitive. The work-group size is then specified in each dimension. The number of work-groups will be determined by the total number of work-items and the work-group size. Proper organization of work-items can significantly improve performance of certain algorithms. Figure 2.2 shows the organization of the work-items and work-groups with an NDRange set to 2. This visualization can be expanded into 1 or 3 dimensions to understand the effect of setting the NDRange.

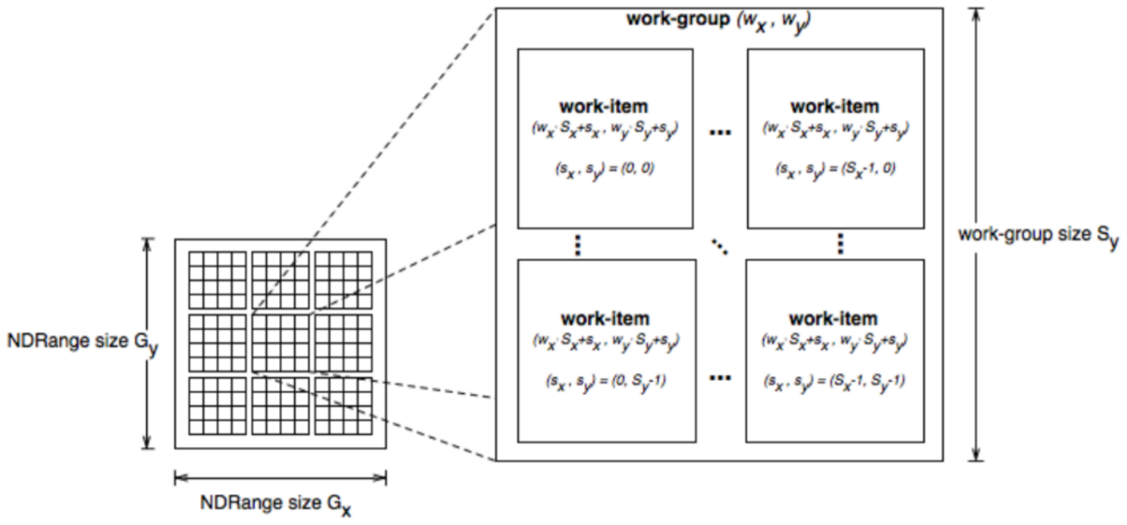
work-group (w_x, w_y)
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (0, 0)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (S_x-1, 0)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (0, S_y-1)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (S_x-1, S_y-1)$
work-group size S_y
NDRange size G_y
NDRange size G_x

Figure 2.2: Organization of OpenCL Work-Groups and Work-Items with NDRange of 2 [7]

Some OpenCL API commands are executed by adding them to a created command queue, which is another type of OpenCL object. A command queue exists for each OpenCL device and executes the appropriate commands in a sequential fashion. The commands that are executed from the queue include reading or writing from OpenCL buffers, and launching the kernel. These operations can be blocking or non-blocking. Blocking operations force the host program to stall until the command has finished execution. This is needed to ensure data integrity when reading/writing to buffers. Without blocking the host could change the data in the middle of processing the memory transfer. Non-blocking commands allow the host program to continue. Then the CPU is able to keep executing commands while the OpenCL device executes the kernel.

The host program can be compiled with any standard C or C++ compiler. There is no requirement to the compilation, other than linking to the appropriate OpenCL SDK libraries for the used devices.

2.3.1.2 OpenCL Kernel

The OpenCL kernel is the second component of an OpenCL application. This is the code that runs on the OpenCL device. Its execution is invoked by the host program. The kernel is written in a language that is a subset of the C-99 programming language [7]. The kernel function is of type *void* and does not return any value. In order to send data back to the host, an OpenCL buffer must be used that is given as an argument to the function. Other arguments can exist of single variables, just like other programming functions.

There are four memory types available in OpenCL implementations. The first is global memory. This is usually the RAM attached to the OpenCL device. It can be accessed by both the host program and the kernel. This is where the OpenCL buffers exist and it is used to transfer data between the host program and the kernel. It has the slowest access time of all the OpenCL memories. The second type of memory is constant memory. This is a subset of global memory that is only writable from the host. The kernel can only read this memory. It is used to transfer constant variables from the host to the kernel. The next type of memory is local memory. It is only accessible to the kernel. It is embedded in the OpenCL device and has very fast access time compared to that of global memory. Work-items in the same work-group access the same block of local memory but work-items in different work-groups cannot. This limitation keeps the memory access very fast and can impact the organization of the work-groups. The size of the local memory is also very small compared to global memory. The last type of memory is private memory which is independent for each work-item. The use of private memory is to hold the internal variables of the work-item. These memory types may be implemented differently in separate OpenCL devices, but they must have those characteristics. Proper utilization of the memory architectures available in OpenCL is the key to achieving good performance.

The OpenCL API has many functions built in that the kernel can execute such as math and print functions. A very important set of functions are the work-item functions. They are used to get the identifier of the current work-item in the work-group or globally. They can also be used to determine the NDRange, the number of work-groups, or number of work-items. These functions are important as they permit an easy way to specify which portion of a data set that specific work-item will operate on. It also allows for the creation of dynamic kernels that can adjust automatically for a different number or size of work-groups, without the need of redesign.

The kernel can be compiled in two different ways. The first is at runtime with the compiler built into the SDK for the target OpenCL device. This provides the greatest portability of the code, and allows the most systems to be supported. However, different types of OpenCL devices can be optimized in different ways to achieve the best performance. If a certain OpenCL device is the known target of the application, the proper optimization can be done and the kernel can be compiled before runtime. This makes the application restricted to certain systems, but with increased performance. It also takes away the need to compile at runtime, making it run slightly faster. This is also the only practical solution for some OpenCL devices as compilation can take much longer than the acceptable amount of time to complete at runtime, such as FPGAs that can take hours to days to compile complex kernels.

2.3.1.3 OpenCL Extensions

Devices must conform to the OpenCL standard to achieve the certification. It is still possible for certain devices to provide more functionality than specified in OpenCL. This is done through OpenCL extensions, which extend the functionality of the API for a certain device. It can be used to provide higher performance that what is offered in the standard, or enable features that are just not possible on different architectures.

2.3.2 The Altera SDK for OpenCL

The AOCL is a HLS tool that focuses on combining reconfigurable hardware and heterogeneous computing by integrating FPGA peripherals into computers. It uses the OpenCL standard to target FPGA accelerator cards as the OpenCL device, with custom hardware that is optimized for each application.

2.3.2.1 Host Program

The host program of an OpenCL application that targets Altera FPGAs is very similar to other OpenCL devices. The one caveat is that the kernel must be constructed in the host from the compiled binary image and not the OpenCL kernel source code. It cannot be compiled at runtime like other OpenCL devices. Other than that limitation, all that needs to be done is to link the host program to the OpenCL libraries included in the

SDK. Once that is completed, the program can be compiled by any C or C++ compiler and then executed.

2.3.2.2 Kernel

As stated previously, the major difference when using the AOCL compared to other OpenCL devices is that compilation of the kernel is not possible at runtime. In order to execute a kernel, it must be first compiled into the FPGA binary for the specific FPGA installed on the system. The reason this needs to be done before runtime is that it can take multiple hours to compile the kernel as FPGA synthesis is a slow process. The AOCL is compliant with version 1.0 of the standard, but the current version is 2.1 [6], [12]. This means that only the features and functions available in the 1.0 specification are available to use. Some components, like the print function, which was added in 1.2, have been added into the AOCL [13].

2.3.2.3 Extensions

As earlier described, it is possible for vendors to add more functionality to their OpenCL SDKs than that specified in the standard. These additions are called extensions. Due to the reconfigurable nature of FPGAs, some features have been added that are not possible to do on other OpenCL device architectures. One of these extensions is channels, which allows two concurrently running kernels to exchange data directly, without the need of global memory. This creates a first-in first-out (FIFO) buffer in the hardware that is responsible for data transfer. In other architectures, like a GPU, something like this is not possible with its static hardware. Depending on the application, this could be a huge performance boost when using FPGA. Other extensions are available that can allow direct input-output (IO) to the FPGA itself. Not needing to access data through RAM can allow for much faster operation.

2.3.2.4 FPGA Accelerator Cards

The FPGA targets of the AOCL are attached to accelerator cards. These cards are peripheral component interconnect express (PCIe) devices that can be installed on most modern computer motherboards. They can be bought off the shelf from manufacturers such as Nallatech, Terasic, and BittWare [14]–[16], or they can designed and built from scratch. Board support packages (BSPs) are available from each vendor for each board. This is required to map the AOCL to the hardware architecture of the board. There is also a blank BSP available to easily create a package for custom made boards. The main components of the FPGA accelerator cards are an Altera Statix V or Arria 10 FPGA, double date rate 3 (DDR3) RAM, interfacing hardware, and cooling components. Some of the cards can also include high speed network interfaces, or other communication ports to direct IO directly into the accelerator. Nallatech has just released an accelerator card featuring dual Arria 10 FPGAs, opening multi-FPGA possibilities on a single peripheral [17]. Other cards differ slightly by the exact FPGA chip installed, having varying amounts of speed and area available.

The other target device of the AOCL is a system on chip (SoC) platform [18]. This system uses an Altera Cyclone V SoC which contains an FPGA and ARM host processor inside a single chip. The ARM processor can execute the host program to execute OpenCL kernels on the FPGA portion of the chip. This opens the HLS aspect of the AOCL to the word of embedded applications, which is more traditional application of FPGAs.

In this thesis, the accelerator card that was used is the Terasic DE5-Net FPGA Developer board. It contains an Altera Stratix V GX FPGA, containing 622,000 LEs, and 4 GB of 1066 MHz DDR3 RAM.

2.3.2.5 The Tools

In order to program using the AOCL, there are three main software components needed. The first is the AOCL itself. Next is the BSP for the accelerator card. The last piece of software needed is the Quartus II CAD tool, which is Altera's tool for FPGA synthesis. Typically used for HDL design, it is very mature and efficient in the generation of FPGA hardware from HDL models. The AOCL contains two important programs. The first is the Altera Offline Compiler (AOC) which is responsible for compiling the kernel. The main function of AOC is to synthesize the OpenCL kernel to an intermediate Verilog form. It then calls Quartus II to compile that intermediate Verilog model into FPGA hardware. The second program included in the AOCL is the AOCL utility command, which is used to install the BSP, link the AOCL to the host program, as well as programming the FPGA device. The combination of these tools provides an intuitive and smooth HLS solution using the OpenCL standard.

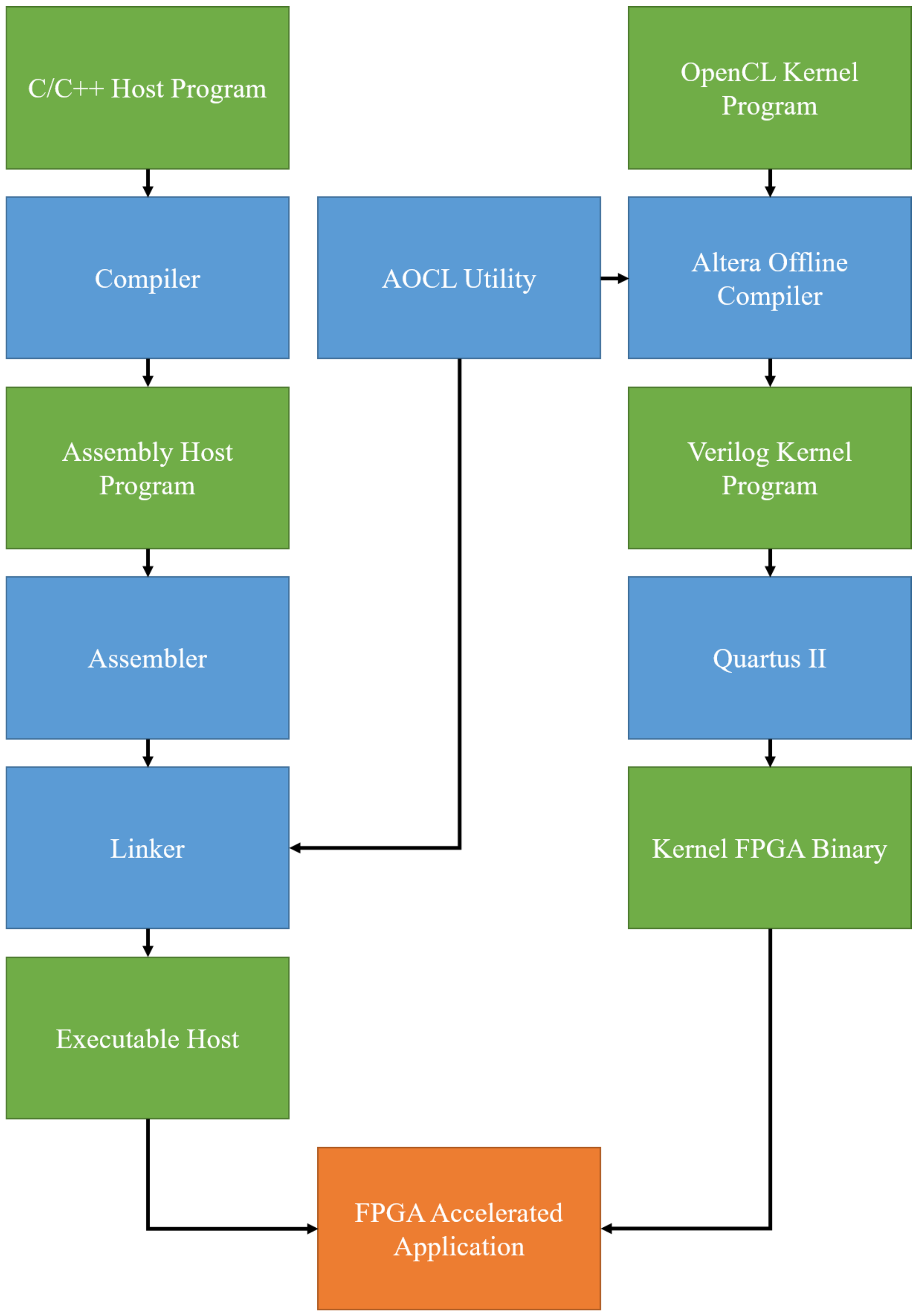

Figure 2.3: Altera SDK for OpenCL Programming Flowchart

2.4 The Secure Hash Standard

The secure hash standard specifies three families of secure cryptographic hash functions [19], [20]. The purpose of these functions is to take an input of any arbitrary length and provide a fixed size output, called the digest. The operation of the hash functions only work in one direction meaning there is no way to generate the input back from the output. They are also designed in such a way that even a small change in the input should produce a drastic change in the output. These secure hash algorithms (SHAs) have many cryptographic applications, such as file integrity, password storage, file identification, pseudorandom number generation, and key derivation [1], [2].

File integrity is a very important aspect of digital communication. When transferring a file from one system to another, it is necessary to ensure that the file has not been changed or corrupted in any way. When the transfer median is the Internet, it becomes even more important as attackers can try to modify or spoof files to execute malicious programs. If firmware files become corrupted in transfer and are then programmed to device, the hardware can fault and become unrepairable. Using SHAs, the file can be examined on both ends of communication and if the outputs match, then the file integrity has been kept and it is identical to the original. If the output is different, than some problem has occurred, such as modification, corruption, or spoofing, and the file can be retransferred before execution.

Another important application of SHAs is password storage. When dealing with password authentication, the ability to safely store that information is vital, as users tend to use the same passwords for multiple applications. If a system is compromised and the passwords are stored as plaintext, then the attacker has access to the users other applications as well. SHAs can be used to securely store password information. When a password is created or changed, it will first be concatenated with another string, called the salt. Then the salt and password will be set as the input of a secure hash function. The output generated can then safely be stored. When the user needs to be authenticated, the same process is done. If the hash matches the stored value, then the password was correct. The purpose of the salt is added security to augment the output hash from other applications using the same SHA and password. Since the input cannot be derived from the stored hash, when the system is compromised there is no way to get the users plaintext password. Furthermore, the salt can then be changed and the user can still safely use the old password again.

SHAs can also be useful for file identification. Files are usually identified by their name, but names can be easily changed. A SHA can produce a fixed size identifier of any size input. Using the SHA output of a file to identify it means that the file is identified by its data only. If the name is changed, the SHA will be the same as the actual data of the file that has not changed. A SHA digest is like a fingerprint of a file. This system is commonly used in peer-to-peer file sharing applications. If a user changes the name, the application can still understand which file it is. If the file is modified, then the digest will change and appropriate measures can be taken to prevent the sharing of that file. This also provides protection against attackers masquerading malicious files as desired files.

Pseudorandom number generation can be accomplished through the use of SHAs. Taking a sequential series of numbers and computing the SHA digest of each will produce a pseudorandom number sequence. Truncating the hash to a certain number of bits can limit the size of the numbers that can be generated.

In public key cryptography there exists two keys needed for secure communication [21]. The first key is called the public key, which is known. The second key is the private key, which is unknown. The relationship between these keys defines asymmetric encryption. When a public key is used to encrypt data, only the private key is able to decrypt it. The reverse is also true, as public key is the only way to decrypt private key encrypted data. There are many advantages to the public key cryptography scheme, such as inherent authentication. Data is known to come from a certain source if the public key is able to decrypt it, as the private key was used to encrypt it. In symmetric key cryptography, there is only a single key that is used for both encryption and decryption. For both symmetric and asymmetric cryptography, some applications require the derivation of the public or shared keys. An example application of this is WPA and WPA2 security for wireless networks [22]. The key is derived by using a password-based key derivation function, which uses a SHA along with the password and wireless network name to generate the key. In a public key setting, using key derivation to generate the public key adds another layer of security to the system.

2.4.1 SHA-1

The SHA-1 variant is the first family of hash functions specified by NIST in the secure hash standard [19]. It was introduced in 1995 as a replacement for SHA-0, which had security flaws. The family only contains a single function that is used to produce a 160-bit digest. The input message must be less than 2^{64} bits in length. The SHA-1 family has been deprecated [23], meaning that it is not to be used in new secure applications, however there are many legacy systems still rely on this 20 year old standard, keeping it relevant.

The algorithm of SHA-1 based on the Merkle–Damgård construction [24]**.** It takes an input message, *M*, which has a length, *l*, that is less than 2^{64} bits. The algorithm operates with a word size of 32-bits. This means that all numbers are represented using 32-bits, and that any addition operations will be complete modulo 2^{32}, as there is no storage for carry bits. The basic operation takes place on 512-bit blocks of the message, with eighty rounds of hashing done per block. This will repeat until all the blocks of the original message have been hashed.

The computation of SHA-1 function requires the following:

- Eighty 32-bit words, known has the message schedule, represented as W_0 through W_{79}. Each word will be used in a single round of hashing.
- Five 32-bit working variables, *a*, *b*, *c*, *d*, and *e*. Another temporary 32-bit variable, *T*, is also needed. They will be used to hold intermediate values of the hash during the computation.
- Sixteen 32-bit words to represent the 512-bit message block. They are denoted as $M_0^{(i)}$ through $M_{15}^{(i)}$ where *i* is used to represent the number of the current message block.
- Five 32-bit words used to represent the 160-bit digest. In between message blocks they will hold the intermediate hash value. Similar to the message block, it is represented as $H_0^{(i)}$ through $H_4^{(i)}$, where *i* is the current message block.

2.4.1.1 Initialization

Before hashing can begin, the initialization of the intermediate hashing variables must be completed. This is setting the initial values for $H_0^{(0)}$ through $H_4^{(0)}$ to the ones specified in the standard. The hexadecimal values can be found in Table 2.1.

Table 2.1: SHA-1 Initial Hash Values

$$
\boxed{\begin{aligned}
H_0^{(0)} &= 67452301 \\
H_1^{(0)} &= \text{efcdab89} \\
H_2^{(0)} &= \text{98badcfe} \\
H_3^{(0)} &= 10325476 \\
H_4^{(0)} &= \text{c3d2e1f0}
\end{aligned}}
$$

2.4.1.2 Message Padding

Another important step to be done before hashing occurs is padding the message. It has already been stated that the hash calculation takes place on a single 512-bit block of the message. For this to work out properly in the final block, the message must be padded to a length that is even multiple of 512-bits. The padding scheme is shown in Figure 2.4. The original message first has a single ‘1’ bit appended to it, then many ‘0’ bits, followed by the 64-bit binary representation of the message length, *l*. The number of ‘0’ bits needed can be found by solving the following equation for *k*.

$$\boxed{l + 1 + k = 448 \text{mod} 512}$$

Once determined, the single ‘1’ bit, the *k* ‘0’ bits, and the 64-bit value of *l*, all appended to the original message will result in a new input that is a multiple of the 512-bit block size.

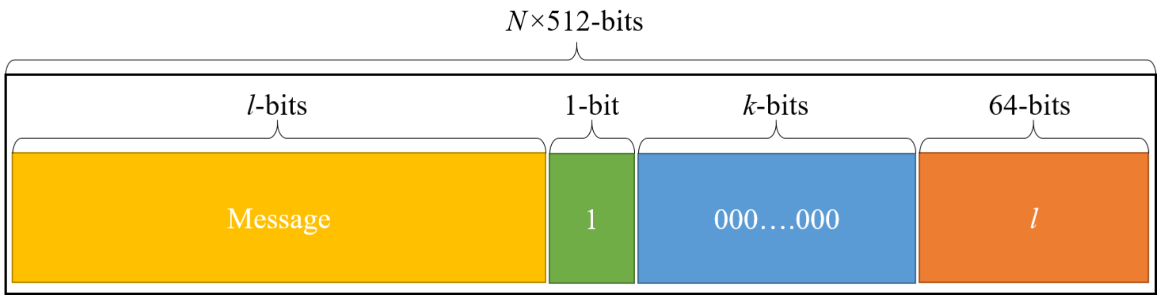

Figure 2.4: Representation of SHA-1 Padding Rule

2.4.1.3 Step 1. – Message Schedule

The first step of the SHA-1 function is to determine the message schedule. The message schedule consists of eighty words, one that will be used in each round of hashing. The first sixteen are directly input from the current message block. The next sixty-four values are a function of previous message schedule values. This function incorporates a *bitwise rotate left* (*ROTL*) by 1 bit and *exclusive-ors* (*XORs*). The formula is shown in Table 2.2.

Table 2.2: SHA-1 Step 1 - Message Schedule

$$W_t = \begin{cases} M_t^{(i)} & 0 \le t \le 15 \\ ROTL^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) & 16 \le t \le 79 \end{cases}$$

2.4.1.4 Step 2. – Update Working Variables

The second step of the algorithm is to simply update the working variables. They are updated to the value of the intermediate hash from the previous message block. In the case of the first message block, the values that were set in the initialization stage will be used. The details of this step is given in Table 2.3.

Table 2.3: SHA-1 Step 2 - Update Working Variables

$$\boxed{\begin{aligned} a &= H_0^{(i-1)} \\ b &= H_1^{(i-1)} \\ c &= H_2^{(i-1)} \\ d &= H_3^{(i-1)} \\ e &= H_4^{(i-1)} \end{aligned}}$$

2.4.1.5 Step 3. – Rounds of Hashing

The third step of SHA-1 is where all the computation actually occurs. In this step the eighty rounds of hashing are completed. The formula is given in Table 2.4. The process starts by setting the temporary variable *T* to the modular addition of five components. The first is a *ROTL* by 5 bits of the working variable *a*. The next is a round dependent function of the variables *b*, *c*, and *d*. Depending on the current round, the function will be either a *choose*, *parity*, or *majority* of the three variables. The third operand of the addition is the working variable *e*, followed by a round constant *K* which is a round dependent constant specified in the standard. The last component is the message schedule of the round. The other parts of each round shift the working variables by 1 word with the exception of *c*, which gets the variable *b ROTL* by 30 bits. Finally, each round is concluded by setting variable *a* to the value stored in temporary variable *T*. The process is completed until all rounds of hashing are done.

Table 2.4: SHA-1 Step 3 - Rounds of Hashing

for $t = 0$ to 79 by 1
{
$T = ROTL^5(a) + f_t(b,c,d) + e + K^t + W^t$
$e = d$
$d = c$
$c = ROTL^{30}(b)$
$b = a$
$a = T$
}

$f_t(x,y,z) = \begin{cases} Ch(x,y,z) = (x \wedge y) \oplus (\overline{x} \wedge z) & 0 \leq t \leq 19 \\ Parity(x,y,z) = x \oplus y \oplus z & 20 \leq t \leq 39 \\ Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) & 40 \leq t \leq 59 \\ Parity(x,y,z) = x \oplus y \oplus z & 50 \leq t \leq 79 \end{cases}$
$K_t = \begin{cases} \text{5a827999} & 0 \leq t \leq 19 \\ \text{6ed9eba1} & 20 \leq t \leq 39 \\ \text{8f1bbcdc} & 40 \leq t \leq 59 \\ \text{ca62c1d6} & 60 \leq t \leq 79 \end{cases}$

2.4.1.6 Step 4. – Update Intermediate Hash

The last step of the SHA-1 algorithm is to update the intermediate hash. This is done by performing modular addition between the previous message block's intermediate hash and the five working variables of the current message block, as shown in Table 2.5. If another message block has yet to be hashed, the entire process from step 1 to 4 will be repeated. If the last message block has been hashed, then the 160-bit digest is available by concatenating the five 32-bit hash variables together.

Table 2.5: SHA-1 Step 4 - Update Intermediate Hash

$$H_0^{(i)} = a + H_0^{(i-1)}$$
$$H_1^{(i)} = b + H_1^{(i-1)}$$
$$H_2^{(i)} = c + H_2^{(i-1)}$$
$$H_3^{(i)} = d + H_3^{(i-1)}$$
$$H_4^{(i)} = e + H_4^{(i-1)}$$

2.4.2 SHA-2

SHA-2 is the second family of hash functions defined by NIST in the secure hash standard, and was published in 2001 [19]. The SHA-2 family contains six functions that provide differing digest sizes and security. The overall function process is very similar to that of SHA-1, and is also based on the Merkle–Damgård construction [24]. The characteristics of the different functions are shown in Table 2.6. The number in the algorithm name specifies the digest length. These six functions are made of only two different algorithms with slight modifications in initialization, and truncation of the output. SHA-224 and SHA-256 follow the exact same process, except the initial hash values that are set. The 256-bit output digest is simply truncated to the first 224-bits to achieve SHA-224. The other functions are the same as the SHA-512 variant, with similar changes made in initial values and truncation of the output.

Table 2.6: Characteristics of SHA-2 Functions

Function	Message Size	Block Size	Word Size	Digest Size
SHA-224	$<2^{64}$ bits	512 bits	32 bits	224 bits
SHA-256	$<2^{64}$ bits	512 bits	32 bits	256 bits
SHA-384	$<2^{128}$ bits	1024 bits	64 bits	384 bits
SHA-512	$<2^{128}$ bits	1024 bits	64 bits	512 bits
SHA-512/224	$<2^{128}$ bits	1024 bits	64 bits	224 bits
SHA-512/256	$<2^{128}$ bits	1024 bits	64 bits	256 bits

2.4.2.1 SHA-256

SHA-256 will produce a 256-bit output digest from any input message with a length smaller than 2^{64} bits. It has a word size of 32-bits and a block size of 512-bits. The number of hashing rounds in this algorithm is sixty-four per message block. This algorithm is identical to SHA-224 with slight changes that are detailed in the appropriate locations.

The following components are required for SHA-256:

- Sixty-four 32-bit words for the message schedule, denoted as W_0 through W_{63}. One word is needed for each round of hashing.
- Eight 32-bit working variables, a, b, c, d, e, f, g, and h. Two temporary variables, T_1 and T_2, are also needed. These will hold intermediate values during each round of the hash computation.
- Sixteen 32-bit words to hold the 512-bit message block. They are defined as $M_0^{(i)}$ through $M_{15}^{(i)}$, using i is to represent the number of the current message block.
- Eight 32-bit variables to represent the 256-bit intermediate and final hash values, $H_0^{(i)}$ through $H_7^{(i)}$, where i is the current message block.

2.4.2.1.1 Initialization

Before the hash calculation can begin, initialization of the intermediate hashing variables must be completed. This is setting the initial values for $H_0^{(0)}$ through $H_7^{(0)}$ to the ones specified in the standard. The values that are set depend on the function used. If implementing SHA-224, this is the first change to the algorithm. The hexadecimal values for both functions can be found in Table 2.7.

Table 2.7: SHA-256 Initial Hash Values

SHA-256	SHA-224
$H_0^{(0)} = \text{6a09e667}$	$H_0^{(0)} = \text{c1059ed8}$
$H_1^{(0)} = \text{bb67ae85}$	$H_1^{(0)} = \text{367cd507}$
$H_2^{(0)} = \text{3c6ef372}$	$H_2^{(0)} = \text{3070dd17}$
$H_3^{(0)} = \text{a54ff53a}$	$H_3^{(0)} = \text{f70e5939}$
$H_4^{(0)} = \text{510e527f}$	$H_4^{(0)} = \text{ffc00b31}$
$H_5^{(0)} = \text{9b05688c}$	$H_5^{(0)} = \text{68581511}$
$H_6^{(0)} = \text{1f83d9ab}$	$H_6^{(0)} = \text{64f98fa7}$
$H_7^{(0)} = \text{5be0cd19}$	$H_7^{(0)} = \text{befa4fa4}$

2.4.2.1.2 Message Padding

The block size and word size of the SHA-256 algorithm are the same as SHA-1. The padding scheme is also identical to the previous hash family. This means that the entire message padding process is equivalent to that of SHA-1. See section 2.4.1.2 for the detailed description.

2.4.2.1.3 Step 1. – Message Schedule

Once again, deriving the message schedule is the first step of the algorithm. In SHA-256, this consists of sixty-four words. The first sixteen are directly input from the message block. The next forty-eight are calculated using a function of four previous message schedule words. This function incorporates modular addition and two other functions, denoted as σ_0 and σ_1. Each of these functions include *bitwise rotate rights* (*ROTRs*), *bitwise right shifts* (*SHR*), and *XORs*. The number of shift and rotate bits differ between the functions. The step outline is shown in Table 2.8.

Table 2.8: SHA-256 Step 1- Messasge Schedule

$$W_t = \begin{cases} M_t^{(i)} & 0 \le t \le 15 \\ \sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16} & 16 \le t \le 63 \end{cases}$$

$$\sigma_0^{\{256\}}(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x)$$
$$\sigma_1^{\{256\}}(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$$

2.4.2.1.4 Step 2. – Update Working Variables

Step two of SHA-256 is updating the working variables. They are set to the value of the intermediate hash from the previous message block. In the case of the first message block, the values that were set in the initialization stage will be used. Again, these values will differ between the SHA-256 and SHA-224 functions. The step is detailed in Table 2.9.

Table 2.9: SHA-256 Step 2 - Update Working Variables

$$\begin{aligned} a &= H_0^{(i-1)} \\ b &= H_1^{(i-1)} \\ c &= H_2^{(i-1)} \\ d &= H_3^{(i-1)} \\ e &= H_4^{(i-1)} \\ f &= H_5^{(i-1)} \\ g &= H_6^{(i-1)} \\ h &= H_7^{(i-1)} \end{aligned}$$

2.4.2.1.5 Step 3. – Rounds of Hashing

The third step in the algorithm is the sixty-four rounds of hashing. This step contains the majority of computation required to calculate a message digest. The first step is calculating the two temporary variables, T_1 and T_2. The value of T_1 is a modular addition of variable *h*, the message schedule for that round, W_t, and the round constant, K_t. There is also two operands to the sum that are the result of different functions, one being a *choose* function with inputs *e, f,* and *g,* while the other is the function $\sum_1$ with input *e*. $\sum_1$ returns the result of *XORing* together three *ROTRs* of the input. The rotate factors are 6-bits, 11-bits, and 25-bits, respectively. To calculate T_2, it is the sum of a *majority* function with the inputs *a*, *b*, and *c*, and $\sum_0$ with the input *a*. $\sum_0$ is the same as $\sum_1$ but with rotate factors of 2-bits, 13-bits, and 22-bits instead. The rest of the round shifts the working variables by one word, with the exception of variable *e*, which gets the addition of *d* and T_1, and variable *a,* which gets the value of summing T_1 and T_2. This step repeats until all sixty-four rounds are completed.

Table 2.10: SHA-256 Step 3 - Rounds of Hashing

for $t = 0$ to 63 by 1
 {
 $T_1 = h + \sum_1^{\{256\}}(e) + Ch(e,f,g) + K_t^{\{256\}} + W_t$
 $T_2 = \sum_0^{\{256\}}(a) + Maj(a,b,c)$
 $h = g$
 $g = f$
 $f = e$
 $e = d + T_1$
 $d = c$
 $c = b$
 $b = a$
 $a = T_1 + T_2$
 }

$Ch(x,y,z) = (x \wedge y) \oplus (\overline{x} \wedge z)$ $Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$
$\sum_0^{\{256\}}(x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x)$ $\sum_1^{\{256\}}(x) = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x)$
Values for $K_t^{\{256\}}$ can be found in the secure has standard documentation [19].

2.4.2.1.6 Step 4. – Update Intermediate Hash

Finally, the intermediate hash variables are updated to the sum of their previous value and the corresponding working variable, as shown in Table 2.11. The entire algorithm is repeated until all the message blocks have been processed. Once that has happened, the 256-bit output is given by concatenating all eight intermediate hash variables together. If the function is SHA-224, that 256-bit digest is truncated to 224-bits.

Table 2.11: SHA-256 Step 4 - Update Intermediate Hash

$$\boxed{\begin{aligned} H_0^{(i)} &= a + H_0^{(i-1)} \\ H_1^{(i)} &= b + H_1^{(i-1)} \\ H_2^{(i)} &= c + H_2^{(i-1)} \\ H_3^{(i)} &= d + H_3^{(i-1)} \\ H_4^{(i)} &= e + H_4^{(i-1)} \\ H_5^{(i)} &= f + H_5^{(i-1)} \\ H_6^{(i)} &= g + H_6^{(i-1)} \\ H_7^{(i)} &= h + H_7^{(i-1)} \end{aligned}}$$

2.4.2.2 SHA-512

The SHA-512 function produces a 512-bit digest of the input message. In this case the size of the file must be less than 2^{128} bits. The algorithm uses a word size of 64 bits with a block size of 1024 bits. SHA-512 computes eighty rounds of hashing on each message block. Other functions of the SHA-2 family use the same algorithm as SHA-512, with slight modifications. These functions are SHA-384, SHA-512/224, and SHA-512/256. The changes needed to implement these functions are descripted where required.

The computation of the SHA-512 function needs the following components:

- Eighty 64-bit words for the message schedule, one for each round of hashing, represented as W_0 through W_{79}.

- Eight 64-bit working variables, a, b, c, d, e, f, g, and h. Two temporary variables, T_1 and T_2, are also needed. They will be used to hold intermediate values during each round of the hash computation.
- Sixteen 64-bit words to hold the 1024-bit message block. They are denoted as $M_0^{(i)}$ through $M_{15}^{(i)}$, using *i* is to represent the number of the current message block.
- Eight 64-bit variables to represent the 512-bit intermediate and final hash values, $H_0^{(i)}$ through $H_7^{(i)}$, where *i* is the current message block.

2.4.2.2.1 Initialization

In a similar fashion to SHA-256, the intermediate hash variables must be initialized to values specified in the standard before the computation can begin. These values are different for each variant that follows the same algorithm. There are defined values for SHA-512, SHA-384, SHA-512/224, as well as SHA-512/256. The values in hexadecimal form for $H_0^{(0)}$ through $H_7^{(0)}$ are shown in Table 2.12.

Table 2.12: SHA-512 Initial Hash Values

SHA-512	SHA-384
$H_0^{(0)}$ = 6a09e667f3bcc908 $H_1^{(0)}$ = bb67ae8584caa73b $H_2^{(0)}$ = 3c6ef372fe94f82b $H_3^{(0)}$ = a54ff53a5f1d36f1 $H_4^{(0)}$ = 510e527fade682d1 $H_5^{(0)}$ = 9b05688c2b3e6c1f $H_6^{(0)}$ = 1f83d9abfb41bd6b $H_7^{(0)}$ = 5be0cd19137e2179	$H_0^{(0)}$ = cbbb9d5dc1059ed8 $H_1^{(0)}$ = 629a292a367cd507 $H_2^{(0)}$ = 9159015a3070dd17 $H_3^{(0)}$ = 152fecd8f70e5939 $H_4^{(0)}$ = 67332667ffc00b31 $H_5^{(0)}$ = 8eb44a8768581511 $H_6^{(0)}$ = db0c2e0d64f98fa7 $H_7^{(0)}$ = 47b5481dbefa4fa4
SHA-512/224	**SHA-512/256**
$H_0^{(0)}$ = 8c3d37c819544da2 $H_1^{(0)}$ = 73e1996689dcd4d6 $H_2^{(0)}$ = 1dfab7ae32ff9c82 $H_3^{(0)}$ = 679dd514582f9fcf $H_4^{(0)}$ = 0f6d2b697bd44da8 $H_5^{(0)}$ = 77e36f7304c48942 $H_6^{(0)}$ = 3f9d85a86a1d36c8 $H_7^{(0)}$ = 1112e6ad91d692a1	$H_0^{(0)}$ = 22312194fc2bf72c $H_1^{(0)}$ = 9f555fa3c84c64c2 $H_2^{(0)}$ = 2393b86b6f53b151 $H_3^{(0)}$ = 963877195940eabd $H_4^{(0)}$ = 96283ee2a88effe3 $H_5^{(0)}$ = be5e1e2553863992 $H_6^{(0)}$ = 2b0199fc2c85b8aa $H_7^{(0)}$ = 0eb72ddc81c52ca2

2.4.2.2.2 Message Padding

The padding scheme for SHA-512 is similar to the one used in SHA-1 and SHA-256, however, the word size and block size have changed. This means that the process of padding the message is a bit different than the other algorithms. The message must know be an even multiple of 1024-bits before hashing can occur. Again, a single ‘1’ bit is appended to the message followed by a number of ‘0’ bits. Then the binary representation of the message length, *l*, using 128-bits is concatenated to the end. The number of '0' bits needed can be determine by solving the equation for the value of *k*.

$$\boxed{l + 1 + k = 896\text{mod}1024}$$

The message with a single '1' bit, *k* number of '0' bits, and the 128-bit binary representation of *l*, will produced a padded message with a total length that is an even multiple of 1024-bits.

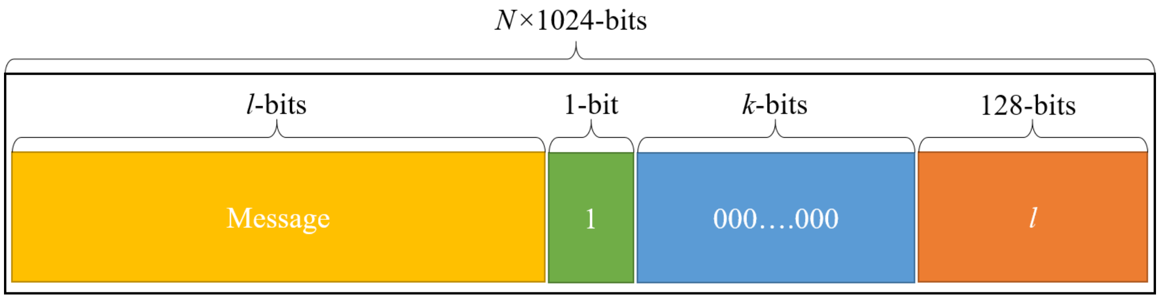

Figure 2.5: Representation of SHA-512 Padding Rule

2.4.2.2.3 Step 1. – Message Schedule

The message schedule derivation is the first step the SHA-512 algorithm. The message schedule in this function consists of eighty words that correlate to the eighty rounds of hashing. The first sixteen words of the message schedule come from the 1024-bit message block. The rest are then a function of previous words that consists of modular addition and two other functions, denoted as σ_0 and σ_1. Each of these functions include *ROTRs*, *SHRs*, and *XORs*. The number of shift and rotate bits differ between the functions. The step outline is shown in Table 2.13.

Table 2.13: SHA-512 Step 1- Message Schedule

$W_t = \begin{cases} M_t^{(i)} & 0 \le t \le 15 \\ \sigma_1^{\{512\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{512\}}(W_{t-15}) + W_{t-16} & 16 \le t \le 79 \end{cases}$
$\sigma_0^{\{512\}}(x) = ROTR^1(x) \oplus ROTR^8(x) \oplus SHR^7(x)$ $\sigma_1^{\{512\}}(x) = ROTR^{19}(x) \oplus ROTR^{61}(x) \oplus SHR^6(x)$

2.4.2.2.4 Step 2. – Update Working Variables

The second step of SHA-512 is to update the working variables to the value of the intermediate hash of the last message block, or the initial values set in the case of the first message block. These initial values are different for each function, and shown in Table 2.12. The process of this step is identical to step 2 of SHA-256, with the exception of using 64-bit words instead of 32-bit. It is outlined in Table 2.9.

2.4.2.2.5 Step 3. – Rounds of Hashing

Step three contains the eighty rounds of hashing for each message block. Like the previously described SHA functions, this is the where the bulk of the computation is completed. The process is very similar to step three of SHA-256, described in section 2.4.2.1.5, with a few small differences aside from the word and block sizes. The first difference is that the number of rounds has increased from sixty-four in SHA-256 to eighty in SHA-512. The round constants, K_t, which can be found in the standard [19], also differ between the algorithms. The last difference involves the functions of $\sum_0$ and $\sum_1$. The rotate factors for both functions have been modified. For $\sum_0$, the three rotate factors are now 28-bits, 34-bits, and 39-bits. In $\sum_1$, the rotate factors have become 14-

bits, 18-bits, and 41-bits. Taking these changes into account, the process is the same as SHA-256. The progression of this step is given in Table 2.14.

Table 2.14: SHA-512 Step 3 - Rounds of Hashing

for $t = 0$ to 79 by 1
{
$T_1 = h + \sum_1^{\{512\}}(e) + Ch(e,f,g) + K_t^{\{512\}} + W_t$
$T_2 = \sum_0^{\{512\}}(a) + Maj(a,b,c)$
$h = g$
$g = f$
$f = e$
$e = d + T_1$
$d = c$
$c = b$
$b = a$
$a = T_1 + T_2$
}

$Ch(x,y,z) = (x \wedge y) \oplus (\overline{x} \wedge z)$ $Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$
$\sum_0^{\{512\}}(x) = ROTR^{28}(x) \oplus ROTR^{34}(x) \oplus ROTR^{39}(x)$ $\sum_1^{\{512\}}(x) = ROTR^{14}(x) \oplus ROTR^{18}(x) \oplus ROTR^{41}(x)$
Values for $K_t^{\{512\}}$ can be found in the secure has standard documentation [19].

2.4.2.2.6 Step 4. – Update Intermediate Hash

The last step to be done on each message block of SHA-512 is updating the intermediate hash variables. This process is also identical, save for the word size, to that of SHA-256, and shown in Table 2.11. Once all the message blocks are hashed, the 512-bit digest is available. For the other SHA-2 functions that are implemented with this algorithm, the digest can then be truncated to the proper length.

2.4.3 SHA-3

The third family of SHAs standardized by NIST is SHA-3 [20]. It was finalized into a published standard in August 2015. The goal of this family is not to replace SHA-2, but supplement it with alternative algorithms. When choosing the algorithm to use in the standard, NIST held a contest for various functions. The winner of that contest was the Keccak family of sponge functions which has now been used to implement the SHA-3 algorithm [25]. The Keccak sponge function has a much different structure compared to the Merkle–Damgård construction of SHA-1 and SHA-2.

SHA-3 contains six hashing functions. Four of these have fixed size output digests that are the same size as the SHA-2 functions. The other two functions are extendable-output functions (XOFs) that allow for any arbitrary digest length. The name of the XOFs specify the maximum security bits available to the function, rather than the digest length like the other algorithms. All the functions use the same process with differing block sizes. There is no input message size limit to any of the SHA-3 functions. During computation, twenty-four rounds of hashing are completed on each message block. The word size is 64-bits, while the block size is a function of the state size and the number of security bits. The characteristics of all SHA-3 functions are given in Table 2.15.

Table 2.15: Characteristics of SHA-3 Functions

Function	Digest Size	Block Size	Security
SHA3-224	224 bits	1152 bits	112 bits
SHA3-256	256 bits	1088 bits	128 bits
SHA3-384	384 bits	832 bits	192 bits
SHA3-512	512 bits	576 bits	256 bits
SHAKE128	*d* bits	1088 bits	min(*d*/2, 128) bits
SHAKE256	*d* bits	576 bits	min(*d*/2, 256) bits

A sponge function is simply a hashing function that can take any arbitrary length input function and produce an arbitrary length output [26]. The function operates on a *state* of bits in two phases. The first phase is the absorbing phase where a message block is *XORed* into the state, and a particular hashing function is executed. When all the message blocks have been absorbed, then the second phase can be done. This is the squeezing phase in which the appropriate number of bits are returned from the state, producing the output. This process is visualized in Figure 2.6. Keccak uses this sponge function architecture with the hashing function of Keccak-*f* in the SHA-3 standard.

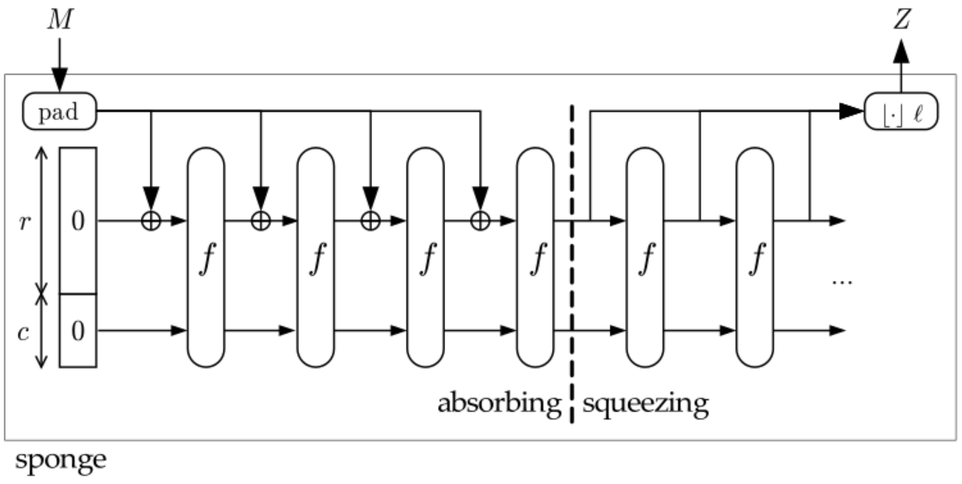

Figure 2.6: Sponge Function Construction [26]

The SHA-3 algorithm operates with a *state* size of 1600 bits, configured into a three dimensional array. The dimensions of this array are 5-bits, by 5-bits, by 64-bits. The *state*, and its components can be visualized as shown in Figure 2.7. Different operations of the algorithm focus on different structures of the *state*, whether it be a plane, slice, column, etc. The coordinate convention of the *state* is specified in the *x*, *y*, and *z* planes. Both *x* and *y* have a range from 0 to 4, while z has a range of 0 to 63. A visualization of the indexing is shown in Figure 2.8.

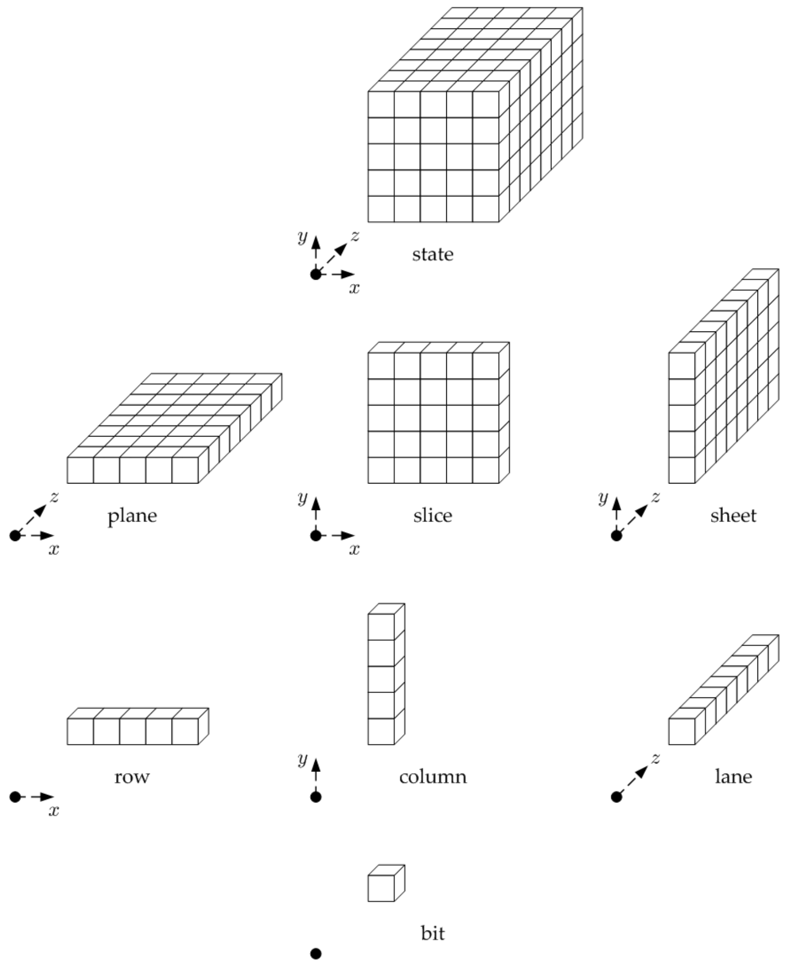

Figure 2.7: SHA-3 State and Its Components [25]

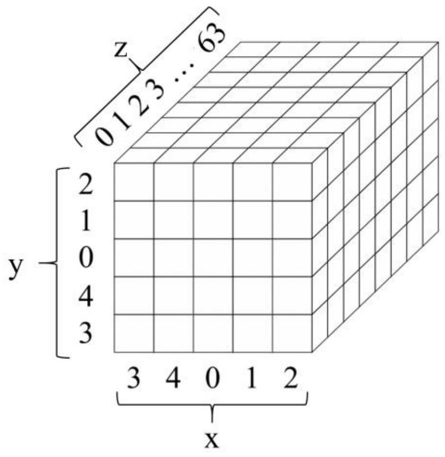

Figure 2.8: Coordinate Convention of State. Adapted from [20]

The Keccak sponge function consists of three main components. The first is the hashing function that will be performed on the state. The next component is the padding rule, to ensure that the input will contain an even number of message blocks, each with a length equal to the block size of the algorithm. The third component of the sponge function is the block size, and denoted as *r*. There are also two inputs needed, the message to be hashed, *N*, and the digest size, *d*. The sponge function will first pad the input message using the padding rule so that the length is an even multiple of the block size. Then the message block is *XORed* into the *state*. The hashing function is then operated on the *state*. This repeats for each message block until the entire message has been processed. Then the digest can be extracted from the *state*. If the amount of data in the *state* is smaller than the requested digest size, then the hash function is operated on the *state* again. This step is repeated until there is enough data to construct the digest.

$$\boxed{\underline{\mathrm{Z} = \mathrm{SPONGE}[f, \mathrm{pad}, r](\mathrm{N}, \mathrm{d})}}$$

The padding rule used in SHA-3 is pad10*1. This padding rule works by appending a single ‘1’ bit to the message, followed by an appropriate number of ‘0’ bits. Then the final ‘1’ bit is attached to the end. The number of ‘0’ bits can be determined using the formula below, solving for *k*, to produce a final message that is a multiple of the block size.

$$\boxed{k + 2 = r - l\,\mathrm{mod}(\mathrm{r})}$$

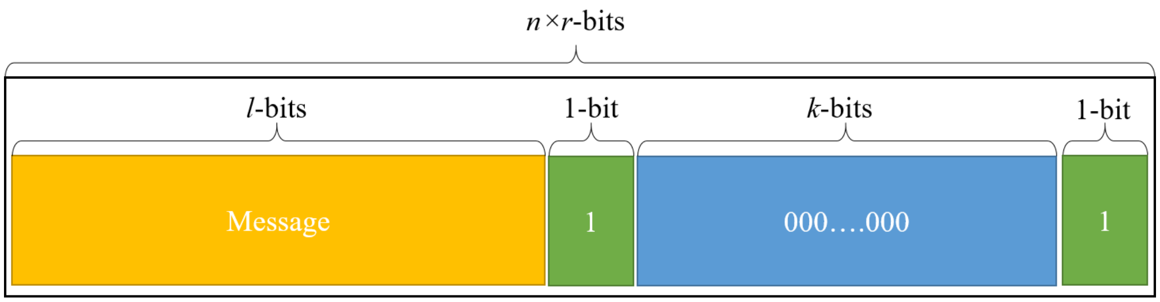

Figure 2.9: Representation of SHA-3 Padding Rule

The hashing function that is used in SHA-3 is Keccak-*f* with a *state* size of 1600 bits. It contains twenty-four hashing rounds each with five step mappings. Each step mapping provides a different operation on the *state*. The formula is given below of the Keccak-*f* function. A description of each of the step mappings can be found starting at section 2.4.3.1.

```
Keccak-$f[1600](A)$ =
for $ROUND$ from 0 to 23:
{
        $A = \iota\left(\chi\left(\pi\left(\rho\left(\theta(A)\right)\right)\right), \mathrm{ir}\right)$
}
```

The final representation of the Keccak sponge function that is used in the SHA-3 standard is given as below. Each function in the standard is implemented by setting the component *c* to the proper value, as shown in Table 2.16. In the four fixed length digest functions there are two bit ‘01’ appended to the message before it is input to the Keccak function. In the XOFs, it is four bits, ‘1111’. This is used to differentiate the digest between the functions if the XOF was set to the same length as the fixed length functions.

$$\boxed{\underline{\mathrm{KECCAK}[c](N, d) = \mathrm{SPONGE}[\mathrm{Keccak}\text{-}f[1600], \mathrm{pad10*1}, 1600 - c](N, d)}}$$

Table 2.16: SHA-3 Function Definitions

SHA-3 Function	Keccak Sponge Function
SHA3-224(M)	KECCAK[448](M||01,224)
SHA3-256(M)	KECCAK[512](M||01,256)
SHA3-384(M)	KECCAK[768](M||01,384)
SHA3-512(M)	KECCAK[1024](M||01,512)
SHAKE128(M, d)	KECCAK[256](M||1111, d)
SHAKE256(M, d)	KECCAK[512](M||1111, d)

2.4.3.1 Theta Step Mapping

The first step mapping is θ. It *XORs* each bit in the *state* with the parity of the column ahead and the column behind it with respect to the *x*-plane. The process is shown in Table 2.17 where A is the *state*. The illustration of the operation is given in Figure 2.10.

Table 2.17: SHA-3 Theta Step Mapping

```
$\theta(A)$:
$C[x,z] = A[x,0,z] \oplus A[x,1,z] \oplus A[x,2,z] \oplus A[x,3,z]$
                $\oplus A[x,4,z]$ $for\ all\ pairs\ (x,z)$
$D[x,z] = C[(x-1)mod5, z] \oplus C[(x+1)mod5, (z-1)mod64]$ $for\ all\ pairs\ (x,z)$
$A'[x,y,z] = A[x,y,z] \oplus D[x,z]$
$Return\ A'$
```

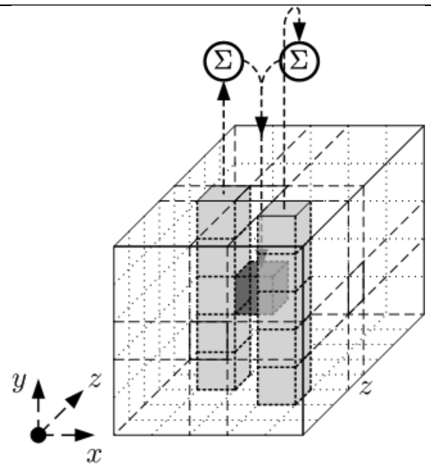

Figure 2.10: SHA-3 Theta Step Mapping [25]

2.4.3.2 Rho Step Mapping

The next step mapping is ρ. It rotates each lane by an offset that is dependent on the (x, y) coordinated of that lane. The formula used to calculate the offsets, as well as the calculated offsets for SHA-3 are given in Table 2.18.

Table 2.18: SHA-3 Rho Step Mapping and Calculated Offsets

```
$\rho(A)$:
$Let\ (x,y) = (1,0)$
$For\ t\ from\ 0\ to\ 23$
{
$A'[x,y,z] = A[x,y,(z-(t+1)(t+2)/2) mod\ 64]$
$Let\ (x,y) = (y,(2x+3y) mod 5)$
}
$Return\ A'$
```

Offset per lane in bits:

Y\X	3	4	0	1	2
2	25	39	3	10	43
1	55	20	36	44	6
0	28	27	0	1	62
4	56	14	18	2	61
3	21	72	41	45	15

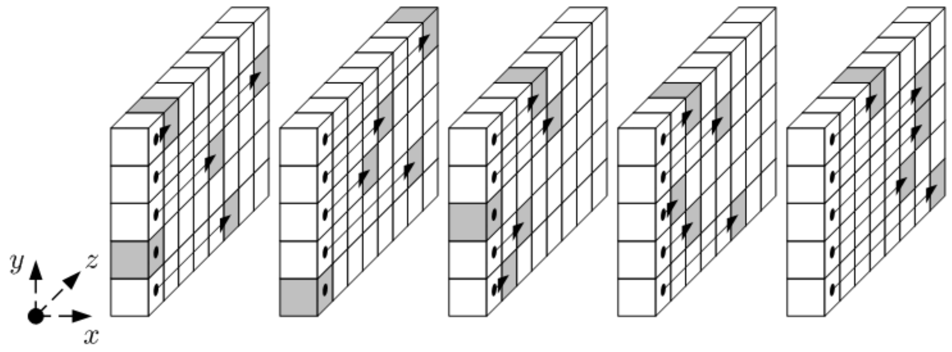

Figure 2.11: SHA-3 Rho Step Mapping [25]

2.4.3.3 Pi Step Mapping

The third step mapping is π. The effect of this step mapping is to rearrange the position of each lane in the *state*. Table 2.19 shows the π step mapping function, while Figure 2.12 gives a depiction of the operation on the *state*.

Table 2.19: SHA-3 Pi Step Mapping

$\pi(A)$:
$A'[x,y,z] = A[(x+3y)mod5, x, z]$ $for\ triples\ (x,y,z)$
$Return\ A'$

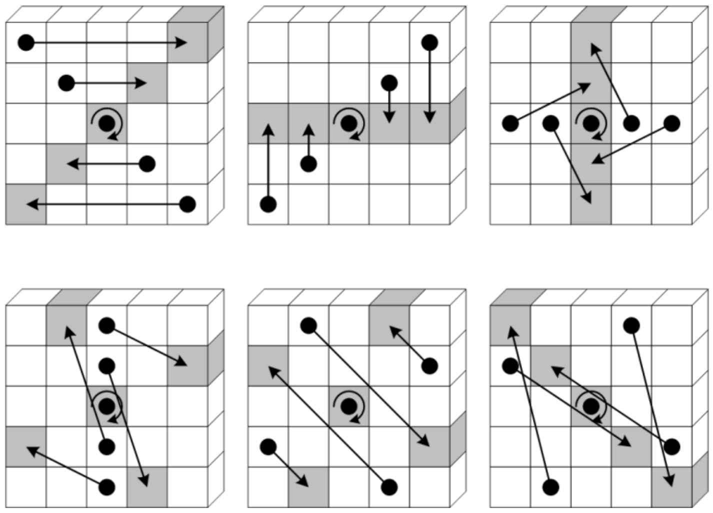

Figure 2.12: SHA-3 Pi Step Mapping [25]

2.4.3.4 Chi Step Mapping

The χ step mapping *XORs* each bit in the *state* with a non-linear function of the next two bits in its row, as given in Table 2.20.

Table 2.20: SHA-3 Chi Step Mapping

$\chi(A)$:
$A'[x,y,z] = A[x,y,z] \oplus \left((A[(x+1)mod5,y,z] \oplus 1) \wedge A[(x+2)mod5,y,z]\right)$
$Return\ A'$

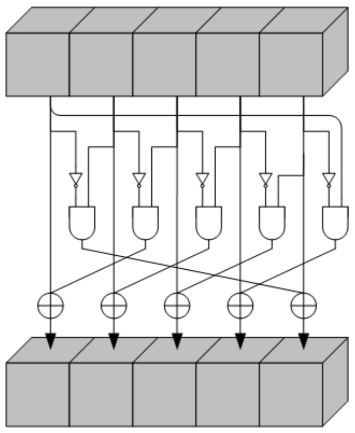

Figure 2.13: SHA-3 Chi Step Mapping [25]

2.4.3.5 Iota Step Mapping

The *ι* step mapping operation is round dependent, and requires the round index, i_r, to be set as an argument to the function. The effect is to *XOR* only the first lane by a round constant. The process is give in Table 2.21, along with the values of the round constants in hexadecimal form.

Table 2.21: SHA-3 Iota Step Mapping and Round Constant Values

```
$\iota(A, i_r)$:
$A'[x, y, z] = A[x, y, z]\ for\ all\ triples\ (x, y, z)$
 $A'[0,0, z] = A'[0,0, z] \oplus RC[i_r]$
$Return\ A'$
```

```
$RC[24] = \{$
 0000000000000001, 0000000000008082, 800000000000808a,
 8000000080008000, 000000000000808b, 0000000080000001,
 8000000080008081, 8000000000008009, 000000000000008a,
 0000000000000088, 0000000080008009, 000000008000000a,
 000000008000808b, 800000000000008b, 8000000000008089,
 8000000000008003, 8000000000008002, 8000000000000080,
 000000000000800a, 800000008000000a, 8000000080008081,
 8000000000008080, 0000000080000001, 8000000080008008
 $\}$
```

2.5 Related Work

SHAs have poor speed performance on CPU architectures [3]. Due to this there has been much research done using hardware implementations, specifically FPGAs, to accelerate these algorithms.

2.5.1 SHA-1

In 2003, Sklavos *et al.* worked to implement the SHA-1 function on an FPGA [27]. The model was designed using VHDL and synthesized on a Xilinx 2V500FG456 FPGA. The design implemented a pipelined structure of the algorithm. The throughput of the proposed design was compared to other work, including other FPGA implementations and CPU implementations coded in assembly. The result was a throughput of 1339 Mbps, which was a 140% speedup compared to previous FPGA results, and about 1125% speedup compared to the CPU result. This work also studied the efficiency of each design, providing a comparison of a performance per unit area of the designs. The proposed design was much more efficient than the CPU versions, but had only marginally better efficiency than the other FPGA designs.

Kakarountas *et al.* improved the performance of the SHA-1 function even further in 2005 [28]. The model was implemented on a Xilinx V150BG352 FPGA using an HDL design methodology. A pipeline structure was used here to increase the speed. The critical path of the calculation was shortened which helped improve performance. This was done by including a pre-computational stage to the pipeline. That allowed the calculation of some of the variables to be done before the round that they are needed in, provided they are independent of other variables. This pre-computation stage is pipelined with the regular calculation stage, giving an overall shortened pipeline, and increasing speed. The results obtained was a throughput of 2526 Mbps, a roughly 37% increase over comparable designs. The area needed was only slightly more than other designs as well.

In another approach in 2006, Yiakoumis *et al.* produced a VHDL model of a key-hashed message authenticate code (HMAC), which is a direct application of the SHA-1 algorithm [29]. The design was synthesized on a Xilinx Virtix-II FPGA. The goal was to design a HMAC with a high throughput but a small size. In order to keep the size to a minimum, pipelining could not be used. Relying on a pre-computational stage, and replacing hardware adders with faster variants, a result throughput of 1024 Mbps was achieved. This result is about 17% faster than comparable HMAC designs. It was also the first SHA-1 implementation to reach 1 Gbps throughput without the use of pipelining.

Lee *et al.* worked to implement an SHA-1 module in 2009 with a two-unfolded architecture [30]. In this technique there are two hashing cores that work in parallel. One handles the pre-computation, while the other handles the current hash cycle. There is no dependency between the two cores in a single step. Combining this technique with pipelining structures produced a very high throughput. The result speed that was achieved was 6040 Mbps on a Xilinx Virtex-II XC2V1000 FPGA. Comparable designs using similar techniques were 26% slower in their hash computation. The area was also decreased from similar designs as well. The proposed model was 32% smaller than previous work.

2.5.2 SHA-2

Sklavos *et al.* implemented SHA-256, SHA-384, and SHA-512 designs in 2003 on a Xilinx V200PQ240 FPGA [31]. The model was specified in VHDL. The goal of the paper was to implement the SHA-2 standard and compare to the SHA-1, as it was a new standard at the time. The results that achieved were 326 Mbps for SHA-256, 350 Mbps for SHA-384, and 480 Mbps for SHA-512. At this time, the performance was better than the related work in SHA-1 and other SHA-2 designs, while having a comparable area.

In 2008, Khalil *et al.* worked on the design of the SHA-2 standard core in FPGA for the application of digital signatures on a SoC [32]. The design was implemented on an Altera Stratix EP1S40F780C5 FPGA using Verilog. The architecture of the design contained a message scheduler unit and an iteration processing unit. The algorithm was split into these processors to try to obtain the best possible speedup by running concurrent operations. The results of this work include a 464 Mbps throughput for a SHA-224/256 core, and a 644 Mbps throughput for a SHA-384/512 core. This was a very modest improvement over the comparable work.

Michail *et al.* produced a high throughput SHA-256 model in 2010 [33]. The VHDL design was synthesized on Xilinx Virtex, Virtex-II, and Virtex-E FPGA technologies. The architecture included a pre-computation unit and a pipelined structure. Other optimizations to the design were done to reduce the critical path of the design, such as replacing full adders with carry save adders to speed up the calculation. The results of the design were impressive, with 2077 Mbps on the Virtex FPGA, 3100 Mbps on the Virtex-II, and 2190 Mbps on the Virtex-E. The variations in throughput can be attributed to performance of the FPGAs themselves. The proposed design was faster than comparable designs by differing amounts for each FPGA. The Virtex and Virtex-II speed was at least doubled compared to the next fastest on their platforms, while the Virtex-E implementation only had a marginal speedup. The size of the designs were very similar to that of the related work.

A SHA-256 and SHA-512 design was made by Mestiri *et al.* in 2014 [34]. It was written in VHDL and synthesized on a Xilinx Virtex-5 XC5VFX70T FPGA. The module that was created can be used to calculate both the SHA-256 and SHA-512 digest that consists of a padding processor, a round computation processor, and a controller to implement the proper function. This design produced a throughput of 1617 Mbps for SHA-256 and 2252 Mbps for SHA-512. The required area of the design was reduced from comparable work, resulting in a very efficient implementation.

2.5.3 SHA-3

In 2010, Baldwin *et al.* implemented all of the second round candidates of the SHA-3 competition in FPGA hardware [35]. The goal was to characterize all the potential functions that could be used to implement the SHA-3 standard. Each function was modeled using VHDL and then synthesized on a Xilinx Virtex-5 FPGA. The four fixed length digest functions were implemented and compared. Keccak was among the top in performance with a throughput of 5915 Mbps on SHA3-224, 6232 Mbps on SHA3-256, 8190 Mbps on SHA3-384, and 8518 Mbps on SHA3-512. The size of Keccak was the second smallest of the fourteen candidates, but was the highest in efficiency.

Jungk and Apfelbeck took a similar approach to Baldwin *et al.* in 2011, this time implementing the SHA-3 competition finalists [36]. Using HDL design techniques, the target FPGA was the Xilinx Virtex-6. Of the five finalists, Keccak was second in terms of speed and efficiency, but took the most area of all the designs. The throughput that was achieved was 864 Mbps. This was only completed on the SHA-256 variant.

The finalists of the SHA-3 competition were also implemented in FPGA by Song *et al.* in 2011 [37]. They used Verilog to design the modules, and then synthesized them on a Xilinx Virtex-5 FPGA. The area of the Keccak functions were very similar to that of the other candidates. The throughput that was achieved on SHA3-256 was 14438 Mbps, and on SHA-512 the throughput was 7066 Mbps. A reconfigurable module to compute either variant was also implemented that gave a throughput of 13414 Mbps with a slight increase in area needed.

2.6 Summary

The background information of FPGAs, HLS, heterogeneous computing systems, OpenCL and the AOCL CAD tool was presented in this chapter. The cryptographic hash functions specified in the secure hash standard were also described in detail. Finally a literature review of published research using FPGAs to accelerate those functions was presented. There is much related work in this area, with a wide range of speed performance results. All of the designs that were reviewed used HDL based design methodologies. To our knowledge, there has been no published work using any HLS design tool to accelerate the SHA functions. This thesis evaluates the ability of the AOCL HLS CAD tool to accelerate this application. The next chapter will focus on the implementation, synthesis, and evaluation of the SHA FPGA accelerated program.

Chapter 3 Synthesis and Evaluation

This chapter focuses on the implementation, synthesis, and evaluation of the FPGA accelerated SHA algorithms. The design of each function is described in detail before the experimental results are given. There is some design space exploration (DSE) of algorithms done to obtain the best possible synthesis results using AOCL. The results show the relationship between input message size and throughput, maximum throughput, and a comparison to the CPU application. Finally, the throughput and FPGA area utilization for each of the algorithms is compared to published literature to determine the competitiveness of the HLS approach to the HDL based design methodology.

3.1 Overview

When using the OpenCL model to accelerate a program, the process begins with implementing the application with traditional CPU programming. Once the process is functional, the program can be modified into an OpenCL model. Then, optimization can be done on the OpenCL model to achieve the best performance. This allows for a verification of the FPGA design as well as a comparison between the CPU and the FPGA implementations.

Typical OpenCL programs have many independent calculations that can be completed in parallel. Running all these calculations simultaneously increases the performance of the application. The secure hash functions do not have this type of structure, since each iteration of the algorithm depends on the calculation of the step before it. There are also dependencies between variables in each iteration. With most OpenCL devices, the only way to accelerate the SHA functions would be using batch parallelization, which would calculate numerous digests simultaneously. This is used in some brute force password cracking attacks. But with the reconfigurable hardware of FPGA, it is still possible to accelerate the calculation of a single digest.

3.2 SHA-1

The SHA-1 algorithm was the first to be implemented. The CPU model was first programmed in C++. It was made by following the description in the secure hash standard [19].

3.2.1 Host Program

Once the C++ program is functioning correctly, the OpenCL model can be implemented. The padding of the input message is still handled on the CPU, as the data must be properly organized before it sent to the kernel. There is not much time lost here by padding the message as the data is organized. The host program sets up the OpenCL model as described in section 2.3.1.1. In this application, two memory buffers to transfer data to and from the OpenCL device are needed. The first buffer is setup as read only by the kernel, and it contains the entirety of the input message. The second buffer is created to be able to transfer the digest calculated in the kernel to the host program. Along with these buffers, there is another argument to the kernel, the number of message blocks required to make up the entire input message. The host program can then launch the kernel to execute the digest calculation. The kernel is launched as a single work-item as the acceleration is a result of the hardware, not parallel kernel instances. Once the computation is complete, the output can be read back from the hash buffer.

3.2.2 Kernel

In the kernel code, the main kernel function as well as custom functions are defined. The main kernel function is of type *void* as it returns no value. The custom functions that are needed in the SHA-1 algorithm include *ROTL*, *choose*, *parity*, and *majority*. There are also functions that returns the proper constant, and function for each round in hashing. All of these functions return a single 32-bit unsigned word and are presented in Figure 3.1.

```
unsigned ROTL( int n, unsigned x){
    return ((x << n)) | ((x) >> (32-n));
}

unsigned choose (unsigned x, unsigned y, unsigned z){
    return (((x&y)^((~x)&z)));
}

unsigned parity (unsigned x, unsigned y, unsigned z){
    return ((x^y^z));
}

unsigned majority (unsigned x, unsigned y, unsigned z){
    return (((x&y)^(x&z)^(y&z)));
}

unsigned funct(unsigned x, unsigned y, unsigned z, int t){
    if ((0 <= t) && (t <= 19)){
        return choose(x,y,z);
    }
    else if (((20 <= t) && (t <=39)) || ((60 <= t) && (t <= 79))){
        return parity(x,y,z);
    }
    else {
        return majority(x,y,z);
    }
}
unsigned getK(int t){
    if ((0 <= t) && (t <= 19)){
        return 0x5a827999;
    }
    else if ((20 <= t) && (t <=39)){
        return 0x6ed9eba1;
    }
    else if ((40 <= t) && (t <=59)){
        return 0x8f1bbcdc;
    }
    else {
        return 0xca62c1d6;
    }
}
```

Figure 3.1: Custom Functions of the SHA-1 Kernel

In the main kernel function, the first step is to declare the variables that will be needed in the hash computation: the five intermediate hash variables, eighty message digest words, and the working and temporary variables. The intermediate hash is initialized to the value of the hash buffer, as the host program sets the initial values specified in the hash. The reason this is done, instead of hard coding it into the kernel, is to allow greater flexibility for non-standard applications that may use slightly modified versions of the SHA-1 algorithm. If these initial hash values need to be changed, the kernel does not need to be recompiled. It also has no effect on performance either way. This step is depicted in Figure 3.2.

```
__kernel  void  sha1(__global  unsigned  char  *restrict  messBlock,
__global unsigned *restrict hash, int blockNum)
{
        unsigned H[5];
        unsigned W[80];
        unsigned a, b, c, d, e, T;
        #pragma unroll
        for (int i = 0; i < 5; i++){
                H[i]=hash[i];
        }
        .....
}
```

Figure 3.2: Declaration of Variables in the SHA-1 Kernel

Once the declaration and initialization is completed, the algorithm steps can begin. The kernel needs to operate on a single message block. The number of message blocks is known as an argument to the kernel. A simple loop ranging from *0* to the block number implements the hashing of the entire message. The four steps of the SHA-1 function are written inside of that loop. Calculating the message schedule in the first step consists of two different loops. The first loop sets the first sixteen words of the message schedule. Each word is the concatenation of four characters of the message block. The index of the characters from the message is a function of the current message block and the number of the message schedule. This is the only place in the kernel where the message buffer input is read. The second loop needed to determine the message schedule calculates the seventeenth to eightieth words, which is a function of four previous message schedule words. The derivation of the message schedule is presented in Figure 3.3.

```
#pragma unroll 2
for (int l = 0; l < blockNum; l++){
   #pragma unroll
   for (int t = 0; t < 16; t++){
        W[t] = (((unsigned) messBlock[(l*64)+(t*4)])<<24)
                |(((unsigned) messBlock[(l*64)+(t*4+1)])<<16)
                |(((unsigned) messBlock[(l*64)+(t*4 + 2)])<<8)
                |(((unsigned) messBlock[(l*64)+(t*4 + 3)]));
    }
   #pragma unroll
   for (int t = 16; t < 80; t++){
        W[t] = ROTL(1,(W[t-3]^W[t-8]^W[t-14]^W[t-16]));
    }
        ...
}
```

Figure 3.3: Message Schedule Derivation in the SHA-1 Kernel

The second step of the algorithm simply updates the working variables to the values in the intermediate hash. It is a simple step with no calculation needed. Next comes the eighty rounds of hashing of step three. As shown in Figure 3.4, this step is implemented in a single loop with a range of *0* to eighty, and makes use of the custom functions defined outside of the main kernel function. The bulk of the calculation of the algorithm is computed in this loop.

```
a = H[0];
b = H[1];
c = H[2];
d = H[3];
e = H[4];

#pragma unroll
for (int t = 0; t < 80; t++){
       T = (ROTL(5, a)+ funct(b,c,d,t) e + getK(t) + W[t]);
       e = d;
    d = c;
    c = ROTL(30, b);
    b = a;
    a = T;
```

Figure 3.4: Steps 2 and 3 of the SHA-1 Kernel

Finally the intermediate hash variables are incremented by the value of the working variables in the four step. These steps all repeat for a number of times equal to the block number. Once the last message block has been hashed, the hash buffer can be updated with the final values of the intermediate hash variables, where the host will be able to read the value. The code that implements this is presented in Figure 3.5.

```
       ...
       H[0] += a;
       H[1] += b;
       H[2] += c;
       H[3] += d;
       H[4] += e;
}

#pragma unroll
for (int i = 0; i <5; i++){
       hash[i] = H[i];
}
```

Figure 3.5: Step 4 and Processing Final Hash Variables of SHA-1 Kernel

The full kernel code for the SHA-1 algorithm can be found in Appendix A: SHA-1 Kernel Code.

3.2.2.1 Optimization

The implemented OpenCL design needed to be optimized to increase performance, as the results were quite poor without optimization. There are two main options when it comes to attempting to optimize the design. The first is implementation of pipeline structures in the design. In the AOCL, pipelining is inferred by the compiler wherever possible. The other optimization technique for this algorithm is operation reordering [3]. This involves rearranging some calculations to try to reduce the dependencies between variables.

For pipelining to be inferred on loops, the *#pragma unroll* directive must be used on compatible loops in the program. This unrolls the loop fully, if the loop bounds are known at compile time. If the loop bounds are not known at this time, or if the loop should not be fully unrolled, then the unrolling factor can be specified by the programmer. Using *#pragma unroll <N>* will attempt to unroll that loop *N* times. The unrolled loop will then be pipelined as much as possible after taking variable dependencies into account.

To optimize SHA-1, both methods were attempted. In the case of operation reordering, there was no change in performance with any attempt to pre-compute any portions of the algorithm. The reason that there was no effect is that the critical path of calculation, which is calculating temporary variable *T*, is not reduced by the operation reordering. The delay of calculating that variable is more than the amount of time required to compute the other variables even in the same round, as the FPGA structure allows for calculations to be completed in parallel. This means that the effect of completing some operations in the previous round does not reduce the computation time of the current round.

Being able to unroll the loops present in the SHA-1 algorithm to allow pipelining structures to be used was very effective in increasing the performance, even though there are data dependencies between stages in the pipeline. The effects of inferring pipelines to different portions of the kernel function are discussed in section 3.2.3.

3.2.3 Results

The SHA-1 kernel without any optimization gives very poor performance. The throughput that was measured was even less than the equivalent C++ program. The lack of any pipelining in the implementation prevents any of the speedup that is possible when utilizing the FPGA hardware. To determine the best possible pipelining of the algorithm, many different kernels were created, compiled, and tested to determine the effect of unrolling certain loops of the kernel code. In the end, fourteen kernels with varying degrees of pipelined hardware were designed. Each kernel was tested on nine different test files that ranged in size from 1Mb to over 1200Mb. The throughput of each test was calculated for both the FPGA accelerated OpenCL model, and a equivalent C++ sequential program. Those results were then averaged for each kernel to provide a representative speed for each design. The results can be found in Figure 3.6 with an explanation of each kernel located in Table 3.1. As can be seen, the first three kernels have a very low throughput. The effect of pipelining the reading of the hash buffer in kernel *B* or the first loop of the message schedule step in kernel *C* does not provide any speedup at all. In kernel *D,* however, the throughput jumps up about 600%. This kernel provides pipelining structures to the second loop of the message schedule derivation, where there is some computation being done. A bit more speed is added in kernel *E* by fully unrolling both loops of the message schedule calculations. Throughput is then more than doubled by pipelining the eighty rounds of hashing in kernel *F*. There is only one other loop in the kernel to target and that is the outer loop that increments through the message blocks. This loop does not have bounds that are known at compile time, so it is not possible to unroll it fully. It is still possible to specify the unroll factor to implement some pipelining structures. The kernel *G* has an unroll factor of 2 on that outer loop. Kernels *H* through *M* have unroll factors from 3 to 8, respectively. It is shown that *G* does increase throughput, but as the unroll factor increases after that, it yields unexpected results. The throughput fluctuates, never reaching the same level as in *G*. The area, however, still increases steadily when increasing the unrolling factor. This creates less efficient hardware as there is no performance gain from the increase in hardware. The longer pipelines are harder to fill and cannot make use of the hardware.

The efficiency of each kernel was also profiled in terms of throughput divided by percentage resource utilization (PRU) in target FPGA. As shown in Figure 3.7, the kernel that is most efficient is kernel *F*, meaning that it is utilizing the generated hardware most effectively. Kernel *G*, which had the highest overall performance, is actually using its generated hardware in a less efficient manner. This means that the area cost of unrolling the message block loop from kernel *F* to kernel *G* was a higher percentage than the performance gain between the two. As that unroll factor is further increased, the efficiency of the hardware is decreased. This is expected as the more pipelining that is done utilizes more hardware without the increase in performance to accompany it.

Since the goal is to achieve the highest throughput in the design, kernel *G* was chosen as the proposed design. This kernel was compiled in 3 hours using the AOC and AOCL utility command for the Terasic DE5-Net FPGA accelerator board.

Table 3.1: SHA-1 Kernel Comparison

Kernel	FPGA Throughput	Speedup	FPGA Utilization
A	174.4829	0.838276	21.54%
B	152.3886	0.711982	27.42%
C	167.0678	0.798211	21.18%
D	1147.8380	5.483753	24.05%
E	1283.7360	6.101327	21.83%
F	2762.9570	13.4015	24.72%
G	3016.5530	14.40144	31.70%
H	2865.0930	13.92289	39.96%
I	2996.4340	14.14296	45.94%
J	2480.7400	12.18132	55.29%
K	2599.8980	12.48147	61.34%
L	2589.8260	12.43014	67.98%
M	2929.6360	14.06274	74.50%

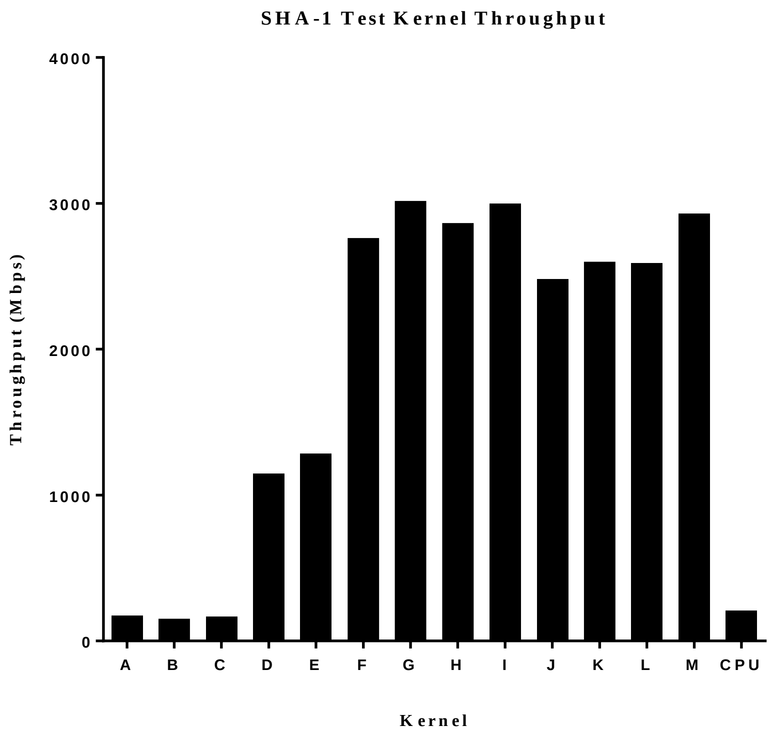

Figure 3.6: SHA-1 Test Kernel Throughput

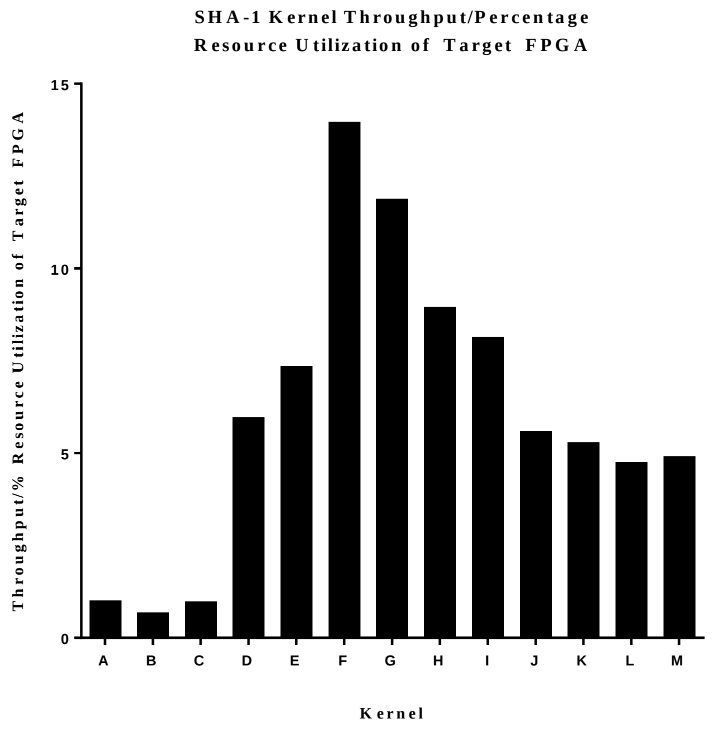

Figure 3.7: SHA-1 Test Kernel Throughput divided by PRU of Target FPGA

The speed of the SHA-1 implementation is dependent on the size of the input message. There are two main reasons for this relationship. The first is that there is an overhead time to launching a kernel. For small messages this can be a large portion of the overall time. The second reason for poor performance on small inputs is the fact the majority of the speedups come from pipelining the design. When there is a small amount of data, the pipelines cannot be fully utilized, reducing the speedup possible. To determine the relationship between the throughput and message size, 372 different input messages were used to test the design. The size of the messages ranged from 10B to 640MB. To ensure an accurate representation of the speed, the digest of each input was calculated six times and the average was taken. The standard deviation between the six values was not significant. The results of this are shown in Figure 3.8. Messages that are smaller than 10^3 bytes give slower performance on FPGA than on CPU. Once the size increases beyond that, there is a steady increase in performance until the message reaches 10^7 bytes, where the speed then plateaus at 3033 Mbps. The CPU implementation has a peak speed of 217 Mbps. The FPGA accelerates the CPU time by a factor of 14 for this algorithm at that size of input.

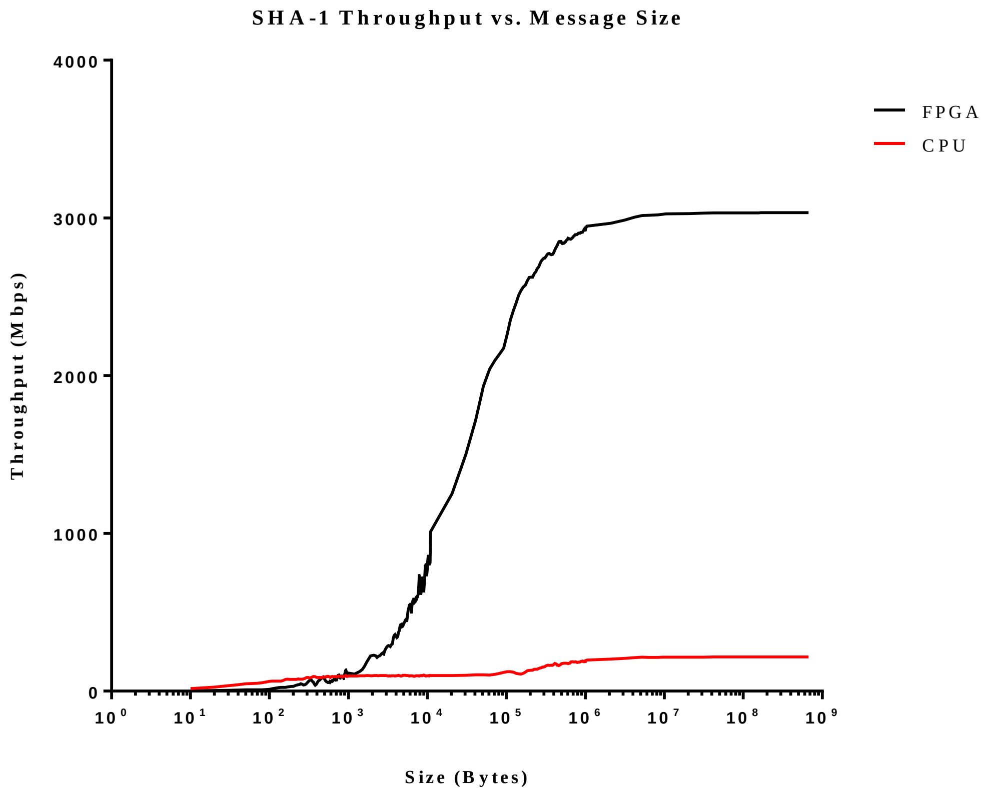

Figure 3.8: SHA-1 Throughput vs. Message Size

The peak throughput of the proposed design is 3033 Mbps. As discussed in section 2.5.1, there has been much work done in academia implementing the SHA-1 in reconfigurable hardware. Table 3.2 compares this design to the related work. The HLS OpenCL model is faster than three of the four other designs. The fastest design, which was implemented by Lee *et al.* is roughly twice the speed that was achieved in this thesis

[30]. Reasons for such a discrepancy in the two designs could be partially due to the throughput calculation. The OpenCL model was measured by recording the time that it took each kernel to run, and dividing the message size by that time. In the related works, the throughput is a function of the number of cycles needed to complete the hash and the clock speed that the system can operate at. They are not showing experimentally determined results, just the theoretical maximum throughput that can be achieved by the hardware. This cannot be evaluated on this design, as the HLS create very complex hardware designs that cannot be traced easily. This work also includes the time needed to read and write to the global memory of the system, as well as the time needed to launch the kernel. All of these factors can affect the measured speed of the FPGA hardware. The performance that was achieved is quite comparable to the other designs, and depending on the final application, this speed is quite impressive. Especially factoring in the decreased design cost and time compared to HDL based design methods.

The area of the proposed design was also compared to that of related work. Due to the differences in area conventions between Altera and Xilinx FPGAs, the area must first be normalized. The area of the Xilinx designs that use slices have been normalized to the LEs of the Altera FPGAs using published formulae [38]. These equations were derived from whitepapers published by both Altera and Xilinx [39], [40]. The area of the HLS model is much greater than the HDL designs of published work. This is because the interface hardware that is automatically built into the kernel to communicate with the RAM and PCIe protocol. The HDL based designs do not incorporate this interface hardware into their area calculations, but it would be needed in actual applications.

Table 3.2: SHA-1 Performance Comparison to Related Work

Design	Throughput	Area	Normalized Area	Target
Sklavos *et al.* [27]	1339 Mbps	2245 CLBs	-	Xilinx 2V500FG456
Kakarountas *et al.* [28]	2526 Mbps	950 CLBs/1164 Dffs	-	Xilinx V150BG352
Yiakoumis *et al.* [29]	1024 Mbps	854 slices	1058 LE	Xilinx Virtix-II
Lee *et al.* [30]	6040 Mbps	2894 Slices	3586 LE	Xilinx Virtex-II XC2V1000
This Work	**3033 Mbps**	**48030 LE**	**48030 LE**	**Altera Stratix V**

3.3 SHA-2

There are six functions available in the SHA-2 family of the secure hash standard [19]. As mentioned in section 2.4.2, there are only two separate algorithms. Because of this, only the SHA-256 and SHA-512 functions were implemented. The performance of the other functions will be identical to these algorithms. Similarly to SHA-1, both of these functions were first implemented on the CPU based on their description in the standard.

3.3.1 SHA-256

The SHA-256 variant was implemented first. There are many similar aspects of SHA-1 and SHA-256 such as the padding, word size, and block size. It can also be used to implement the SHA-224 function by using different initialization values and truncating the final output digest to 224-bits.

3.3.1.1 Host Program

Due to the similarities between SHA-1 and SHA-256, the host programs of the two are very similar. Like in all OpenCL applications, the host must define the OpenCL device, kernel, command queue, and memory objects that were discussed in detail in section 2.3.1.1. The host program sets the three arguments to the SHA-256 kernel, the message buffer, the hash buffer, and the number of message blocks. The creation of the buffers are identical to that in the SHA-1. The host implements the padding of the input message before executing the kernel as a single work-item on the FPGA accelerator card to calculate the hash.

3.3.1.2 Kernel

This time, there are eight custom functions defined in the kernel apart from the main kernel function. The eight custom functions all return an unsigned 32-bit word, and implement the following functions; *ROTR*, *SHR*, *choose*, *majority*, $\sum_0$, $\sum_1$, σ_0, and σ_1. All of these are described in section 2.4.2.1 and the code is shown in Figure 3.9.

```
unsigned ROTR( int n, unsigned x){
        return ((x >> n)) | ((x) << (32-n));
}
unsigned SHR( int n, unsigned x){
        return ((x >> n));
}
unsigned choose (unsigned x, unsigned y, unsigned z){
        return (((x&y)^((~x)&z)));
}
unsigned majority (unsigned x, unsigned y, unsigned z){
        return (((x&y)^(x&z)^(y&z)));
}
unsigned SIGMA0(unsigned x){
        return ((ROTR(2, x)^ROTR(13, x)^ROTR(22,x)));
}
unsigned SIGMA1(unsigned x){
        return ((ROTR(6, x)^ROTR(11, x)^ROTR(25,x)));
}
unsigned sig0(unsigned x){
        return ((ROTR(7, x)^ROTR(18, x)^SHR(3,x)));
}
unsigned sig1(unsigned x){
        return ((ROTR(17, x)^ROTR(19, x)^SHR(10,x)));
}
```

Figure 3.9: Custom Functions of the SHA-256 Kernel

In the main kernel function, the first step is to define the variables needed in the calculation. These consist of sixty-four 32-bit words for the message schedule, eight 32-bit words to make up the 256-bit intermediate hash, the eight working variables, and the two temporary variables. This kernel also needs sixty-four constants that are defined here, as well as another sixty-four characters to cache the message block. The intermediate hash is set to the initial values specified in the host program. This way, the kernel can be used for the SHA-256 and the SHA-224 algorithms by setting the proper initialization values specified in section 2.4.2.1.1. The OpenCL implementation of this step is presented in Figure 3.10.

```
__kernel  void  sha256(__global  const  unsigned  char  *restrict
messBlock, __global unsigned *restrict hash, int blockNum) {
        unsigned char messCache[64];
        unsigned H[8];
        unsigned W[64];
        unsigned a, b, c, d, e, f, g, h, T1, T2;
        const unsigned K[64] = {
                0x428a2f98,  0x71374491,  0xb5c0fbcf,  0xe9b5dba5,
                0x3956c25b,  0x59f111f1,  0x923f82a4,  0xab1c5ed5,
                0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,
                0x72be5d74,  0x80deb1fe,  0x9bdc06a7,  0xc19bf174,
                0xe49b69c1,  0xefbe4786,  0x0fc19dc6,  0x240ca1cc,
                0x2de92c6f,  0x4a7484aa,  0x5cb0a9dc,  0x76f988da,
                0x983e5152,  0xa831c66d,  0xb00327c8,  0xbf597fc7,
                0xc6e00bf3,  0xd5a79147,  0x06ca6351,  0x14292967,
                0x27b70a85,  0x2e1b2138,  0x4d2c6dfc,  0x53380d13,
                0x650a7354,  0x766a0abb,  0x81c2c92e,  0x92722c85,
                0xa2bfe8a1,  0xa81a664b,  0xc24b8b70,  0xc76c51a3,
                0xd192e819,  0xd6990624,  0xf40e3585,  0x106aa070,
                0x19a4c116,  0x1e376c08,  0x2748774c,  0x34b0bcb5,
                0x391c0cb3,  0x4ed8aa4a,  0x5b9cca4f,  0x682e6ff3,
                0x748f82ee,  0x78a5636f,  0x84c87814,  0x8cc70208,
                0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2
                };

#pragma unroll
        for (int i = 0; i < 8; i++){
                H[i]=hash[i];
        }
```

Figure 3.10: Variable Declaration and Initialization of the SHA-256 Kernel

Just like in the SHA-1 kernel, the hash operation is done by looping through the message blocks from *0* to the block number argument specified to the kernel function. The first thing that is done on each message block is to cache the 512-bit block, or sixty-four 8-bit characters of the message into the local memory of the FPGA device. Then, the four steps that makeup the algorithm are processed. The message schedule is calculated using two loops, one for the first sixteen words, and another for the rest. Instead of using the message block to determine the first set of message schedule words, the cached characters are used. The seventeenth to sixty-fourth words of the message schedule are determined using four previous message schedule words, and the two custom functions, σ_0, and σ_1. This is depicted in Figure 3.11. Compared to the SHA-1 algorithm, the SHA-256 message schedule requires much more complex calculations.

```
#pragma unroll 2
for (int l = 0; l < blockNum; l++){
        int offset = l*64;
#pragma unroll
        for (int z = 0; z < 64; z++){
                messCache[z] = messBlock[z + offset];
        }
#pragma unroll
        for (int t = 0; t < 16; t++){
                int t4 = t*4;
                W[t] = (((unsigned) messCache[(t4)]) << 24)
                     | (((unsigned) messCache[(t4 + 1)]) << 16)
                     | (((unsigned) messCache[(t4 + 2)]) << 8)
                     | (((unsigned) messCache[(t4 + 3)]));
        }
#pragma unroll
        for (int t = 16; t < 64; t++){
                W[t] = (sig1(W[t-2])+W[t-7]+sig0(W[t-15])+W[t-16]);
        }
```

Figure 3.11: Message Block Caching and Message Schedule Derivation of the SHA-256 Kernel

The next step, shown in Figure 3.12, is updating of the working variables is a simple assignment operation of the eight words to the intermediate hash variables. In the third step of the SHA-256 kernel, the sixty-four rounds of hashing are completed by using a loop with bounds of *0* to *64*. The two temporary variables are dependent on the *choose*, *majority*, $\sum_0$, and $\sum_1$ custom functions, the defined constants, the message schedule, and the working variables.

```
a = H[0];
b = H[1];
c = H[2];
d = H[3];
e = H[4];
f = H[5];
g = H[6];
h = H[7];
#pragma unroll
for (int t = 0; t < 64; t++){
        T1 = (h + SIGMA1(e) + choose(e,f,g) + K[t] + W[t]);
        T2 = (SIGMA0(a) + majority(a,b,c));
        h = g;
        g = f;
        f = e;
        e = (d + T1);
        d = c;
        c = b;
        b = a;
        a = (T1 + T2);
}
```

Figure 3.12: Steps 2 and 3 of the SHA-256 Kernel

The intermediate hash variables are lastly incremented by the values of working variables. The process will repeat for all the message blocks contained in the input message. Finally, the value of the hash buffer is updated to the calculated value to allow the host to read the digest from global memory. The code of this is presented in Figure 3.13.

```
    ...
    H[0] += a;
    H[1] += b;
    H[2] += c;
    H[3] += d;
    H[4] += e;
    H[5] += f;
    H[6] += g;
    H[7] += h;
}
#pragma unroll
for (int i = 0; i <8; i++){
    hash[i] = H[i];
}
```

Figure 3.13: Step 4 and Processing Final Hash Variables of SHA-256 Kernel

The full kernel code of the SHA-256 algorithm can be found in Appendix B: SHA-256 Kernel Code.

3.3.1.2.1 Optimization

The optimization of the SHA-256 kernel is done in a similar fashion to the SHA-1 kernel as the algorithms, which are both made of the Merkle–Damgård construction, are quite similar. The methods of optimization used are detailed in section 3.2.2.1.

In SHA-256 the critical path of the algorithm occurs when updating the working variable *e*, in the hashing rounds. Since variable *e* depends on temporary variable T_1, which contains the most computation to determine in the entire function. This is the step that limits the ability to optimize the algorithm further.

3.3.1.3 Results

Using the knowledge of optimizing the SHA-1 kernel, and the similarity of the algorithms, it was simple to implement pipelining structures in the appropriate loops of the algorithm. All of the internal loops that implement the message caching, schedule derivation, and the rounds of hashing were unrolled to infer pipelines. The overall message block loop was unrolled by a factor of two. The higher complexity of the calculations in the algorithm cause the stalling of this pipeline, and performance suffered if the loop was unrolled further than that. The kernel was compiled using the AOCL tools in 3 hours and 47 minutes.

Using the same 372 test messages as the SHA-1 test, the relationship between the size of the message and the throughput of the design was measured. The test was completed six times on each message to give a good representation of the average speed. This relationship is represented in Figure 3.14. The pattern that is exhibited by the results has the identical shape as the previous algorithm. For messages smaller than 10^3 bytes, the CPU implementation is faster. As the message size increases, however, the throughput increases quickly as well. At a size between 10^6 and 10^7 bytes the throughput levels off at 1489 Mbps. The CPU program has a maximum throughput of 140 Mbps. This throughput is about a 10 fold increase over that of the CPU at this input size.

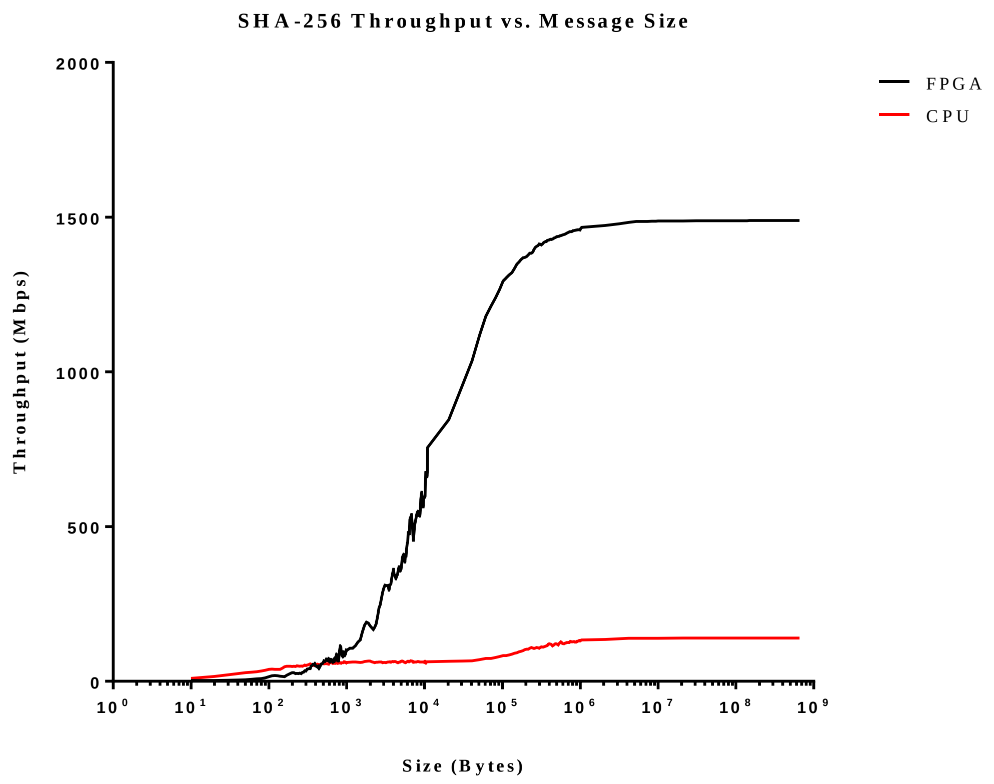

Figure 3.14: SHA-256 Throughput vs. Message Size

The maximum throughput of the SHA-256 kernel was 1489 Mbps which is less than half of the throughput achieved in the SHA-1 case. The more complex calculation of the SHA-256 algorithm is to blame for this reduction in performance, but the security of the function is much higher. Comparing this work to other SHA-256 FPGA implementations in literature show that the performance achieved was on par with the top contenders. The details are given in Table 3.3. Of the seven other designs that were compared, only two have a higher throughput than the HLS OpenCL model. Michail *et al.* implemented their design on three different FPGAs, producing a range of results from 2077 Mbps to 3100 Mbps [33]. This work was not far off of the low end of that range. Even closer in throughout was Mestiri *et al.* with a 1618 Mbps result [34]. This shows that the acceleration by AOCL provided performance that is very similar to the top performing designs in literature.

The area of the proposed SHA-256 model was compared to published designs using the same method described in section 3.2.3 to normalize the area of the Xilinx models. The AOCL design has an area much smaller than the capacity of the target FPGA, but it was observed that the area of this work was much larger than any HDL based designs. Again, this is mainly due to the incorporation of the host interface hardware needed in this method. For any application to fully utilize the circuitry given in the published literature, interfacing hardware would need to be designed to communicate with the FPGA hardware. This would drastically increase the area needed and the design time and cost, which are greatly reduced with this HLS tool.

Table 3.3: SHA-256 Performance Comparison to Related Work

Design	Throughput	Area	Normalized Area	Target
Sklavos *et al.* [31]	326 Mbps	1060 CLBs	-	Xilinx V200PQ240
Ling Bai and Shuguo Li [41]	327 Mbps	2633 LE	2633 LE	Altera Cyclone III EP3C25U240C8
Khalil *et al.* [32]	634 Mbps	4489 LE	4489 LE	Altera Stratix EP1S10
McEvoy et. al [42]	1009 Mbps	1373 slices	2510 LE	Virtex II XC2V2000-BF957
Mohamed and Nadjia [43]	1360 Mbps	1203 slices	2200 LE	Virtex-5
Michail *et al.* [33]	2077-3100 Mbps	1534-1708 slices	2805-3123 LE	Virtex, Virtex-II, Virtex-E
Mestiri *et al.* [34]	1618 Mbps	387 Slices	708 LE	Xilinx Virtex-5 XC5VFX70T
This Work	**1489 Mbps**	**42486 LE**	**42486 LE**	**Altera Stratix V**

3.3.2 SHA-512

The SHA-512 variant has a similar structure to that of SHA-256, but there are some very important differences. The major differences between this and the previous algorithms is that the word size is 64-bits and the block size is 1024-bits. The padding rule is also slightly changed to account for these different sizes. The SHA-512 function is also used to implement SHA-384, SHA-512/224, and SHA-512/256.

3.3.2.1 Host Program

The host program of the SHA-512 application declares and initializes all the OpenCL components needed that are described in section 2.3.1.1. The padding of the input message is handled by the host program. It is more complex in this algorithm as the padding rule needs a 128-bit representation of the messages length. The largest variable that is available is 64-bits long. This means that two words must be used to represent the length of the message. Other than the padding, the host program is very similar to the other algorithms. The two buffers needed to send data to the kernel are setup, adjusting for the proper length of the digest size. The initial hash values are also set in the host to provide the flexibility of implementing the other SHA-2 functions.

3.3.2.2 Kernel

The kernel of the SHA-512 follows an almost identical pattern to SHA-256 kernel in section 3.3.1.2. Again, there are eight custom functions defined outside the main kernel function, but this time, they all return an unsigned 64-bit word. The *ROTR*, *SHR*, *choose*, *majority*, $\sum_0$, $\sum_1$, σ_0, and σ_1 functions of SHA-512 are described in section 2.4.2.2, and somewhat differ in functionality to those in SHA-256. The OpenCL implementation code is presented in Figure 3.15.

```
unsigned long ROTR( int n, unsigned long x){
        return ((x >> n)) | ((x) << (64-n));
}
unsigned long SHR( int n, unsigned long x){
        return ((x >> n));
}
unsigned long choose (unsigned long x, unsigned long y, unsigned long z){
        return (((x&y)^((~x)&z)));
}
unsigned long majority (unsigned long x, unsigned long y, unsigned long z){
        return (((x&y)^(x&z)^(y&z)));
}
unsigned long SIGMA0(unsigned long x){
        return ((ROTR(28, x)^ROTR(34, x)^ROTR(39,x)));
}
unsigned long SIGMA1(unsigned long x){
        return ((ROTR(14, x)^ROTR(18, x)^ROTR(41,x)));
}
unsigned long sig0(unsigned long x){
        return ((ROTR(1, x)^ROTR(8, x)^SHR(7,x)));
}
unsigned long sig1(unsigned long x){
        return ((ROTR(19, x)^ROTR(61, x)^SHR(6,x)));
}
```

Figure 3.15: Custom Functions of the SHA-512 Kernel

In the actual kernel function, the variables declared are: eight 64-bit working and two 64-bit temporary variables, eight 64-bit words to implement the 512-bit intermediate hash, eighty 64-bit words for the message schedule, one hundred and twenty-eight 8-bit characters for message block caching, and eighty 64-bit defined constants, as shown in Figure 3.16.

```
__global unsigned long *restrict hash, int blockNum) {
unsigned char messCache[128];
unsigned long H[8];
unsigned long W[80];
unsigned long a, b, c, d, e, f, g, h, T1, T2;
const unsigned long long K[80] = {
        0x428a2f98d728ae22, 0x7137449123ef65cd, 0xb5c0fbcfec4d3b2f,
        0xe9b5dba58189dbbc, 0x3956c25bf348b538, 0x59f111f1b605d019,
        0x923f82a4af194f9b, 0xab1c5ed5da6d8118, 0xd807aa98a3030242,
        0x12835b0145706fbe, 0x243185be4ee4b28c, 0x550c7dc3d5ffb4e2,
        0x72be5d74f27b896f, 0x80deb1fe3b1696b1, 0x9bdc06a725c71235,
        0xc19bf174cf692694, 0xe49b69c19ef14ad2, 0xefbe4786384f25e3,
        0x0fc19dc68b8cd5b5, 0x240ca1cc77ac9c65, 0x2de92c6f592b0275,
        0x4a7484aa6ea6e483, 0x5cb0a9dcbd41fbd4, 0x76f988da831153b5,
        0x983e5152ee66dfab, 0xa831c66d2db43210, 0xb00327c898fb213f,
        0xbf597fc7beef0ee4, 0xc6e00bf33da88fc2, 0xd5a79147930aa725,
        0x06ca6351e003826f, 0x142929670a0e6e70, 0x27b70a8546d22ffc,
        0x2e1b21385c26c926, 0x4d2c6dfc5ac42aed, 0x53380d139d95b3df,
        0x650a73548baf63de, 0x766a0abb3c77b2a8, 0x81c2c92e47edaee6,
        0x92722c851482353b, 0xa2bfe8a14cf10364, 0xa81a664bbc423001,
        0xc24b8b70d0f89791, 0xc76c51a30654be30, 0xd192e819d6ef5218,
        0xd69906245565a910, 0xf40e35855771202a, 0x106aa07032bbd1b8,
        0x19a4c116b8d2d0c8, 0x1e376c085141ab53, 0x2748774cdf8eeb99,
        0x34b0bcb5e19b48a8, 0x391c0cb3c5c95a63, 0x4ed8aa4ae3418acb,
        0x5b9cca4f7763e373, 0x682e6ff3d6b2b8a3, 0x748f82ee5defb2fc,
        0x78a5636f43172f60, 0x84c87814a1f0ab72, 0x8cc702081a6439ec,
        0x90befffa23631e28, 0xa4506cebde82bde9, 0xbef9a3f7b2c67915,
        0xc67178f2e372532b, 0xca273eceea26619c, 0xd186b8c721c0c207,
        0xeada7dd6cde0eb1e, 0xf57d4f7fee6ed178, 0x06f067aa72176fba,
        0x0a637dc5a2c898a6, 0x113f9804bef90dae, 0x1b710b35131c471b,
        0x28db77f523047d84, 0x32caab7b40c72493, 0x3c9ebe0a15c9bebc,
        0x431d67c49c100d4c, 0x4cc5d4becb3e42b6, 0x597f299cfc657e2a,
        0x5fcb6fab3ad6faec, 0x6c44198c4a475817
        };
#pragma unroll
        for (int i = 0; i < 8; i++){
                H[i]=hash[i];
        }
```

Figure 3.16: Variable Declaration and Initialization of the SHA-512 Kernel

The current 1024-bit message block is first cached from the characters from the global memory buffer argument of the kernel. To compute the message schedule, again there are two loops. The first’s bounds are *0* to *16*, and each schedule word is eight caching characters concatenated together. The second loop’s bounds are *16* to *80*, using the functions σ_0, and σ_1, with four previous schedule word values. This message block caching and message schedule derivation is shown in Figure 3.17.

```
for (int l = 0; l < blockNum; l++){
        int offset = l*128;
        #pragma unroll
        for (int z = 0; z < 128; z++){
                messCache[z] = messBlock[z + offset];
        }
        #pragma unroll
        for (int t = 0; t < 16; t++){
                int t4 = t*8;
                W[t] = (((unsigned long) messCache[(t4)]) << 56)
                     | (((unsigned long) messCache[(t4 + 1)]) << 48)
                     | (((unsigned long) messCache[(t4 + 2)]) << 40)
                     | (((unsigned long) messCache[(t4 + 3)]) << 32)
                     | (((unsigned long) messCache[(t4 + 4)]) << 24)
                     | (((unsigned long) messCache[(t4 + 5)]) << 16)
                     | (((unsigned long) messCache[(t4 + 6)]) << 8)
                     | (((unsigned long) messCache[(t4 + 7)]));
        }
        #pragma unroll
        for (int t = 16; t < 80; t++){
                W[t] = (sig1(W[t-2])+W[t-7]+sig0(W[t-15])+W[t-16]);
        }
        ...
```

Figure 3.17: Message Block Caching and Message Schedule Derivation of the SHA-512 Kernel

Step two, presented in Figure 3.18, simply updates the working variables, similar to both the SHA-1 and SHA-2 functions. In the third step, there are eighty rounds of hashing computed in the loop instead of sixty-four in SHA-256. The calculations are the same, however, with the exception of the increased word size and the redefinition of the custom functions $\sum_0$, and $\sum_1$ for this algorithm.

```
a = H[0];
b = H[1];
c = H[2];
d = H[3];
e = H[4];
f = H[5];
g = H[6];
h = H[7];
#pragma unroll
for (int t = 0; t < 80; t++){
        T1 = (h + SIGMA1(e) + choose(e,f,g) + K[t] + W[t]);
        T2 = (SIGMA0(a) + majority(a,b,c));
        h = g;
        g = f;
        f = e;
        e = (d + T1);
        d = c;
        c = b;
        b = a;
        a = (T1 + T2);
}
```

Figure 3.18: Steps 2 and 3 of the SHA-512 Kernel

The intermediate hash is updated, and the entire process is repeated for each message block. Lastly, the OpenCL buffer for the hash is updated to be able to have the host program access it, as shown in Figure 3.19.

```
    ...
    H[0] += a;
    H[1] += b;
    H[2] += c;
    H[3] += d;
    H[4] += e;
    H[5] += f;
    H[6] += g;
    H[7] += h;
}
#pragma unroll
for (int i = 0; i <8; i++){
    hash[i] = H[i];
}
```

Figure 3.19: Step 4 and Processing Final Hash Variables of SHA-512 Kernel

The full kernel code for the SHA-512 algorithm can be found in Appendix C: SHA-512 Kernel Code.

3.3.2.2.1 Optimization

Refer to section 3.3.1.2.1 as the optimization of SHA-512 is identical to that of SHA-256.

3.3.2.3 Results

Like the SHA-256 kernel, all of the inner loops were fully unrolled to create the pipelines for those portions of the algorithm. The outer loop, which cycles through message blocks was not unrolled at all, due to the performance penalty of stalling the long pipeline. This kernel was compiled in 4 hours and 40 minutes using the AOCL.

Again, the algorithm was tested using 372 input messages to determine the relationship between the message size and throughput of the proposed design. The test was completed six times for every input message and the average throughput was taken. Figure 3.20 shows the results of this test. A message with the length of 10^3 bytes can be calculated on a CPU or the FPGA in the same time. Smaller messages than that are completed faster on the CPU. Messages bigger than 10^3 bytes have a steady increase in throughput with increasing size. When the message size reaches between 10^6 and 10^7, the increase in throughput stops and the performance of the FPGA implementation plateaus at 1692 Mbps. The throughput of the CPU design also levels off around this spot to be 208 Mbps, giving the FPGA accelerated program an 8 fold speedup.

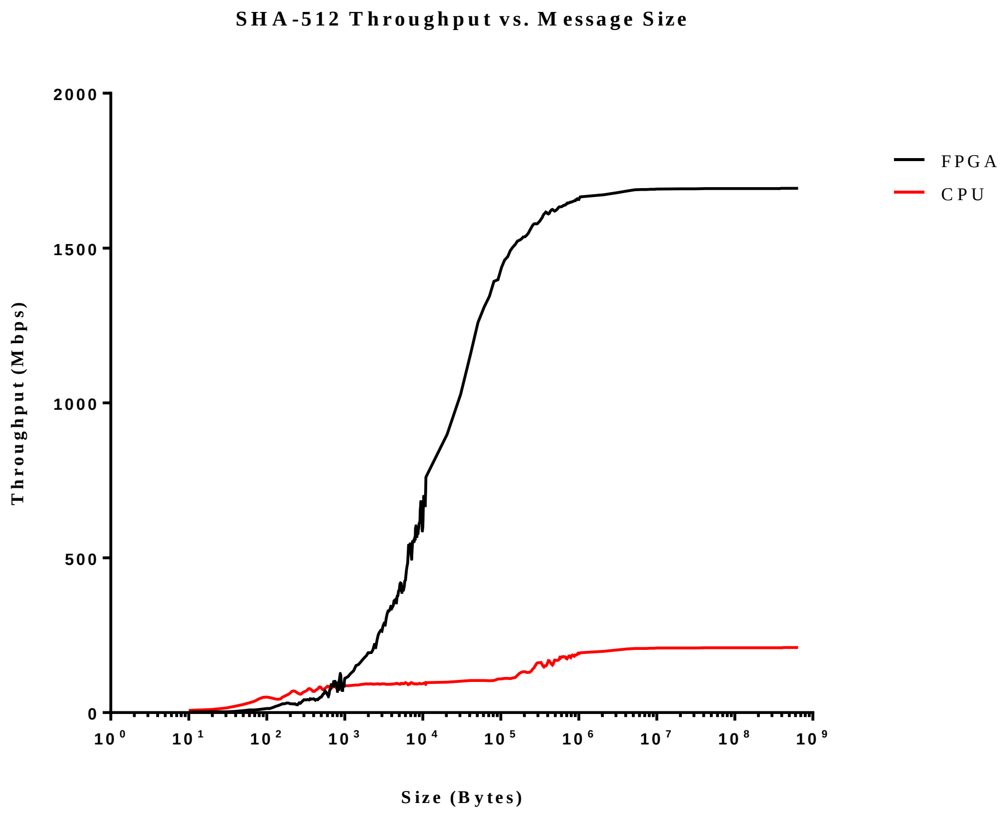

Figure 3.20: SHA-512 Throughput vs. Message Size

When comparing the proposed SHA-512 HLS OpenCL model to the HDL based FPGA designs in published literature, five other designs were considered. Of the five, only one had a higher throughput, which was 2253 Mbps and was achieved by Mestiri *et al.* [34]. The next closest to this work was McEvoy *et al.* with a throughput of 1466 Mbps [42]. The other designs had even lower performance than that. In terms of competing with HDL based designs, the OpenCL HLS implementation proposed in this work can provide just as fast results.

In terms of area, the comparison of the SHA-512 to that of the published HDL based designs is similar to SHA-1 and SHA-256. The area of the Xilinx designs were normalized to Altera LEs using the method described in section 3.2.3. Once again, the proposed HLS model had a huge area compared to any of the other designs. The FPGA hardware needed to interface with the host PC is the main contributor to the large area. The ability to have that interfacing logic automatically generated saves a huge amount of design time and cost.

Table 3.4: SHA-512 Performance Comparison to Related Work

Design	Throughput	Area	Normalized Area	Target
Sklavos *et al.* [31]	480 Mbps	2237 CLBs	-	Xilinx V200PQ240
Mestiri *et al.* [34]	2253 Mbps	874 slices	1598 LE	Xilinx Virtex-5 (XC5VFX70T)
McEvoy *et al.* [42]	1466 Mbps	4107 slices	7508 LE	Virtex II (XC2V2000-BF957)
Khalil *et al.* [32]	644 Mbps	4957 LE	4957 LE	Altera Stratix EP1S10
He *et al.* [44]	1216 Mbps	1151 slices	2104 LE	Xilinx XC5VLX220
This Work	**1692 Mbps**	**114771 LE**	**114771 LE**	**Altera Stratix V**

3.4 SHA-3

The SHA-3 algorithm was the last of the secure hash functions to be implemented. Like the other SHAs, the first step was to create a working model on the CPU. Using the SHA-3 specification documentation, the CPU program was designed using C++ [20].

3.4.1 Host Program

After a working CPU program was completed, designing the OpenCL host program is the first step in the creating the FPGA accelerated version. The OpenCL device, kernel, command queue, and buffers are all declared. The kernel arguments must also be set by the host program. In this algorithm, there are three arguments to set. The first is the memory buffer to transfer the input message to the kernel. The second argument is the memory buffer that will hold the *state* of the Keccak function. The last argument is the number of message blocks. In SHA-3 the message block size is dependent on the digest size as described in section 2.4.3. Due to this, the number of message blocks required change depending on the function implemented. The host program implements the padding of the message as well, before it is written to the memory buffer. The kernel is then executed as a single work-item, and the final *state* is available in the state memory buffer. The digest is then read from the *state*.

3.4.2 Kernel

In the kernel file, there are only two custom functions and the main kernel function. The first custom function is *ROTL* and it returns an unsigned 64-bit word. The other is a function to return an integer that is the modulus 5 of the input argument. There is a modulus operator available in OpenCL, however it uses division. This application is not using the modulus function on very large numbers, so a simple iterative subtraction implementation of the modulus function is faster and will use less FPGA resources. The code for these functions are presented in Figure 3.21.

```
unsigned long ROTL(  unsigned long x, int n){
        return ((x << n)) | ((x) >> (64-n));
}

int mod5(int x){
        while (x > 4){
                x -= 5;
        }
        return x;
}
```

Figure 3.21: Custom Functions of the SHA-3 Kernel

The main function of the kernel first defines the message block size for the function. Each of the SHA-3 variants needs a separate kernel as the block size must be hard coded into the function to be able to optimize the design properly. The kernel then declares two arrays *A* and *B*, which are each twenty-five 64-bit words in size. These are to hold the *state* and a temporary version of the *state* respectively. The reason that the *state* is implemented as a one-dimensional array is to help indexing and optimizing of certain steps of the algorithm. The kernel also declares two small five word arrays to hold coefficients calculated in the function. The *state* array is initialized to contain all zeros. The twenty-four constants needed for the *ι* step mapping are defined here as well. The declaration and initialization is presented in Figure 3.22.

```
__kernel void sha3(__global const unsigned char *restrict PM, __global
unsigned long *restrict hash, int blockNum)
{

        int blockSize = 144; //for SHA3-224. Replace with 136 for SHA3-256,
                               104 for SHA3-384, and 72 for SHA3-512.

        unsigned long A[25], B[25], C[5], D[5];
        for (int i = 0; i < 25; i++){
                A[i]= 0x00;
        }
        unsigned long iota[24] =
                {
                0x0000000000000001, 0x0000000000008082, 0x800000000000808a,
                0x8000000080008000, 0x000000000000808b, 0x0000000080000001,
                0x8000000080008081, 0x8000000000008009, 0x000000000000008a,
                0x0000000000000088, 0x0000000080008009, 0x000000008000000a,
                0x000000008000808b, 0x800000000000008b, 0x8000000000008089,
                0x8000000000008003, 0x8000000000008002, 0x8000000000000080,
                0x000000000000800a, 0x800000008000000a, 0x8000000080008081,
                0x8000000000008080, 0x0000000080000001, 0x8000000080008008
                };
        ...
```

Figure 3.22: Declaration and Initialization of the Variables of the SHA-3 Kernel

The SHA-3 kernel then uses a loop to cycle through each message block. Inside of this loop, the first step is to copy the message block into the *state*. Each word of the *state* is set to itself *XORed* by the concatenation of eight 8-bit characters of the message block. After the message block has been absorbed by the *state*, the hashing rounds are implemented by loop with bounds *0* to *24*. This step is depicted in Figure 3.23.

```
for (int l = 0; l < blockNum; l++){

        int offset = blockSize*l;
        #pragma unroll
        for (int k = 0; k < blockSize/8; k++){
                A[k] ^= (((unsigned long) PM[(k*8) + offset])
                       | (((unsigned long) PM[k*8 +1 + offset]) << 8)
                       | (((unsigned long) PM[k*8 +2 + offset]) << 16)
                       | (((unsigned long) PM[k*8 +3 + offset]) << 24)
                       | (((unsigned long) PM[k*8 +4 + offset]) << 32)
                       | (((unsigned long) PM[k*8 +5 + offset]) << 40)
                       | (((unsigned long) PM[k*8 +6 + offset]) << 48)
                       | (((unsigned long) PM[k*8 +7 + offset]) << 56));

        }
        for (int roundNum = 0; roundNum < 24; roundNum++){
                ...
```

Figure 3.23: Absorbing the Message Block into the State of SHA-3 Kernel

The first step mapping, θ, is implemented in three loops, all with bounds *0* to *5*. The first two loops are used to calculate coefficients that are determined using the *state* and the custom functions described above. The last loop of θ applies each of these coefficients by *XORing* them into the appropriate row of the *state*. The θ step mapping is shown in Figure 3.24.

```
#pragma unroll
for (int i = 0; i < 5; i++){
        C[i] = A[i]^A[5+i]^A[10+i]^A[15+i]^A[20+i];
}
#pragma unroll
for (int i = 0; i < 5; i++){
        D[i] = C[mod5(i+4)] ^ ROTL(C[mod5(i+1)],1);
}
#pragma unroll
for (int i = 0; i < 5; i++){
        A[i] ^= D[i];
        A[5+i] ^= D[i];
        A[10+i] ^= D[i];
        A[15+i] ^= D[i];
        A[20+i] ^= D[i];
}
```

Figure 3.24: Theta Step Mapping of the SHA-3 Kernel

The next two step mappings, ρ and π, are implemented together. Since ρ rotates each row, and π reorders them, the assignment of each rotated row to the proper coordinate of a temporary *state* is all that is need to achieve both of these steps. Figure 3.25 presents the combined implementation of the ρ and π step mappings.

```
B[0] = A[0];
B[10] = ROTL(A[1] ,1);
B[7] = ROTL(A[10] ,3);
B[11] = ROTL(A[7] ,6);
B[17] = ROTL(A[11] ,10);
B[18] = ROTL(A[17] ,15);
B[3] = ROTL(A[18] ,21);
B[5] = ROTL(A[3] ,28);
B[16] = ROTL(A[5] ,36);
B[8] = ROTL(A[16] ,45);
B[21] = ROTL(A[8] ,55);
B[24] = ROTL(A[21] ,2);
B[4] = ROTL(A[24] ,14);
B[15] = ROTL(A[4] ,27);
B[23] = ROTL(A[15] ,41);
B[19] = ROTL(A[23] ,56);
B[13] = ROTL(A[19] ,8);
B[12] = ROTL(A[13] ,25);
B[2] = ROTL(A[12] ,43);
B[20] = ROTL(A[2] ,62);
B[14] = ROTL(A[20] ,18);
B[22] = ROTL(A[14] ,39);
B[9] = ROTL(A[22] ,61);
B[6] = ROTL(A[9] ,20);
B[1] = ROTL(A[6] ,44);
```

Figure 3.25: Rho and Pi Step Mappings of the SHA-3 Kernel

The χ step mapping sets each word in the *state* to the value of the corresponding word in the temporary *state*, *XORed* with a function of the next word of the *state* and the word after that. This step is implemented using a loop with bounds *0* to *25*, but increments by *5* each iteration. The χ step mapping is shown in Figure 3.26.

```
#pragma unroll
for (int i = 0; i < 25; i+=5){
       A[i] = B[i] ^(~(B[(i+1)])  & B[(i+2)]);
       A[i+1] = B[i+1] ^(~(B[(i+2)])  & B[(i+3)]);
       A[i+2] = B[i+2] ^(~(B[(i+3)])  & B[(i+4)]);
       A[i+3] = B[i+3] ^(~(B[(i+4)])  & B[(i)]);
       A[i+4] = B[i+4] ^(~(B[(i)])  & B[(i+1)]);

}
```

Figure 3.26: Chi Step Mapping of the SHA-3 Kernel

The last step mapping, *ι*, absorbs a constant into the first word of the *state* that is dependent on the current round number. These step mappings are all executed twenty-four times for each message block. The final process of the kernel is to set the *state* memory buffer to the *state* calculated in the kernel. This way the host program can extract the digest from the *state*. The *ι* step mapping and last of the kernel code is shown in Figure 3.27.

```
              ...
              A[0] ^= iota[roundNum];
       }
}

for (int i = 0; i < 25; i++){
              hash[i]= A[i];
       }
```

Figure 3.27: Iota Step Mapping and Final Processing of the State in the SHA-3 Kernel

The full kernel code of the SHA-3 algorithm can be found in Appendix D: SHA-3 Kernel Code.

3.4.2.1 Optimization

The main optimization technique that can be performed on the SHA-3 kernel is to unroll the loops of the algorithm using the *#pragma unroll* directive to infer pipelining structures in the FPGA hardware. The block size was required to be hard coded into the algorithm, meaning separate kernels needed to be compiled for each SHA-3 function. The reason for that is that there is a loop in the kernel that have bounds dependent on the block size, and without knowing that size at compile time, it is not possible to fully unroll that loop to increase performance. The other key to optimization of this algorithm was to use a one-dimensional index on the *state* array rather than a two-dimensional one that would be easier to compare to the *state* representation in the standard documentation. The pipelining of the hardware is more difficult to implement on nested loops, which would be more plentiful in a two-dimensional indexing scheme.

3.4.3 Results

The SHA-3 algorithm is based on the Keccak sponge function, rather than the Merkle–Damgård construction like SHA-1 and SHA-2 [24], [25]. The effects of unrolling each loop of the kernel cannot be extrapolated from the previous algorithms, like was done in SHA-2. To determine the effects of pipelining different portions of the algorithm, six different kernels of the SHA3-224 variant were constructed and tested on a 70Mb input message. Each test was done in triplicate, and the average throughput was taken along with the speed of the CPU implementation. In kernel *A*, there is no unrolling done at all in the algorithm and the performance is very poor. The throughput is only about half of the CPU implementation. The next kernel, *B*, attempts to unroll the loop to absorb the message block into the *state*, along with the θ and χ step mapping loops. The ρ and π step mappings were implemented in loop in this kernel, which was not unrolled. The throughput marginally increase, but the speed was still slower than the CPU model. In kernel *C*, the ρ and π step mappings were rewritten without the use of a loop. Since there are no data dependencies in the step, the hardware generated performs very well. The increase in throughput was approximately 400% over the previous kernel. The round loop was unable to be fully unrolled as the generated design was too big for the FPGA. In kernel *D*, the only thing that change was unrolling the round loop by a factor of 2. The throughput dropped slightly with this modification. Kernel *E* is similar to kernel *C* but the outer message block loop was unrolled by a factor of 2. The result was that the speed of the design dropped by 75%. The last kernel implemented does not have the block size hard coded into the function, but is set through a kernel argument. The same loops were unrolled as kernel *C* except for the loop that absorbs the message block into the *state*. It cannot be fully unrolled as the bounds are not known at compile time. The throughput of this kernel was only a third of kernel *C*. The results of this test are given in Figure 3.28.

The efficiency of each of the kernels was determined by dividing the throughput by the PRU of the target FPGA. The results are shown in Figure 3.29. The relationship follows a very similar trend to the throughput of each kernel. Kernels *A* and *B* both use about the same area of the FPGA, and have similar throughputs. The area is about 7% less in both kernel *C* and *D* but the throughput is much higher. This shows that the hardware generated is much more efficient, and the effect of modifying the combined ρ and π step mapping was beneficial to speed and area of the design. In kernel *E* the effect of unrolling the message block loop by a factor of 2 greatly increased the area needed to contain the design while decreasing the throughput. In this kernel there is a high amount of hardware that is not being efficiently utilized due to data dependencies and stalling of the pipeline. The final kernel *F*, which implements a reprogrammable kernel for all the functions, did not use much more area that kernel *C* but had poor performance, resulting in a low efficiency.

The kernel that was chosen to be used as the proposed design was kernel *C* as it had the highest throughput. The AOCL tools managed to compile the kernel in 3 hours and 16 minutes.

Table 3.5: SHA-3 Kernel Comparison

Kernel	FPGA Throughput	Speedup	FPGA Utilization
A	87.68176	0.5314	35.17%
B	105.0503	0.6356	34.41%
C	4074.093	24.6697	28.57%
D	3821.666	23.1434	28.90%
E	1076.896	9.7829	47.21%
F	1347.537	8.1710	29.69%

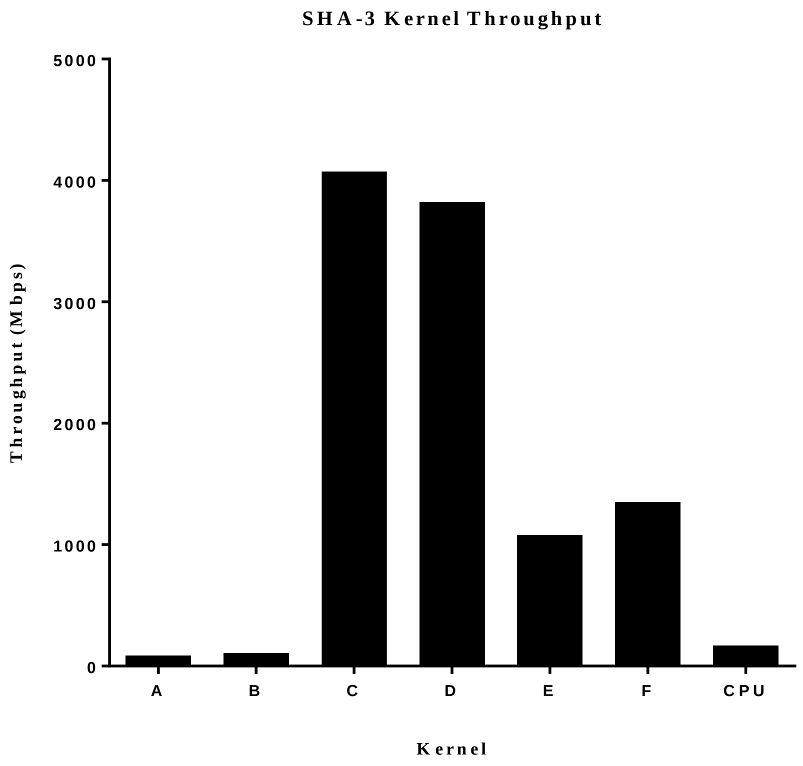

Figure 3.28: Comparison of SHA-3 Test Kernel Throughput

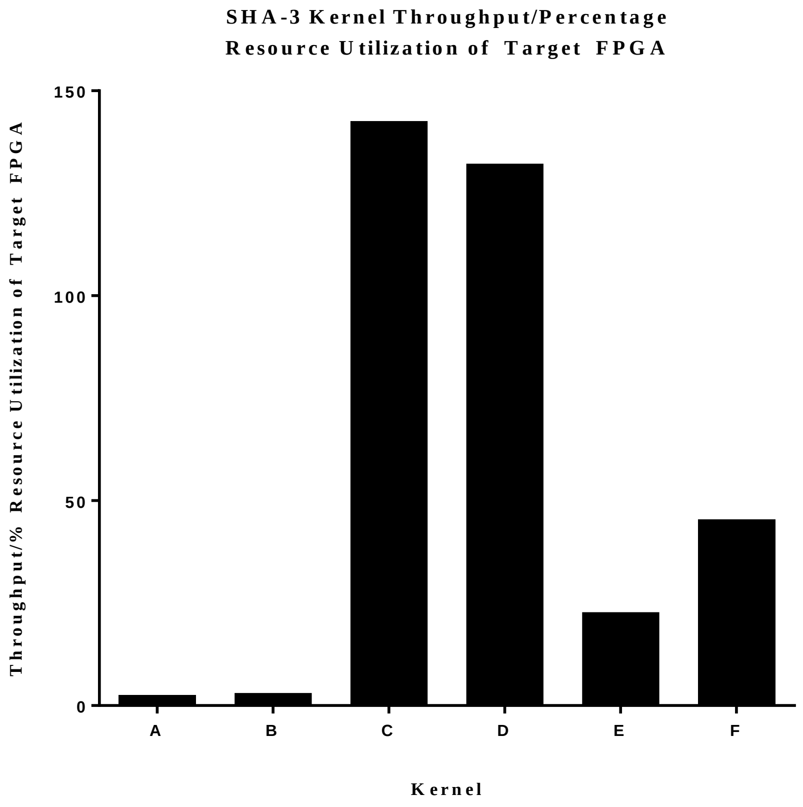

Figure 3.29: SHA-3 Test Kernel Throughput divided by PRU of Target FPGA

Once the best kernel was determined, the relationship between the message size and throughput could be determined. The same 372 input messages, ranging in size from 10B to 640MB, which were used in the SHA-1 and SHA-2 tests, were again used to characterize the kernel. The process of the algorithm is the same for each function in the SHA-3 standard, but the message block size does affect the performance. Because of this, separate kernels for SHA3-224, SHA3-256, SHA3-384, and SHA3-512 were compiled and tested. Each message was hashed six times with each of the functions to ensure a fair representation of the throughput. The behaviour of the functions all followed a similar trend, only differing in the steady throughput of large messages. The CPU model performs better than any of the FPGA implementations as long as the size is below 10^2 to 10^3 bytes. After that, the FPGA implementations have a higher throughput that increases with message size until about 10^7 bytes. The throughput of the implementation remains constant above that sized input. The SHA3-224 kernel is the fastest, reaching a throughput of 4075 Mbps. SHA3-256 is a bit slower, with 3960 Mbps. The SHA3-384 variant only managed to achieve a peak throughput of 3045 Mbps, while SHA3-512 maxed out at 2947 Mbps. These results are expected given the implementation of the algorithm used. The hardware used is the same for each SHA-3 function. The only difference in the calculation of each is the block size. As the digest size increases, the block size decreases, meaning that more message blocks are required to hash the same input between the functions. More message blocks lead to more iterations of the function and a slower time to compute the output. This characteristic of the implementation also affected the CPU model in the same way. The throughput of the SHA3-224 is the highest on CPU, and drops to less than half the throughput at SHA3-512. The throughput of the CPU design is 165 Mbps for SHA3-224, 156 Mbps for SHA3-256, 120 Mbps for SHA3-384, and 84 Mbps for SHA3-512. The relationship between FPGA and CPU throughput is shown in Figure 3.34.

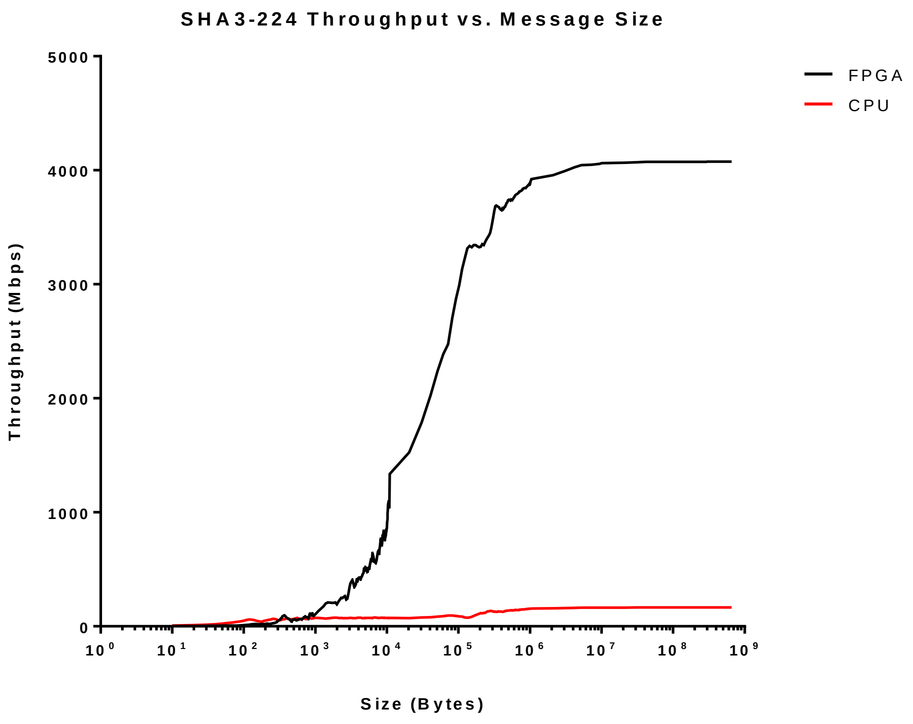

Figure 3.30: SHA3-224 Throughput vs. Message Size

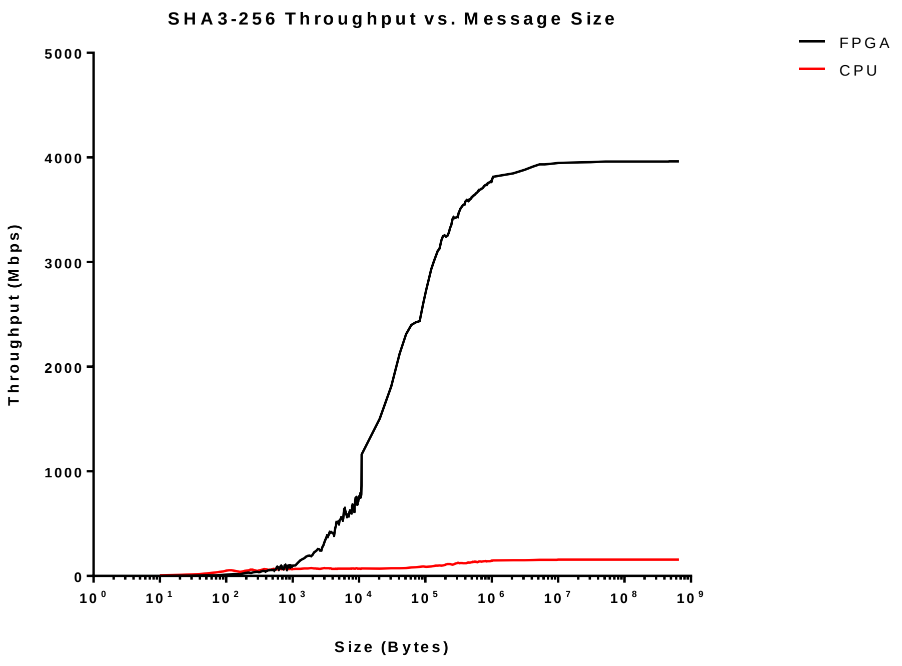

Figure 3.31: SHA3-256 Throughput vs. Message Size

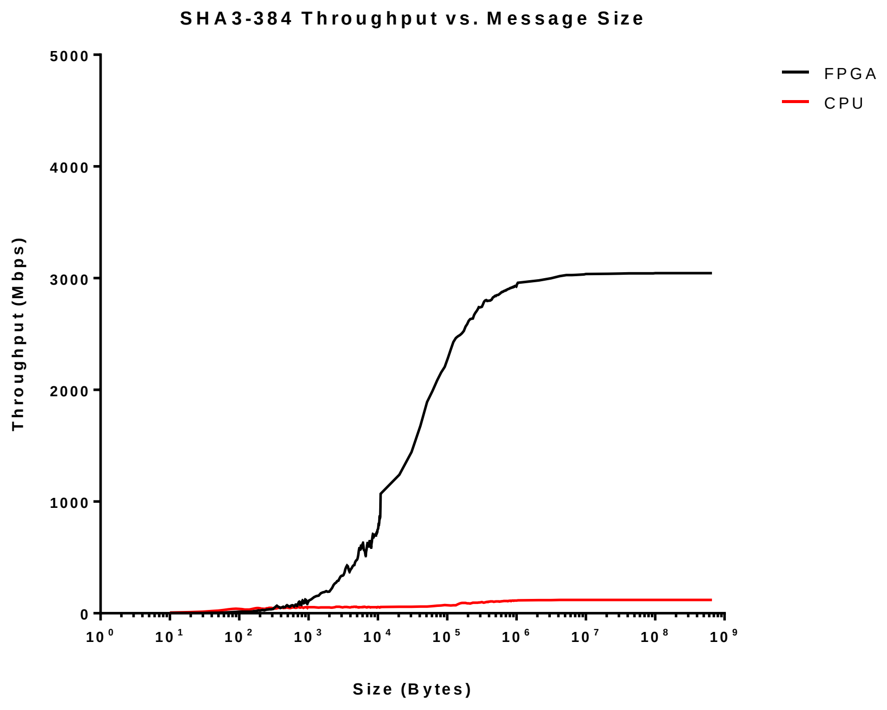

Figure 3.32: SHA3-384 Throughput vs. Message Size

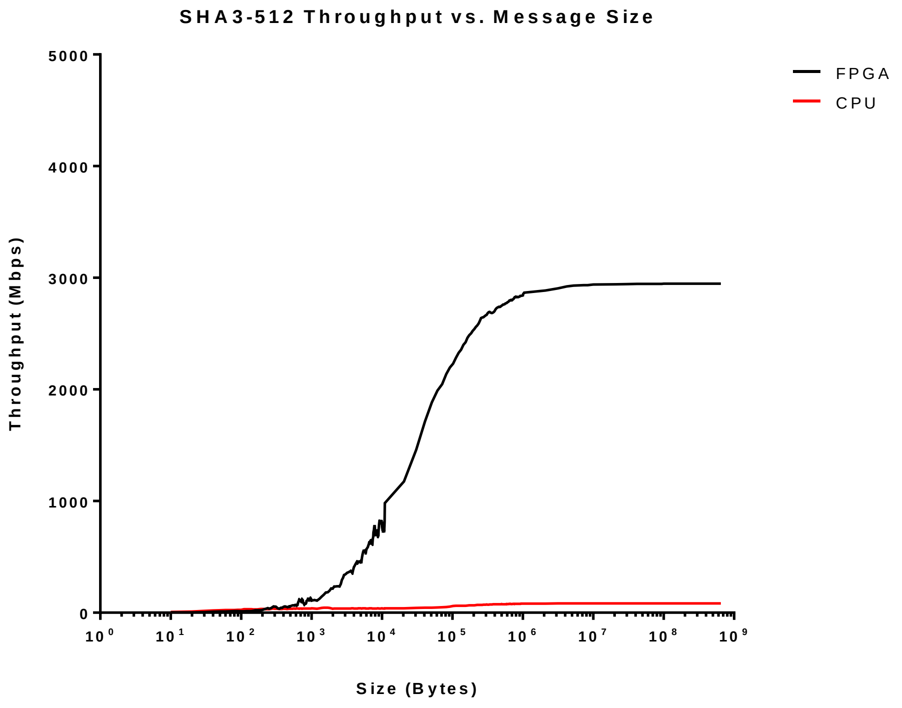

Figure 3.33: SHA3-512 Throughput vs. Message Size

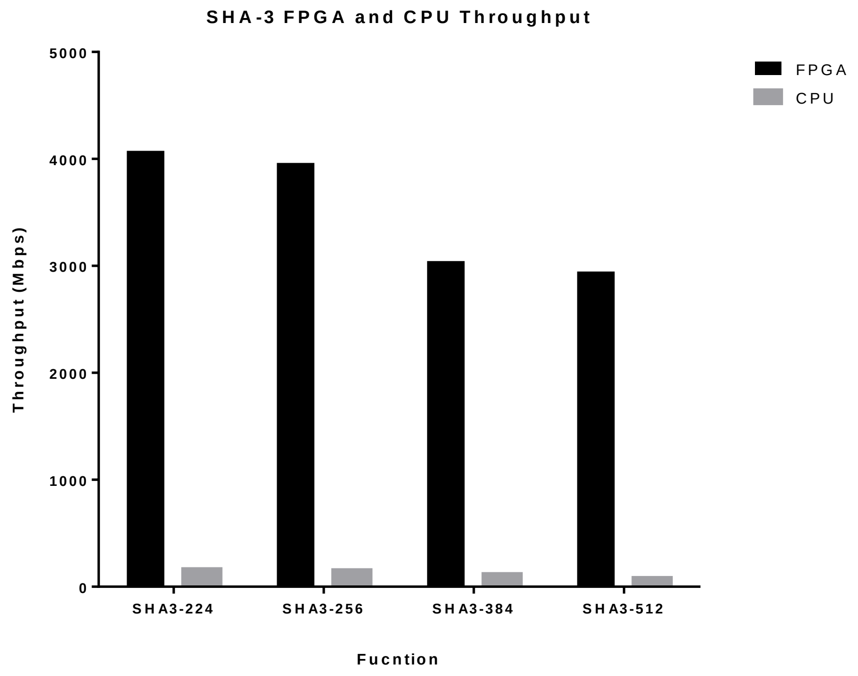

Figure 3.34: SHA-3 FPGA and CPU Throughput

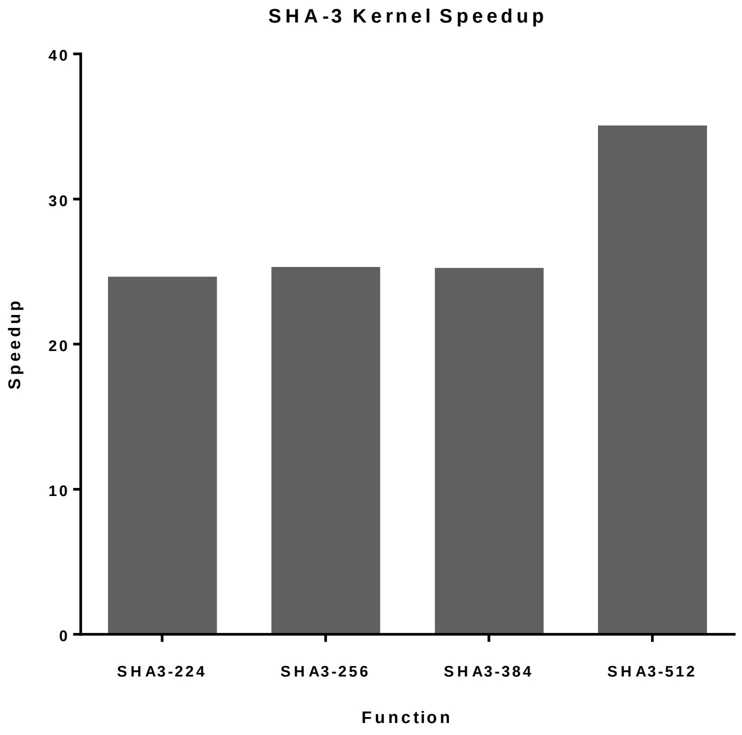

Figure 3.35: Comparison of SHA-3 Kernel Speedup

Although there is not as much work done in FPGA implementation of the SHA-3 as there was done to the SHA-1 and SHA-2 algorithms, the result of the proposed design was compared to the results in published literature. The HLS OpenCL model defined here did not have the same level of performance as comparable HDL base designs. Of the three designs compared, two were much faster. The only other authors to give speeds of the four fixed length digest functions was Baldwin *et al.* [35]. Looking at SHA3-224, the speed between their design and the proposed one are not drastically different. However as their designs increased in digest size, the performance also increase. This is the opposite relation of the AOCL accelerated algorithm as the performance decreased with an increase in digest size. The difference in speed is almost 3 fold between the two designs for SHA-512. The work done by Song *et al.* only implemented SHA3-256 and SHA3-512 [37]. Their SHA3-256 design was twice the speed of their SHA3-512 implementation, but was also 3.6 fold faster than the proposed HLS design. Jungk and Apfelbeck implemented just SHA3-256 [36]. This design was much slower at 864 Mbps, which is about 21% the speed of the proposed design. The HLS OpenCL model results are not comparable with the HDL models that have been published thus far. More work into optimization of the kernel would need to be done before it is possible to compete with HDL based models of the SHA-3 standard. Although the speed is not competitive, the results are still impressive. The speed that was achieved by the AOCL implementation is quite high and the design time and cost is greatly reduced from HDL based design methodologies.

Table 3.6: SHA-3 Performance Comparison to Related Work

Design	Throughput				Target
	SHA3-224	SHA3-256	SHA3-384	SHA3-512	
Baldwin *et al.* [35]	5915 Mbps	6232 Mbps	8190 Mbps	8518 Mbps	Xilinx Virtex-5
Jungk and Apfelbeck [36]	-	864 Mbps	-	-	Xilinx Virtex-6
Song *et al.* [37]	-	14438 Mbps	-	7066 Mbps	Xilinx Virtex-5
This Work	**4075 Mbps**	**3960 Mbps**	**3045 Mbps**	**2947 Mbps**	**Altera Stratix V**

3.5 Comparison

Each of the SHA functions were implemented and compared against their comparable equivalent from published literature but not to each other. There is significant difference in the throughput and area between the different secure hash functions. The reason for the variation between the functions is due to the structure of the algorithms themselves. The throughput comparison can be found in Figure 3.36, while the area comparison of the functions is in Figure 3.37**.**

The proposed SHA-1 design achieved a maximum throughput of 3033 Mbps. Since SHA-1 has be deprecated, this speed is only applicable to use in existing legacy applications. The SHA-256 throughput was a little less than half of the SHA-1 throughput at 1489 Mbps. The higher security of the algorithm and more complex operations that exist in the SHA-256 function compared to those in SHA-1 attribute to this reduction in speed. The results of the SHA-512 implementation were a bit higher than SHA-256, but still around half of SHA-1 at 1692 Mbps. The increased word size and block size of the SHA-512 function allows a higher throughput to be achieved compared to SHA-256. Both of the SHA-2 functions are still widely used in cryptographic applications. The

SHA-3 functions are meant to augment those in SHA-2 to provide substitute, rather than replacement functions, for similar applications. The throughput of the SHA3-224 function was determined to be 4075 Mbps. The comparable SHA-2 family algorithm would be SHA-224, which would have the same throughput as SHA-256. The SHA3-256 throughput achieved was 3960 Mbps. As discussed in section 3.4.3, the reason for the differences between the SHA-3 functions have to do with the block size of the algorithms. SHA3-384 had a maximum throughput of 3045 Mbps, while SHA3-512 achieved 2947 Mbps. The SHA-2 comparable function to SHA3-384 is SHA-384 which is the same algorithm as SHA-512 with a different initialization and a truncated output. The SHA-3 family of functions outperformed each of their SHA-2 comparable functions. In an applications needing a 224-bit secure digest, the SHA-3 implementation proposed could offer a 2.73 fold improvement in throughout over the SHA-2 equivalent. As the digest size increases, that boost in performance is not as drastic, but still present. A 512-bit digest can be calculated 1.74 times faster with the SHA-3 algorithm compared to the SHA-2 function.

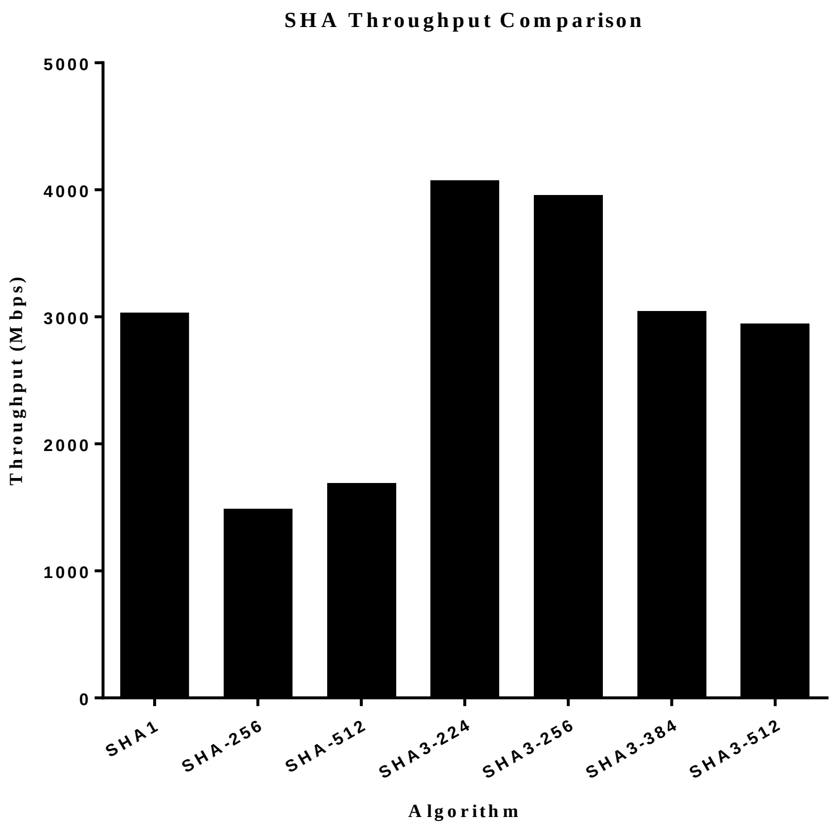

Figure 3.36: Throughput Comparison of All SHAs

The area of each SHA implementation was also compared. The SHA-1 design was implemented using 48030 LEs. That size was slightly reduced in SHA-256 to 42486 LEs. The reason for the decrease is that SHA-256 has fewer hashing rounds that that of SHA-1. The hardware must be more complex in SHA-256, but not as much is needed to implement the algorithm. For SHA-512, the area of SHA-256 is almost tripled. This is due to the increased number of rounds and the increased word size, both of which require a lot more hardware resources to implement. Moving to SHA3-224, the area needed to implement the function dropped down to 27689 LEs. The operations in the SHA-3 are less complex than those in SHA-1 and SHA-2 leading to a vast reduction in the area needed. Comparing to the equivalent SHA-2 function, the area needed for the design is only about 65% in the SHA-3 function. The area drops even further down for the other SHA-3 implementation, due to the smaller block size, less hardware is needed to copy the message block into the state. The size of the SHA3-512 is only 20135 LEs, which is about a third of the SHA-1 implementation with an almost equal throughput between the two. The difference compared to the SHA-512 function is a reduction in area by 5.7 times. The SHA-3 functions can provide a much smaller design than their comparable SHA-2 functions.

The reason for both the increase in throughput and reduction in area between the SHA-3 and SHA-2 functions is the design differences between the algorithms. The SHA-2 functions have trouble being parallelized due to the many dependencies between the algorithm steps. The inability to run many computations simultaneously hurts the acceleration of the algorithm. The nature of the operations is much more complex in SHA-2. These operations take a long time to calculated, and combined with the dependencies, stall the pipelines used in acceleration. In SHA-3, there is much less dependencies between the steps of the algorithm and the operations are simpler. The pipelines can work much more efficiently, and do not require as much hardware as the SHA-2 ones. Due to these factors, the SHA-3 design is faster and takes up less area than the SHA-2 equivalent.

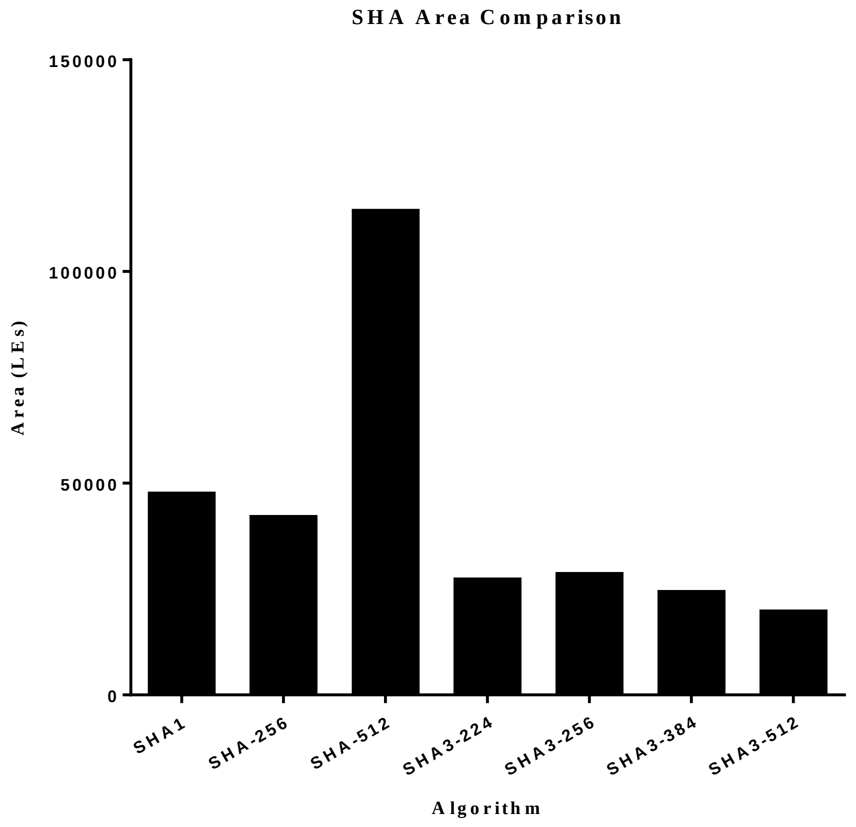

Figure 3.37: Area Comparison of All SHAs

3.6 Summary

This chapter gave a detailed description of each SHA algorithm implementation. This was followed by comprehensive analysis of the performance of each accelerated hash function. The FPGA accelerated OpenCL program was shown to give dramatic speedup over the CPU in all cases. The performance comparison to published designs showed very competitive speeds with the HLS model compared to HDL based design methods, especially with the decreased design time and cost. The area is much larger than HDL based designs, mainly because HDL based designs do not include area cost for interfacing FPGA accelerator to the host computer.

The differences in performance and area between the SHA variants were also compared. The SHA-3 family which was developed to augment the SHA-2 family to provide alternative functions for similar applications. The SHA-3 functions were shown to have faster performance with a lower area than the equivalent SHA-2 functions on this platform.

Chapter 4 Conclusion and Future Work

Secure hash functions are important for cryptographic applications. As digital communication speeds increase, the need to have fast performance of these algorithms is vitally important to maintain security. Through the use of the reprogrammable hardware of FPGAs, it is possible to accelerate these functions to speeds infeasible on CPU architectures.

There has been much work done in the area of FPGA acceleration of the SHAs defined in the secure hash standard. All of these implementations use a HDL based design methodology. To our knowledge, HLS tools have not been used to design and synthesize these algorithms. In this thesis, the AOCL tool was used to create and synthesize the SHA-1, SHA-2, and SHA-3 algorithms using a HLS design approach. OpenCL models were optimized to the extent possible to get the best results. These HLS models were then compared to the related work done with HDL design techniques. The performance of each implementation was evaluated in great detail. The results obtained in the proposed designs had varying levels of competitiveness with published work. The SHA-1 and SHA-2 algorithms performed very comparably to their HDL designed counterparts. The SHA-3 algorithm models performance did not come as close to published results, albeit the performance was much better than the comparable algorithms of the SHA-2 family. In addition to throughput performance, the area of each implementation was also evaluated. The size of the HLS models are much larger than the equivalent HDL designs. This was due to the host interface hardware included in the AOCL design that was not needed in the HDL designs. Further evaluation of the proposed SHA models was performed including rapid design space exploration of the implementations to optimize the results. This would be a very time consuming process for HDL based design.

4.1 Future Work

There is still much work that can be done in this area. The biggest focus would be on further acceleration of the SHA-3 algorithms. The difference in performance between the HLS and HDL designed models is significant. By studying the underlying functions further, it may be possible to create a new HLS model that can have comparable results to the related work. Another area to explore in this research is implementation of the AOCL design in accelerator cards containing the higher performing Arria 10 FPGA to determine the effect of utilizing a faster FPGA. The Cyclone V SoC platform is another platform that could be tested to analyze the potential of using this model in embedded systems. There are also other HLS tools that exist that could be used to implement these designs to compare the performance of the AOCL in this application.

The design space of the SHAs were explored in this thesis, but much further evaluation can still be done to determine the effect of modifying all aspects of the kernel on throughput, area, or efficiency. With the HLS tools, this type of analysis is now possible, which was not able to be done practically using HDL design methods.

The use of dedicated hardware can benefit power consumption as well as throughput. There is an opportunity here to study the difference in power consumption between the FPGA accelerated program and the CPU only application. Adding the savings in power to the improvements in compute time can further extend the benefits of using AOCL HLS CAD tool.

Appendix A: SHA-1 Kernel Code

```
unsigned ROTL( int n, unsigned x){
    return ((x << n)) | ((x) >> (32-n));
}

unsigned choose (unsigned x, unsigned y, unsigned z){
    return (((x&y)^((~x)&z)));
}

unsigned parity (unsigned x, unsigned y, unsigned z){
    return ((x^y^z));
}

unsigned majority (unsigned x, unsigned y, unsigned z){
    return (((x&y)^(x&z)^(y&z)));
}

unsigned funct(unsigned x, unsigned y, unsigned z, int t){
    if ((0 <= t) && (t <= 19)){
        return choose(x,y,z);
    }
    else if (((20 <= t) && (t <=39)) || ((60 <= t) && (t <= 79))){
        return parity(x,y,z);
    }
    else {
        return majority(x,y,z);
    }
}
unsigned getK(int t){
    if ((0 <= t) && (t <= 19)){
        return 0x5a827999;
    }
    else if ((20 <= t) && (t <=39)){
        return 0x6ed9eba1;
    }
    else if ((40 <= t) && (t <=59)){
        return 0x8f1bbcdc;
    }
    else {
        return 0xca62c1d6;
    }
}

__kernel void sha1(__global unsigned char *restrict messBlock, __global unsigned *restrict hash,
int blockNum)
{
        unsigned H[5];
        unsigned W[80];
        unsigned a, b, c, d, e, T;
        #pragma unroll
        for (int i = 0; i < 5; i++){
                H[i]=hash[i];
        }



#pragma unroll 2
for (int l = 0; l < blockNum; l++){

   #pragma unroll
   for (int t = 0; t < 16; t++){
        W[t] = (((unsigned) messBlock[(l*64) + (t * 4)]) << 24)
             | (((unsigned) messBlock[(l*64) + (t * 4 + 1)]) << 16)
             | (((unsigned) messBlock[(l*64) + (t * 4 + 2)]) << 8)
             | (((unsigned) messBlock[(l*64) + (t * 4 + 3)]));
    }
   #pragma unroll
   for (int t = 16; t < 80; t++){
        W[t] = ROTL(1, (W[t-3] ^ W[t-8] ^ W[t-14] ^ W[t-16]));
    }

    //STEP 2. Initialize the working variables

    a = H[0];
    b = H[1];
    c = H[2];
    d = H[3];
    e = H[4];

    //STEP 3. 80 Rounds of HASHING

    #pragma unroll
    for (int t = 0; t < 80; t++){
        T = (ROTL(5, a)+ funct(b,c,d,t) + e + getK(t) + W[t]);
        e = d;
        d = c;
        c = ROTL(30, b);
        b = a;
        a = T;
    }

    //STEP 4. Update intermediate hashes

    H[0] += a;
    H[1] += b;
    H[2] += c;
    H[3] += d;
    H[4] += e;

    }

    #pragma unroll
    for (int i = 0; i <5; i++){
        hash[i] = H[i];
        }
}
```

Appendix B: SHA-256 Kernel Code

```
unsigned ROTR( int n, unsigned x){
        return ((x >> n)) | ((x) << (32-n));
}
unsigned SHR( int n, unsigned x){
        return ((x >> n));
}
unsigned choose (unsigned x, unsigned y, unsigned
z){
        return (((x&y)^((~x)&z)));
}
unsigned majority (unsigned x, unsigned y, unsigned
z){
        return (((x&y)^(x&z)^(y&z)));
}
unsigned SIGMA0(unsigned x){
        return ((ROTR(2, x)^ROTR(13, x)^ROTR(22,x)));
}
unsigned SIGMA1(unsigned x){
        return ((ROTR(6, x)^ROTR(11, x)^ROTR(25,x)));
}
unsigned sig0(unsigned x){
        return ((ROTR(7, x)^ROTR(18, x)^SHR(3,x)));
}
unsigned sig1(unsigned x){
        return ((ROTR(17, x)^ROTR(19, x)^SHR(10,x)));
}
__kernel void sha256(__global const unsigned char *restrict messBlock,
__global unsigned *restrict hash, int blockNum)
{
        unsigned char messCache[64];
        unsigned H[8], W[64], a, b, c, d, e, f, g, h, T1, T2;
        const unsigned long K[64] = {
                        0x428a2f98,  0x71374491,0xb5c0fbcf,  0xe9b5dba5,  0x3956c25b,  0x59f111f1,
                        0x923f82a4,  0xab1c5ed5,0xd807aa98,  0x12835b01,  0x243185be,  0x550c7dc3,
                        0x72be5d74,  0x80deb1fe,0x9bdc06a7,  0xc19bf174,  0xe49b69c1,  0xefbe4786,
                        0x0fc19dc6,  0x240ca1cc,0x2de92c6f,  0x4a7484aa,  0x5cb0a9dc,  0x76f988da,
                        0x983e5152,  0xa831c66d,0xb00327c8,  0xbf597fc7,  0xc6e00bf3,  0xd5a79147,
                        0x06ca6351,  0x14292967,0x27b70a85,  0x2e1b2138,  0x4d2c6dfc,  0x53380d13,
                        0x650a7354,  0x766a0abb,0x81c2c92e,  0x92722c85,  0xa2bfe8a1,  0xa81a664b,
                        0xc24b8b70,  0xc76c51a3,0xd192e819,  0xd6990624,  0xf40e3585,  0x106aa070,
                        0x19a4c116,  0x1e376c08,0x2748774c,  0x34b0bcb5,  0x391c0cb3,  0x4ed8aa4a,
                        0x5b9cca4f,  0x682e6ff3,0x748f82ee,  0x78a5636f,  0x84c87814,  0x8cc70208,
                        0x90befffa, 0xa4506ceb,0xbef9a3f7, 0xc67178f2};

        H[0] = 0x6a09e667;
        H[1] = 0xbb67ae85;
        H[2] = 0x3c6ef372;
        H[3] = 0xa54ff53a;
        H[4] = 0x510e527f;
        H[5] = 0x9b05688c;
        H[6] = 0x1f83d9ab;
        H[7] = 0x5be0cd19;

        for (int l = 0; l < blockNum; l++){
                int offset = l*64;
                #pragma unroll
                for (int z = 0; z < 64; z++){
                        messCache[z] = messBlock[z + offset];
                 }

                #pragma unroll
                for (int t = 0; t < 16; t++){
                        int t4 = t*4;
                        W[t] = (((unsigned) messCache[(t4)]) << 24) | (((unsigned)
                        messCache[(t4 + 1)]) << 16) | (((unsigned) messCache[(t4 + 2)]) << 8)
                        |(((unsigned) messCache[(t4 + 3)]));
                }
                #pragma unroll
                for (int t = 16; t < 64; t++){
                        W[t] = (sig1(W[t-2])+W[t-7]+sig0(W[t-15])+W[t-16]);
                }
                a = H[0];
                b = H[1];
                c = H[2];
                d = H[3];
                e = H[4];
                f = H[5];
                g = H[6];
                h = H[7];
                #pragma unroll
                for (int t = 0; t < 64; t++){
                        T1 = (h + SIGMA1(e) + choose(e,f,g) + K[t] + W[t]);
                        T2 = (SIGMA0(a) + majority(a,b,c));
                        h = g;
                        g = f;
                        f = e;
                        e = (d + T1);
                        d = c;
                        c = b;
                        b = a;
                        a = (T1 + T2);
                }
                H[0] += a;
                H[1] += b;
                H[2] += c;
                H[3] += d;
                H[4] += e;
                H[5] += f;
                H[6] += g;
                H[7] += h;
        }
        #pragma unroll
        for (int i = 0; i <8; i++){
                hash[i] = H[i];
        }
}
```

Appendix C: SHA-512 Kernel Code

```
unsigned long ROTR( int n, unsigned long x){
        return ((x >> n)) | ((x) << (64-n));
}
unsigned long SHR( int n, unsigned long x){
        return ((x >> n));
}
unsigned long choose (unsigned long x, unsigned long y, unsigned long
z){
        return (((x&y)^((~x)&z)));
}
unsigned long majority (unsigned long x, unsigned long y, unsigned long
z){
        return (((x&y)^(x&z)^(y&z)));
}
unsigned long SIGMA0(unsigned long x){
        return ((ROTR(28, x)^ROTR(34, x)^ROTR(39,x)));
}
unsigned long SIGMA1(unsigned long x){
        return ((ROTR(14, x)^ROTR(18, x)^ROTR(41,x)));
}
unsigned long sig0(unsigned long x){
        return ((ROTR(1, x)^ROTR(8, x)^SHR(7,x)));
}
unsigned long sig1(unsigned long x){
        return ((ROTR(19, x)^ROTR(61, x)^SHR(6,x)));
}
__kernel void sha512(__global const unsigned char *restrict messBlock,
__global unsigned long *restrict hash, int blockNum)
{
        unsigned char messCache[128];
        unsigned long W[80], H[8], a, b, c, d, e, f, g, h, T1, T2;
        const unsigned long long K[80] = {
                0x428a2f98d728ae22,  0x7137449123ef65cd,  0xb5c0fbcfec4d3b2f,  0xe9b5dba58189dbbc,
                0x3956c25bf348b538,  0x59f111f1b605d019,  0x923f82a4af194f9b,  0xab1c5ed5da6d8118,
                0xd807aa98a3030242, 0x12835b0145706fbe, 0x243185be4ee4b28c, 0x550c7dc3d5ffb4e2,
                0x72be5d74f27b896f,  0x80deb1fe3b1696b1,  0x9bdc06a725c71235,  0xc19bf174cf692694,
                0xe49b69c19ef14ad2,  0xefbe4786384f25e3,  0x0fc19dc68b8cd5b5,  0x240ca1cc77ac9c65,
                0x2de92c6f592b0275, 0x4a7484aa6ea6e483, 0x5cb0a9dcbd41fbd4, 0x76f988da831153b5,
                0x983e5152ee66dfab,  0xa831c66d2db43210,  0xb00327c898fb213f,  0xbf597fc7beef0ee4,
                0xc6e00bf33da88fc2,  0xd5a79147930aa725,  0x06ca6351e003826f,  0x142929670a0e6e70,
                0x27b70a8546d22ffc, 0x2e1b21385c26c926, 0x4d2c6dfc5ac42aed, 0x53380d139d95b3df,
                0x650a73548baf63de,  0x766a0abb3c77b2a8,  0x81c2c92e47edaee6,  0x92722c851482353b,
                0xa2bfe8a14cf10364,  0xa81a664bbc423001,  0xc24b8b70d0f89791,  0xc76c51a30654be30,
                0xd192e819d6ef5218, 0xd69906245565a910, 0xf40e35855771202a, 0x106aa07032bbd1b8,
                0x19a4c116b8d2d0c8,  0x1e376c085141ab53,  0x2748774cdf8eeb99,  0x34b0bcb5e19b48a8,
                0x391c0cb3c5c95a63,  0x4ed8aa4ae3418acb,  0x5b9cca4f7763e373,  0x682e6ff3d6b2b8a3,
                0x748f82ee5defb2fc, 0x78a5636f43172f60, 0x84c87814a1f0ab72, 0x8cc702081a6439ec,
                0x90befffa23631e28,  0xa4506cebde82bde9,  0xbef9a3f7b2c67915,  0xc67178f2e372532b,
                0xca273eceea26619c,  0xd186b8c721c0c207,  0xeada7dd6cde0eb1e,  0xf57d4f7fee6ed178,
                0x06f067aa72176fba, 0x0a637dc5a2c898a6, 0x113f9804bef90dae, 0x1b710b35131c471b,
                0x28db77f523047d84,  0x32caab7b40c72493,  0x3c9ebe0a15c9bebc,  0x431d67c49c100d4c,
                0x4cc5d4becb3e42b6, 0x597f299cfc657e2a, 0x5fcb6fab3ad6faec, 0x6c44198c4a475817};

        H[0] = 0x6a09e667f3bcc908;
        H[1] = 0xbb67ae8584caa73b;
        H[2] = 0x3c6ef372fe94f82b;
        H[3] = 0xa54ff53a5f1d36f1;
        H[4] = 0x510e527fade682d1;
        H[5] = 0x9b05688c2b3e6c1f;
        H[6] = 0x1f83d9abfb41bd6b;
        H[7] = 0x5be0cd19137e2179;

        for (int l = 0; l < blockNum; l++){
                int offset = l*128;
                #pragma unroll
```

```
            for (int z = 0; z < 128; z++){
                    messCache[z] = messBlock[z + offset];
            }
            #pragma unroll
            for (int t = 0; t < 16; t++){
                    int t4 = t*8;
                    W[t]  =  (((unsigned  long)  messCache[(t4)])  <<  56)  |  (((unsigned  long)
                    messCache[(t4 + 1)]) << 48) | (((unsigned long) messCache[(t4 + 2)]) <<
                    40) | (((unsigned long) messCache[(t4 + 3)]) << 32) |
                    (((unsigned  long)  messCache[(t4  +  4)])  <<  24)  |  (((unsigned  long)
                    messCache[(t4 + 5)]) << 16) | (((unsigned long) messCache[(t4 + 6)]) <<
                    8) | (((unsigned long) messCache[(t4 + 7)]));
            }
            #pragma unroll
            for (int t = 16; t < 80; t++){
                    W[t] = (sig1(W[t-2])+W[t-7]+sig0(W[t-15])+W[t-16]);
            }
            a = H[0];
            b = H[1];
            c = H[2];
            d = H[3];
            e = H[4];
            f = H[5];
            g = H[6];
            h = H[7];
            #pragma unroll
            for (int t = 0; t < 80; t++){
                    T1 = (h + SIGMA1(e) + choose(e,f,g) + K[t] + W[t]);
                    T2 = (SIGMA0(a) + majority(a,b,c));
                    h = g;
                    g = f;
                    f = e;
                    e = (d + T1);
                    d = c;
                    c = b;
                    b = a;
                    a = (T1 + T2);
            }
            H[0] += a;
                    H[1] += b;
                    H[2] += c;
                    H[3] += d;
                    H[4] += e;
                    H[5] += f;
                    H[6] += g;
                    H[7] += h;
    }
    #pragma unroll
    for (int i = 0; i <8; i++){
            hash[i] = H[i];
    }
}
```

Appendix D: SHA-3 Kernel Code

```
unsigned long ROTL(  unsigned long x, int n){
        return ((x << n)) | ((x) >> (64-n));
}

int mod5(int x){
        while (x > 4){
                x -= 5;
        }
        return x;
}


__kernel void sha3(__global const unsigned char *restrict PM, __global unsigned long *restrict
hash, int blockNum)
{

        int blockSize = 144; //for SHA3-224. Replace with 136 for SHA3-256, 104 for SHA3-384, and
72 for SHA3-512.

        unsigned long A[25], B[25], C[5], D[5];

        unsigned long iota[24] =
                        {
                        0x0000000000000001, 0x0000000000008082, 0x800000000000808a,
                        0x8000000080008000, 0x000000000000808b, 0x0000000080000001,
                        0x8000000080008081, 0x8000000000008009, 0x000000000000008a,
                        0x0000000000000088, 0x0000000080008009, 0x000000008000000a,
                        0x000000008000808b, 0x800000000000008b, 0x8000000000008089,
                        0x8000000000008003, 0x8000000000008002, 0x8000000000000080,
                        0x000000000000800a, 0x800000008000000a, 0x8000000080008081,
                        0x8000000000008080, 0x0000000080000001, 0x8000000080008008
                        };

        for (int i = 0; i < 25; i++){
                A[i]= 0x00;
        }

        for (int l = 0; l < blockNum; l++){

                int offset = blockSize*l;
                #pragma unroll
                for (int k = 0; k < blockSize/8; k++){
                        A[k] ^= (((unsigned long) PM[(k*8) + offset]) |
                        (((unsigned long) PM[k*8 +1 + offset]) << 8) |
                        (((unsigned long) PM[k*8 +2 + offset]) << 16) |
                        (((unsigned long) PM[k*8 +3 + offset]) << 24) |
                        (((unsigned long) PM[k*8 +4 + offset]) << 32) |
                        (((unsigned long) PM[k*8 +5 + offset]) << 40) |
                        (((unsigned long) PM[k*8 +6 + offset]) << 48) |
                        (((unsigned long) PM[k*8 +7 + offset]) << 56));

                }
                //#pragma unroll 2
                for (int roundNum = 0; roundNum < 24; roundNum++){


                        //Theta
                        #pragma unroll
                        for (int i = 0; i < 5; i++){
                                C[i] = A[i]^A[5+i]^A[10+i]^A[15+i]^A[20+i];
                        }
                        #pragma unroll
                        for (int i = 0; i < 5; i++){
```

```
                        D[i] = C[mod5(i+4)] ^ ROTL(C[mod5(i+1)],1);
                }
                #pragma unroll
                for (int i = 0; i < 5; i++){
                        A[i] ^= D[i];
                        A[5+i] ^= D[i];
                        A[10+i] ^= D[i];
                        A[15+i] ^= D[i];
                        A[20+i] ^= D[i];
                }

                //Rho Pi

                B[0] = A[0];
                B[10] = ROTL(A[1] ,1);
                B[7] = ROTL(A[10] ,3);
                B[11] = ROTL(A[7] ,6);
                B[17] = ROTL(A[11] ,10);
                B[18] = ROTL(A[17] ,15);
                B[3] = ROTL(A[18] ,21);
                B[5] = ROTL(A[3] ,28);
                B[16] = ROTL(A[5] ,36);
                B[8] = ROTL(A[16] ,45);
                B[21] = ROTL(A[8] ,55);
                B[24] = ROTL(A[21] ,2);
                B[4] = ROTL(A[24] ,14);
                B[15] = ROTL(A[4] ,27);
                B[23] = ROTL(A[15] ,41);
                B[19] = ROTL(A[23] ,56);
                B[13] = ROTL(A[19] ,8);
                B[12] = ROTL(A[13] ,25);
                B[2] = ROTL(A[12] ,43);
                B[20] = ROTL(A[2] ,62);
                B[14] = ROTL(A[20] ,18);
                B[22] = ROTL(A[14] ,39);
                B[9] = ROTL(A[22] ,61);
                B[6] = ROTL(A[9] ,20);
                B[1] = ROTL(A[6] ,44);


                //Chi
                #pragma unroll
                for (int i = 0; i < 25; i+=5){
                        A[i] = B[i] ^(~(B[(i+1)])  & B[(i+2)]);
                        A[i+1] = B[i+1] ^(~(B[(i+2)])  & B[(i+3)]);
                        A[i+2] = B[i+2] ^(~(B[(i+3)])  & B[(i+4)]);
                        A[i+3] = B[i+3] ^(~(B[(i+4)])  & B[(i)]);
                        A[i+4] = B[i+4] ^(~(B[(i)])  & B[(i+1)]);

                }


                //iota

                A[0] ^= iota[roundNum];

        }

    }

    for (int i = 0; i < 25; i++){
            hash[i]= A[i];
    }

}
```

References

[1] LAN/MAN Standards Committee of the IEEE Computer Society, *IEEE Standard for Information Technology - Telecommunications and Information Exchange Between Systems - Local and Metropolitan Area Networks - Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specificatio*, vol. 2007, no. March. 2000.

[2] G. W. Wood and A. V Bhinge, "Network file update mechanism with integrity assurance." Google Patents, 08-Jan-2013.

[3] R. Chaves, G. Kuzmanov, L. Sousa, and S. Vassiliadis, "Improving SHA-2 Hardware Implementations," in *Cryptographic Hardware and Embedded Systems - CHES 2006 SE - 24*, vol. 4249, L. Goubin and M. Matsui, Eds. Springer Berlin Heidelberg, 2006, pp. 298–310.

[4] Xilinx Inc., "Accelerating Integration." [Online]. Available: http://www.xilinx.com/products/design-tools/vivado/integration.html. [Accessed: 18-Aug-2015].

[5] A. Canis, "High-Level Synthesis with LegUp," 2015. [Online]. Available: http://legup.eecg.utoronto.ca/. [Accessed: 18-Aug-2015].

[6] Altera Corporation, "Altera SDK for OpenCL - Overview," 2015. [Online]. Available: https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html. [Accessed: 18-Aug-2015].

[7] Khronos Group, “The OpenCL Specification 1.0,” *Khronos Gr. Specif.*, pp. 1–385, 2009.

[8] J. Rose, R. J. Francis, D. Lewis, and P. Chow, “Architecture of field-programmable gate arrays: The effect of logic block functionality on area efficiency,” *IEEE J. Solid-State Circuits*, vol. 25, no. 5, pp. 1217–1225, 1990.

[9] J. Wawrzynek, “CS150 Unit 6 Classnotes,” 2000. [Online]. Available: http://www-inst.eecs.berkeley.edu/~cs150/sp00/classnotes/u6.1/6_1_3.html. [Accessed: 18-Aug-2015].

[10] I. Tartalja and V. Milutinovic, “A survey of heterogeneous computing: concepts and systems,” *Proc. IEEE*, vol. 84, no. 8, pp. 1127–1144, 1996.

[11] Khronos Group, “About The Khronos Group,” 2015. [Online]. Available: https://www.khronos.org/about/. [Accessed: 11-Aug-2015].

[12] A. Munshi, “The OpenCL Specification 2.1,” *Khronos OpenCL Work. Gr.*, no. January, pp. 1–385, 2015.

[13] A. Munshi, “The OpenCL Specification 1.2,” p. 380, 2012.

[14] Nallatech, “OpenCL FPGA Cards by Nallatech,” 2015. .

[15] Terasic Inc., “Terasic - News & Events - OpenCL for Terasic DE5-Net Board,” 2015. [Online]. Available: http://www.terasic.com.tw/cgi-

bin/page/archive.pl?Language=English&CategoryNo=11&No=911. [Accessed: 14-Aug-2015].

[16] Bittware Inc., “OpenCL for Altera FPGAs - BittWare,” 2015. [Online]. Available: http://www.bittware.com/opencl-for-altera-fpgas. [Accessed: 14-Aug-2015].

[17] Nallatech, “Nallatech 510T - Nallatech,” 2015. [Online]. Available: http://www.nallatech.com/store/pcie-accelerator-cards/nallatech-510t-fpga-computing-acceleration-card/. [Accessed: 18-Aug-2015].

[18] Altera Corporation, “Cyclone V SoC Development Kit and SoC Embedded Design Suite,” 2015. [Online]. Available: https://www.altera.com/products/boards_and_kits/dev-kits/altera/kit-cyclone-v-soc.html. [Accessed: 28-Jun-2015].

[19] NIST, “Secure Hash Standard (SHS) (FIPS PUB 180-4),” no. March, 2012.

[20] NIST, “FIPS PUB 202 SHA-3 Standard : Permutation-Based Hash and,” no. August, 2015.

[21] W. Diffie and M. Hellman, “New directions in cryptography,” *IEEE Trans. Inf. Theory*, vol. 22, no. 6, 1976.

[22] M. Khasawneh, I. Kajman, R. Alkhudaidy, and A. Althubyani, “A Survey on Wi-Fi Protocols: WPA and WPA2,” in *Recent Trends in Computer Networks and Distributed Systems Security SE - 44*, vol. 420, G. Martínez Pérez, S. Thampi, R. Ko, and L. Shu, Eds. Springer Berlin Heidelberg, 2014, pp. 496–511.

[23] E. Barker and a Roginsky, “NIST Special Publication 800-131A Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths,” no. January, 2011.

[24] J.-S. Coron, Y. Dodis, C. Malinaud, and P. Puniya, “Merkle-Damgård Revisited: How to Construct a Hash Function,” in *Advances in Cryptology – CRYPTO 2005 SE - 26*, vol. 3621, V. Shoup, Ed. Springer Berlin Heidelberg, 2005, pp. 430–448.

[25] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, “The keccak reference,” *Submiss. to NIST (Round 3)*, pp. 1–14, 2011.

[26] B. Guido, “Cryptographic sponge functions,” pp. 1–93, 2011.

[27] N. Sklavos, G. Dimitroulakos, and O. Koufopavlou, “An ultra high speed architecture for VLSI implementation of hash functions,” *10th IEEE Int. Conf. Electron. Circuits Syst. 2003. ICECS 2003. Proc. 2003*, vol. 3, pp. 990–993, 2003.

[28] A. P. Kakarountas, G. Theodoridis, T. Laopoulos, C. E. Goutis, N. Sklavos, and C. Efstathiou, “High-Speed FPGA Implementation of the SHA-1 Hash Function,” *Intell. Data Acquis. Adv. Comput. Syst. Technol. Appl. 2005. IDAACS 2005. IEEE*, vol. 1, no. September, pp. 211–215, 2005.

[29] I. I. Yiakoumis, M. E. Papadonikolakis, H. E. Michail, A. P. Kakarountas, and C. E. Goutis, “Maximizing the hash function of authentication codes,” *Potentials, IEEE*, vol. 25, no. 2, pp. 9–12, 2006.

[30] E. H. Lee, J. H. Lee, I. H. Park, and K. R. Cho, “Implementation of high-speed SHA-1 architecture,” *IEICE Electron. Express*, vol. 6, no. 16, pp. 1174–1179, 2009.

[31] N. Sklavos and O. Koufopavlou, “On the hardware implementations of the SHA-2 (256, 384, 512) hash functions,” *Proc. 2003 Int. Symp. Circuits Syst. 2003. ISCAS ’03.*, vol. 5, no. 256, pp. 153–156, 2003.

[32] M. Khalil, M. Nazrin, and Y. W. Hau, “Implementation of SHA-2 hash function for a digital signature System-on-Chip in FPGA,” *2008 Int. Conf. Electron. Des. ICED 2008*, pp. 3–8, 2008.

[33] G. C. Michail H, Athanasiou G., Gregoriades A., Panagiotou L., “High Throughput Hardware/Software Co-Design Approach for SHA-256 Hashing Cryptographic Module In IPSec/IPv6,” *Glob. J. Comput. Sci. Technol.*, vol. 10, no. 4, pp. 54–59, 2010.

[34] H. Mestiri, F. Kahri, B. Bouallegue, and M. Machhout, “Efficient FPGA Hardware Implementation of Secure Hash Function SHA-2,” *Int. J. Comput. Netw. Inf. Secur.*, vol. 7, no. 1, pp. 9–15, 2014.

[35] B. Baldwin, A. Byrnet, L. Lu, M. Hamilton, N. Hanley, M. O’Neill, and W. P. Marnane, “FPGA implementations of the round two SHA-3 candidates,” *Proc. - 2010 Int. Conf. F. Program. Log. Appl. FPL 2010*, pp. 400–407, 2010.

[36] B. Jungk and J. Apfelbeck, “Area-efficient FPGA implementations of the SHA-3 finalists,” *Proc. - 2011 Int. Conf. Reconfigurable Comput. FPGAs, ReConFig 2011*, pp. 235–241, 2011.

[37] Q. Song, Y. Wang, Z. Li, Q. Zhou, W. Wu, D. Han, W. Xu, Z. Chen, and R. Li, “FPGA based optimized SHA-3 finalist in reconfigurable hardware,” *2011 Int. Symp. Integr. Circuits, ISIC 2011*, pp. 508–511, 2011.

[38] A. Imbewa and M. A. S. Khalid, “FLNR: A fast light-weight NoC router for FPGAs,” in *2013 IEEE 56th International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2013, pp. 445–448.

[39] Altera Corporation, “Stratix II vs . Virtex-4 Density Comparison,” no. August, pp. 1–19, 2005.

[40] F. Rivoallon, “Comparing Virtex-II and Stratix Logic Utilization,” vol. 161, pp. 1–6, 2002.

[41] L. Bai and S. Li, “VLSI implementation of high-speed SHA-256,” in *2009 IEEE 8th International Conference on ASIC*, 2009, pp. 131–134.

[42] R. P. McEvoy, F. M. Crowe, C. C. Murphy, and W. P. Marnane, “Optimisation of the SHA-2 family of hash functions on FPGAs,” *Proc. - IEEE Comput. Soc. Annu. Symp. Emerg. VLSI Technol. Archit. 2006*, vol. 2006, pp. 317–322, 2006.

[43] A. Mohamed and A. Nadjia, “SHA-2 Hardware core for Virtex-5 FPGA,” no. figure 2, pp. 3–7, 2015.

[44] J. H. J. He, H. C. H. Chen, and H. H. H. Huang, “A compatible SHA series design based on FPGA,” *Electr. Eng. Comput. Telecommun. Inf. Technol. (ECTI-CON), 2010 Int. Conf.*, 2010.

Vita Auctoris

NAME:	Ian Janik
PLACE OF BIRTH:	Leamington, ON
YEAR OF BIRTH:	1991
EDUCATION:	University of Windsor B.A.Sc. in Electrical Engineering Windsor, ON, 2013
	University of Windsor M.A.Sc. in Electrical Engineering Windsor, ON, 2015