Rochester Institute of Technology
RIT Digital Institutional Repository

Theses

8-1-2010

Cryptographic algorithm acceleration using CUDA enabled GPUs in typical system configurations

Maksim Bobrov

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

Recommended Citation
Bobrov, Maksim, "Cryptographic algorithm acceleration using CUDA enabled GPUs in typical system configurations" (2010). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

Cryptographic Algorithm Acceleration Using CUDA Enabled GPUs in Typical System Configurations

By

Maksim Bobrov

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

Supervised by

Dr. Roy Melton
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, NY
August, 2010

Approved By:

___

Dr. Roy Melton
Primary Advisor — Dept. of Computer Engineering

___

Dr. Marcin Lukowiak
Secondary Advisor — Dept. of Computer Engineering

___

Dr. Stanislaw P. Radziszowski
Secondary Advisor — Dept. of Computer Science

Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title: Cryptographic Algorithm Acceleration Using CUDA Enabled GPUs in Typical System Configurations

I, MAKSIM BOBROV, HEREBY GRANT PERMISSION TO THE WALLACE MEMORIAL LIBRARY TO REPRODUCE MY THESIS IN WHOLE OR PART.

____
Maksim Bobrov

____
Date

Acknowledgements

This research would not have been possible without the invaluable support and guidance of my advisor, Dr. Roy Melton. Special thanks go to my committee members, Dr. Marcin Lukowiak and Stanislaw P. Radziszowski. Additionally, I acknowledge the Rochester Institute of Technology for providing all the resources necessary to complete this research. Lastly, I would like to thank all those that helped and supported me in any respect throughout this process.

Abstract

The need to encrypt data is becoming more and more necessary. As the size of datasets continues to grow, the speed of encryption must increase to keep up or it will become a bottleneck. CUDA GPUs have been shown to offer performance improvements versus conventional CPUs for some data-intensive problems. This thesis evaluates the applicability of CUDA GPUs in accelerating the execution of cryptographic algorithms, which are increasingly used for growing amounts of data and thus will require significantly faster encryption and hashing throughput. Specifically, the CUDA environment was used to implement and experiment with three distinct cryptographic algorithms — AES, SHA-2, and Keccak — in order to show the applicability for various cryptographic algorithm classes. They were implemented in a system that emulates the conditions present in a real world environment, and the effects of offloading these tasks from the CPU to the GPU were assessed.

Speedups up to 2.6x relative to the CPU were seen for single-kernel AES, but SHA-2 and Keccak did not perform as well as on the GPU as on the CPU. Multi-kernel AES saw speedups over single-kernel AES up to 1.4x, 1.65x, and 1.8x for two, three, and four kernels, respectively. This translates to speedups between 3.6x and 4.7x over CPU implementations of AES. Introducing a CPU load had a minimal effect on throughput whereas a GPU load was seen to decrease throughput by as much as 4%. Overall, CUDA GPUs appear to have potential for improving encryption throughputs if a parallelizable algorithm is selected.

Table of Contents

THESIS RELEASE PERMISSION FORM .... **II**

ACKNOWLEDGEMENTS .... **III**

ABSTRACT .... **IV**

TABLE OF CONTENTS .... **V**

LIST OF FIGURES .... **VII**

LIST OF TABLES .... **IX**

CHAPTER 1 INTRODUCTION **1**

CHAPTER 2 MOTIVATION **3**

CHAPTER 3 CUDA PROGRAMMING MODEL **6**

CHAPTER 4 AES **13**

4.1. ALGORITHM OVERVIEW 13
4.2. GPU DESIGN AND IMPLEMENTATION 15
4.2.1 Direct Design 18
4.2.2 Lookup table Design 20
4.2.3 Implementation 21

CHAPTER 5 SHA-2 .... **22**

5.1. ALGORITHM OVERVIEW 22
5.2. GPU DESIGN AND IMPLEMENTATION 24

CHAPTER 6 KECCAK **26**

6.1. ALGORITHM OVERVIEW 26
6.2. GPU DESIGN AND IMPLEMENTATION 28

CHAPTER 7 TEST METHODOLOGY **31**

7.1. ALGORITHM GPU PERFORMANCE 31
7.2. TYPICAL SYSTEM LOADING EFFECTS 32

CHAPTER 8 RESULTS AND ANALYSIS **34**

8.1. SINGLE KERNEL GPU PERFORMANCE 34
8.1.1 AES 34
8.1.2 SHA-2 38
8.1.3 Keccak 39
8.2. MULTI-KERNEL GPU PERFORMANCE 40
8.3. TYPICAL SYSTEM LOADING EFFECTS 41
8.3.1 Offloading Effects 41
8.3.2 CPU Load Effects 43
8.3.3 GPU Load Effects 45

CHAPTER 9 CONCLUSIONS **49**

BIBLIOGRAPHY **51**

APPENDIX A AES CODE **53**

APPENDIX B SHA-2 CODE **58**

APPENDIX C KECCAK CODE **60**

APPENDIX D GRAPHS OF PERFORMANCE DATA **63**

List of Figures

Figure 1: NVIDIA GPU vs. Intel CPU Performance Comparison [5] 6
Figure 2: CUDA Thread Hierarchy [5].. 7
Figure 3: CUDA Hardware Architecture [4] 8
Figure 4: CUDA Process Flow 9
Figure 5: AES Encryption Algorithm 13
Figure 6: AES State Array [11] 14
Figure 7: Parallel AES in CTR Mode Implementation.. 16
Figure 8: S-Box Sub_bytes Implementation... 18
Figure 9: Column Byte Selection.. 20
Figure 10: SHA-2 Algorithm 23
Figure 11: Keccak State [20] 26
Figure 12: Sponge Construction [21].. 27
Figure 13: GPU Load Application 33
Figure 14: AES-128 Threads/Block Comparison 35
Figure 15: AES-192 Threads/Block Comparison 35
Figure 16: AES-256 Threads/Block Comparison 35
Figure 17: AES-128 Performance... 36
Figure 18: AES-192 Performance... 36
Figure 19: AES-256 Performance... 37
Figure 20: AES Speedups 37
Figure 21: SHA-256 Performance 38
Figure 22: SHA-512 Performance 39
Figure 23: Keccak-256 Performance 39
Figure 24: AES-256 Multi-Kernel Comparison 40
Figure 25: AES Offloading Effects... 42
Figure 26: SHA-2 Offloading Effects.. 42
Figure 27: Keccak Offloading Effects 43
Figure 28: AES-256 CPU Load Effects.. 44
Figure 29: SHA-512 CPU Load Effects 44
Figure 30: Keccak-512 CPU Load Effects 45
Figure 31: AES-256 GPU Load Effects on Total Time...................................... 46
Figure 32: SHA-512 GPU Load Effects on Total Time 46
Figure 33: Keccak-512 GPU Load Effects on Total Time 47
Figure 34: SHA-512 GPU Load Effects on GPU Time...................................... 47
Figure 35: Keccak-512 GPU Load Effects on GPU Time................................... 48
Figure 36: SHA-224 Performance 63
Figure 37: SHA-384 Performance 63

Figure 38: Keccak-224 Performance 64
Figure 39: Keccak-384 Performance 64
Figure 40: Keccak-512 Performance 64
Figure 41: AES-128 Multi-Kernel Comparison 65
Figure 42: AES-192 Multi-Kernel Comparison 65
Figure 43: AES CPU Load Effects 65
Figure 44: SHA-2 CPU Load Effects 66
Figure 45: Keccak CPU Load Effects 66
Figure 46: SHA-2 GPU Load Effects on GPU Time 67
Figure 47: Keccak GPU Load Effects on GPU Time 67
Figure 48: AES GPU Load Effects on Total Time 68
Figure 49: SHA-2 GPU Load Effects on Total Time 68
Figure 50: Keccak GPU Load Effects on Total Time 69

List of Tables

Table 1: SHA-2 Properties 22
Table 2: Keccak Properties 28

Chapter 1 Introduction

Vast amounts of data are stored and transmitted around the world at any given time. Some, if not most, of this data is private information which the owner may not want viewed publicly. To ensure that this does not happen, encryption algorithms are used. However, many of these algorithms were designed in the early age of computers when datasets were significantly smaller than they are now. Datasets which are now considered average may take long periods of time to encrypt or hash. It is in the interest of security to improve the performance of these algorithms.

The advent of the Compute Unified Device Architecture (CUDA) from NVIDIA and ATI's FireStream Technology has shifted Graphics Processing Units (GPUs) from primarily graphics enabling devices to general purpose stream processing devices. These devices are a cost effective alternative to traditional parallel processing machines with comparable performance. For certain applications, this change ushers in a new era in computing which allows any modern personal computer to take advantage of parallel processing capabilities previously available only in specialized systems.

This thesis attempts to leverage the benefits provided by the parallel architecture of CUDA GPUs in the field of cryptography. Three algorithms were selected for investigation; AES, SHA-2, and Keccak. CUDA implementations of each algorithm were designed and implemented. Where published results were available for GPU implementations, the algorithms in this thesis exhibited comparable performance. Each implementation was then evaluated with varying CPU and GPU loads ranging from completely unloaded to fully loaded. Finally, the results of these tests were analyzed to

determine the effects of offloading encryption and hashing algorithms from a CPU to a GPU.

AES, which has a high degree of parallelism, saw speedups as high as 2.6x for data sizes greater than 1 MB. SHA-2 and Keccak, on the other hand, have minimal parallelism and thus saw no speedup. Performing each encryption and hashing with a base CPU led to degradations of throughput up to 22%. A GPU load was found to decrease the throughput by as much as 36%. Although these slowdowns would be common in a typical system, the throughput may still be significantly faster when utilizing the GPU. Thus, it is believed that CUDA enabled GPUs could play a role as encryption and hashing accelerators in typical future systems.

The chapters that follow document the motivation, work, results and conclusions of this thesis. Chapter 2 has a detailed discussion of the encryption and hashing problem and why it should be investigated. Chapter 3 discusses the CUDA architecture, and Chapters 4 through 6 describe AES, SHA-2, and Keccak, respectively. The test methodology is discussed in Chapter 7, and the results of those tests are presented in Chapter 8. Final conclusions and suggestions for future work are made in Chapter 9.

Chapter 2 Motivation

Since their introduction, computers have permeated all facets of modern life and are now used for storing and transmitting information that is often of great value. To ensure this information is accessible only to those with the clearance to see it, encryption and hashing techniques have been established. However, dataset sizes today are much larger than they were a decade ago so it may take minutes, or even hours, to encrypt or hash what may now be considered a moderate dataset. This timing poses a problem to many users from a convenience perspective — users are often unwilling to wait or may not have the time — and, more importantly, it introduces a period of time during which the data are vulnerable, which should be reduced as much as possible.

One solution to this problem is to reduce the dataset size. However, in many cases the user does not have control (e.g., files produced by third party application). In cases where the user does have some control, it may be difficult to produce any significant reduction in size or may be undesirable to alter the data. Consequently, this solution is not viable for many cases.

Alternatively, the implementation of the encryption or hashing algorithm can be altered to execute faster. The case for parallelizing hash functions is made by Kaminsky and Radsiszowski in [1]. They argue in favor of designing the next generation of hash functions with a focus on potential for high throughput by way of parallelization. This focus is necessary for hashing large datasets in reasonable amounts of time as well as quickly hashing small datasets in specific applications such as on-the-fly hash message authentication. A similar argument can be made for ciphers, although many already exhibit significant parallelism.

Parallel implementations of encryption and hashing algorithms have seen success on various platforms including ASICs and FPGAs. FPGA implementations of AES are now capable of encryption throughputs as high as 9.22 GB/sec, more than 40 times faster than the average CPU based implementation [2]. ASIC implementations have produced speeds as high as 1.45 GB/sec, more than 6 times faster than the average CPU based implementation [3].

In contrast, CPU implementations have lagged behind as a direct result of their inherently serial nature. CPUs are best at performing sequential operations and are thus limited in their ability to take advantage of any parallelism. Multi-core CPUs have improved the CPUs ability to handle multiple threads of execution in parallel but are still restricted to small numbers of threads. Consequently, typical computer systems have traditionally relied on expensive expansion cards for high-speed encryption and hashing.

However, now all computers are equipped with some sort of GPU. The parallel processing limitations inherent in CPUs can now be compensated by utilizing a GPU capable of general purpose stream processing [4] [5]. Currently, two such GPU technologies exist — NVIDIA's CUDA [6] and ATI's Stream [7]. Of these two, CUDA is somewhat more developed and has had wider industry use [6] [7]. Thus, it has been selected as the target platform for this thesis.

The algorithms selected for investigation were AES, SHA-2, and Keccak. These were selected as they represent ciphers, the current generation of hash functions, and the future generation of hash functions. Additionally, they fall into three classes of parallelism; AES is highly parallel, Keccak is moderately parallel, and SHA-2 is minimally parallel. Detailed discussion and implementations of each algorithm follow in

chapters 4–6. The next chapter describes that NVIDIA CUDA platform targeted by this thesis.

Chapter 3 CUDA Programming Model

CUDA, developed by NVIDIA, is a highly parallel computing architecture implemented on newer models of NVIDIA GPUs [6]. Unlike traditional GPUs, CUDA GPUs are designed with greater focus on data processing as opposed to flow control and caching. They are capable of executing thousands of lightweight threads simultaneously with millions more queued. This high degree of parallelism leads to an immense increase in potential performance such that current generation GPUs can vastly outperform contemporary CPUs in certain applications. This comparison is visible in Figure 1. However, these benefits cannot be realized by all applications. CUDA is based on the stream processing model, which is an extension of the SIMD (single instruction, multiple data) paradigm. This design paradigm, as the name suggests, makes CUDA optimal for performing a single instruction many times, in parallel, on different sets of data. To better understand how the SIMD paradigm is a key aspect of CUDA, an investigation of the architecture is required. The discussion that follows is based on [4], [5], [6], and [8].

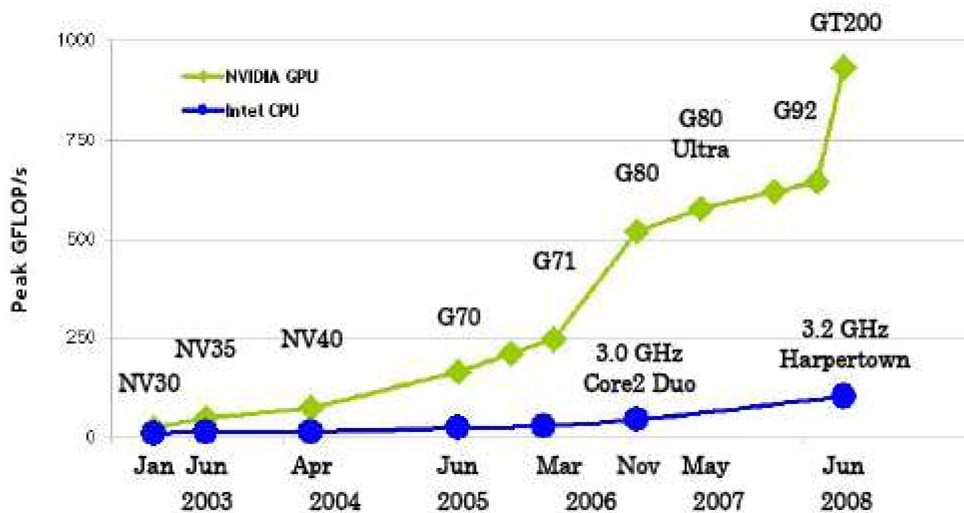

Figure 1: NVIDIA GPU vs. Intel CPU Performance Comparison [5]

The CUDA architecture consists of two main components — the memory and the processing cores — which work in conjunction and must be carefully considered when designing an application. Memory is divided into five categories — global, constant, textured, shared, and local — all of which have distinct features and can be accessed only by certain groups of threads. Processor cores are divided into a hierarchy of blocks of threads, and grids of blocks of threads. The thread hierarchy can be seen in Figure 2, and the overall architecture is shown in Figure 3.

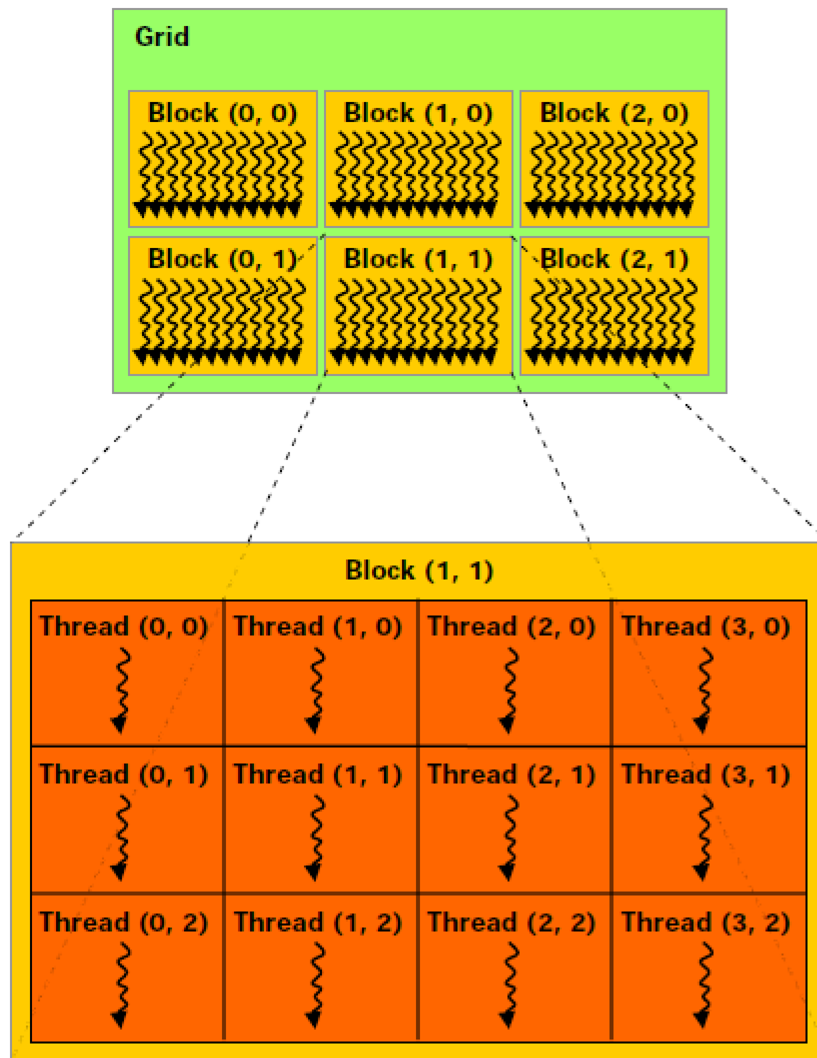

Figure 2: CUDA Thread Hierarchy [5]

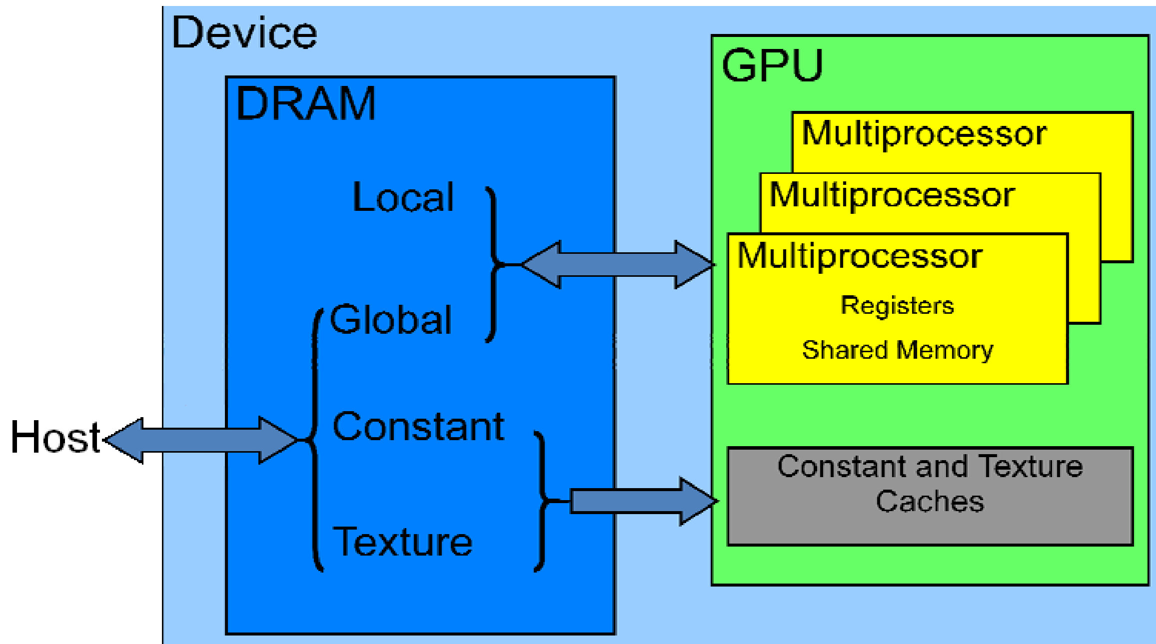

Figure 3: CUDA Hardware Architecture [4]

The most basic unit of execution in the CUDA architecture is the thread. Each thread is allocated a segment of local memory for local variables. Threads are grouped into 1-dimensional, 2-dimensional or 3-dimensional blocks (up to 512 threads per block), depending on application design. Each block has a unique section of shared memory allotted to it which can be accessed by all threads belonging to that block. The block is also the basic unit in terms of synchronization; all blocks execute independently, but within a block, all threads are executed simultaneously and can be synchronized using the synchthreads command. Finally, blocks are grouped into 1-dimensional or 2-dimensional grids. All grids have access to global memory, which includes constant memory and textured memory.

To facilitate software design, CUDA implements numerous extensions to the standard-C programming language (support for additional programming languages is supposed be added in the future). Applications are divided into two categories — code designed to execute on the host CPU and code designed to execute on the GPU. The code

that is to execute on the GPU is called the kernel. Communication between the CPU and GPU is achieved through memory reads and writes. Before executing a kernel, the data that are to be processed must be transferred to memory on the GPU. Next, the CPU initiates kernel execution on the GPU. Once execution is complete, the CPU retrieves the processed data from the GPU. This process is illustrated in Figure 4. Since communication between a CPU and its peripherals is relatively slow, the process of copying data back and forth can often be a major bottleneck. Therefore, an important aspect of an efficient CUDA implementation is the ability to overlap GPU communication from/to the CPU or from/to PC memory with GPU computation.

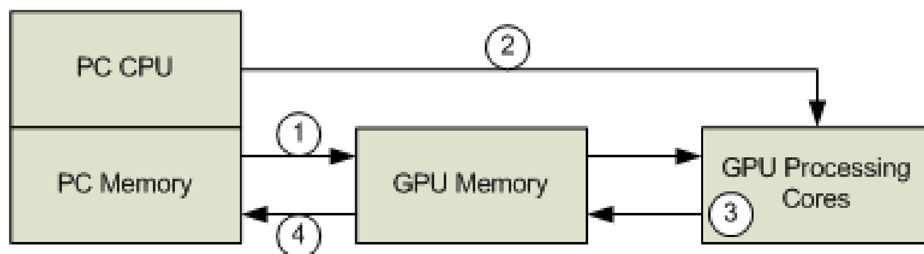

Figure 4: CUDA Process Flow

The performance enhancements obtainable with CUDA are heavily algorithm dependent but there are some general guidelines which can be followed. The overall goal is to maximize occupancy of the GPU such that each multiprocessor has the maximum possible number of threads queued at any time. There are two main factors which determine occupancy: register requirements per thread and shared memory usage.

As a thread executes, registers are allocated for that thread in order to store any variables that are not explicitly stored in memory. However, the quantity of this storage is limited per processor, and thus the number of registers available per thread is limited. For example, the GTX 285 contains 16 KB of registers per processor and can execute a maximum of 1024 threads simultaneously per multi-processor. Consequently, to achieve

a full occupancy of 1024 threads, each thread cannot use more than 16 registers. If the register utilization per thread were increased beyond 16, the number of threads executing would generally decrease. In the worst case scenario, some register values may be placed in local memory. This is extremely detrimental to throughput as local memory is actually a specially allocated portion of global memory, which is extremely slow.

If an application is found to utilize an excessive amount of registers or local memory, it may be necessary to move some data to other types of memory. Shared memory is the fastest memory available on the GPU and thus should be the primary memory used. Its speed is a result of its proximity to the processing cores — it is the CPU equivalent of cache but with complete user control. Additionally, all read and write operations to shared memory are guaranteed never to miss. However, the quantity of this memory is limited per processor. Since this memory is common to all blocks of a processing core, excessive usage of this type of memory can limit the number of active thread blocks per processing core. For example, the GTX 285 contains 16 KB of shared memory per multiprocessor. Executing 256 threads per block with each block utilizing 4 KB of shared memory will produce occupancy of 100% as this configuration will allow the maximum of 1024 threads to execute on each core. However, increasing the shared memory usage per block to 8 KB will allow for only 50% occupancy since the number of simultaneously active blocks is limited to two. If using a CUDA device with compute capability 2.0 or higher, it may not be necessary or desirable to obtain full occupancy. These devices are capable of executing multiple kernels simultaneously in parallel so it may be more valuable to leave some percentage of the GPU available for other tasks.

If shared memory is not sufficient, global memory must be used. This type of memory is significantly slower than shared memory but can be favorable if used correctly. There are two subsets of global of memory — constant and textured. Unlike global memory, constant, and textured memory are cached, greatly reducing the probability of a miss and thus increasing throughput. Furthermore, each type of memory is optimized for specific types of operations. Constant memory should be used when many threads perform simultaneous reads from a single memory location. Textured memory is preferred when memory accesses occur with either 1-D or 2-D spatial locality.

The final consideration when dealing with memory is how to maximize coalescing. Coalescing is the combination of multiple small memory accesses into several wide memory accesses. This combination takes place at the warp level. If the threads of a warp perform a memory access which does not utilize the full width of the data bus, it is possible that these memory accesses may be combined into a smaller set of larger memory accesses. For example, the data bus on the GTX 285 is 64 bytes wide. If all 32 threads of a warp were each to perform a four-byte memory read, instead of performing 32 reads, these reads could be coalesced into two reads of 64 bytes to improve performance by a factor of 16. This optimization occurs only when the threads of a warp read memory in sequential order. An additional problem may arise if the memory access is not correctly aligned with memory. A CUDA word is 64 bytes; however, memory is partitioned into 128-byte blocks. If the memory access spans multiple words, both of which do not fall in the same 128-byte block, two memory accesses are required for a single word.

In addition to maximizing occupancy and properly utilizing memory, it is important to minimize the amount of branching per thread. Branching refers to any variations in execution paths amongst threads. Unlike a traditional CPU scheduler which schedules a single thread at a time, the CUDA scheduler schedules threads in “warps” of 32 threads. For a warp to be considered done and a new one scheduled in its place, all threads of that warp must finish executing. Thus, ideally all threads in a warp should take an equal amount of time to execute so that no warp thread waits idly for other threads in the warp to finish. This condition can be ensured by reducing or completely removing branching.

Finally, it is important to decrease the number and size of data transfers between the CPU and GPU as they are often the largest bottleneck when working with CUDA. Data transfers occur over the PCI Express bus which has a bandwidth of 16 GB/sec for the most recent generation (older generations are much slower at 4 GB/sec and 8 GB/sec). Meanwhile, the bandwidth of the memory on the 280GTX is approximately 130 GB/sec, more than eight times faster than the PCI Express bus. Thus, it is imperative that data transfers are minimized as much as possible. Future generations of CUDA devices may mitigate this issue by combining the CPU and GPU into a single device, which would effectively remove the need for data transfers all together [9].

The next chapters describe the encryption and hashing algorithms. The design and implementation of each algorithm with CUDA is discussed. Preliminary results used to guide the final implementation are also presented and discussed.

Chapter 4 AES

The Advanced Encryption Standard (AES) was first introduced in 1998 as the Rijndael algorithm. In 2001, after a five-year standardization process, it was selected by the National Institute of Standards and Technology (NIST) as the most suitable replacement for its predecessor, the Data Encryption Standard (DES) [10]. Since then, it has become one of the most popular encryption algorithms and is likely the most widely used block cipher today. It has been selected as a candidate for this thesis as a direct result of its popularity — improving the performance of this algorithm may be very beneficial to its future users.

4.1. Algorithm Overview

```
AES(plaintext, ciphertext, key){
        state = init_state(plaintext)

        add_round_key(state, key)

        for(num_rounds){
                sub_bytes(state)
                shift_rows(state)
                mix_columns(state)
                add_round_key(state, key)
        }

        sub_bytes(state)
        shift_rows(state)
        add_round_key(state, key)

        ciphertext = state
}
```

Figure 5: AES Encryption Algorithm

AES is based on the principles of substitution-permutation networks (SP networks); the process of encryption utilized by AES is roughly divided into two major

functions: substitution and permutation [11]. These substitutions and permutations, along with key based operations, are performed multiple times on the input data to obtain the encrypted ciphertext. An overview of this algorithm is presented in Figure 5.

To begin, the plaintext to be encrypted is divided into 16-byte segments. As a block cipher, AES operates on fixed-length blocks of data. These blocks are then converted into a 4x4 array of bytes, known as the state, as illustrated in Figure 6. The shading signifies grouping of bytes in columns.

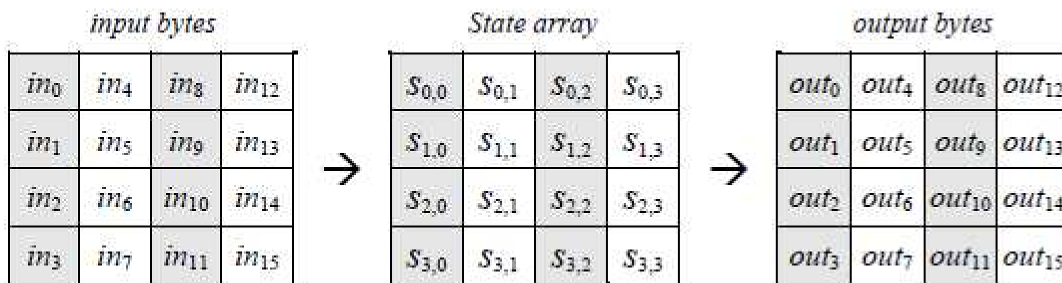

Figure 6: AES State Array [11]

The substitution portion of the cipher, referred to as sub_bytes in Figure 5, is a simple replacement of each byte in the array with its entry in a fixed 8-bit substitution box (s-box), as in the Rijndael s-box. This value can also be calculated mathematically.

The permutation portion of the cipher is split into two functions: shift_rows and mix_columns in Figure 5. Shift_rows performs a row permutation in the form of a left-circular shift on each of the four rows of the state starting with a zero shift for the top row and increasing by one for each consecutive row. Next, mix_columns performs a column permutation which can best be described as a matrix multiplication as shown in (1). *C* is the pre mix_columns state and *C'* is the post mix_columns state. Alternatively, both shift_rows and mix_columns can be implemented as lookup tables.

$$\underbrace{\begin{bmatrix} C'_{00} & C'_{01} & C'_{02} & C'_{03} \\ C'_{10} & C'_{11} & C'_{12} & C'_{13} \\ C'_{20} & C'_{21} & C'_{22} & C'_{23} \\ C'_{30} & C'_{31} & C'_{32} & C'_{33} \end{bmatrix}}_{C'} = \underbrace{\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} C_{00} & C_{01} & C_{02} & C_{03} \\ C_{10} & C_{11} & C_{12} & C_{13} \\ C_{20} & C_{21} & C_{22} & C_{23} \\ C_{30} & C_{31} & C_{32} & C_{33} \end{bmatrix}}_{C} \quad (1)$$

Finally, the key is incorporated during add_round_key in Figure 5. At each round, the round key is determined using Rijndael's key schedule. It is then added to the state via a bitwise XOR.

To increase the security of block ciphers like AES, numerous modes of operation have been developed [12]. These modes modify the algorithm in order to ensure that identical message blocks encrypted at different positions in the plaintext with identical keys will not produce equal values. This thesis focuses on counter mode (CTR) which performs the AES encryption on a counter value and XORs the result of that with the corresponding message block to obtain the encrypted output. CTR mode presents two beneficial qualities which can be advantageous in a CUDA implementation. First, it preserves the block-level parallelism, which represents the bulk of the parallelism available in AES. And second, its encryption of a counter value instead of the plaintext provides the potential for reducing data transfers between the CPU and GPU; the final XOR can be performed on either the GPU or CPU.

4.2. GPU Design and Implementation

Many instances of parallelism are inherent in the AES algorithm [13][14]. A coarse-grained level of parallelism is provided by the independence of each 16-byte block, allowing all bytes of a block to be encrypted independently of each other. This independence is reduced for certain modes of operation, but those modes are not

considered in this thesis. Within each block encryption, each function within a round can be performed independently of other instances of that function: an s-box substitution can be performed on each byte independent of any other byte, each row can be shifted independent of any other row, each column can be mixed independent of any other column, and each word of the round key can be added independent of any other round word. To achieve the desired maximum occupancy, it is necessary to consider only the coarse grained parallelism provided by individual data blocks. Each CUDA thread completes the AES encryption on a single 16-byte block of data. All 16-byte blocks are computed in parallel. Implementing the aforementioned block level parallelism would be beneficial only for smaller datasets, which do not generate full occupancy of the GPU.

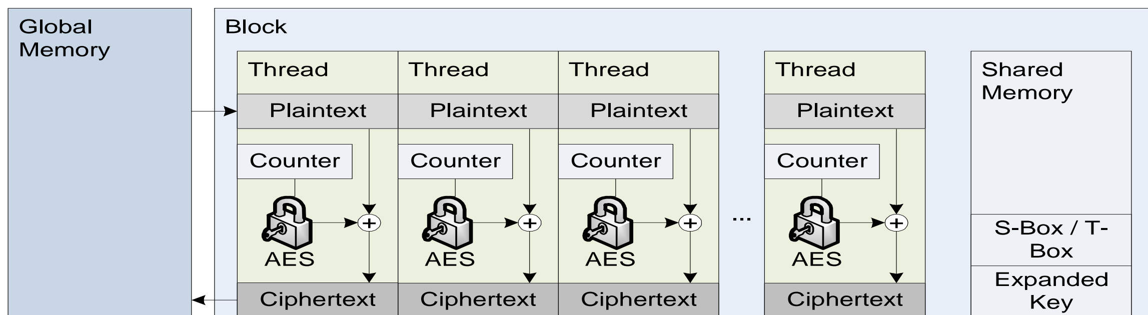

Figure 7: Parallel AES in CTR Mode Implementation

Two methods of implementing AES were investigated: a direct approach and one using lookup tables (LUTs). Figure 7 illustrates the overall approach to both designs; the two methods vary in the specifics of how they implement the AES step. All other steps are common to both implementations. To begin, Rijndael's key expansion algorithm is used to compute the key schedule. This computation is performed on the CPU so that each thread does not have to replicate this process as the key schedule is identical for every block. The key schedule is then copied to the GPU along with the data and any necessary LUTs. At this point, the dataset is partitioned into 16-byte data blocks; each

block is assigned a thread. The number of CUDA blocks is calculated based on the total number of threads and the chosen threads per block value (the details of this choice are further discussed in Chapter 8.1.1).

The kernel that is to perform the AES encryption is started with the calculated number of threads and blocks. The first thing each thread does is determine its unique identification number. This id number is used to determine the counter value required for CTR mode and to access the proper portion of the dataset. Next, the threads of each block collectively move all necessary LUTs from global memory to shared memory. Once the threads have transferred all the necessary LUTs, they are synchronized, and the counter value is encrypted.

The final step is to XOR the encryption of the counter with the corresponding data block. This operation reads the data directly from global memory and writes the result back to the same location in global memory. Shared memory is not used in this case since it does not remove the need to read and write global memory once and actually adds the need to read and write shared memory once. Once all threads have completed this process the kernel returns, and the CPU transfers the encrypted data from GPU memory to CPU memory.

It was found that a significant improvement in performance can be achieved by moving the final XOR operation from the GPU to the CPU. This removes the need to transfer the plaintext data from the CPU to the GPU effectively reducing data transfers by 50%. To obtain the full performance benefits, the kernel was divided into four mini-kernels. This division allowed for the pipelining of kernel execution, data transmission from the GPU to the CPU, and XORing with the plaintext.

4.2.1 Direct Design

This design took a direct approach to implementing AES. Each function is implemented individually as shown in Figure 5. Little effort is required to implement the shift_rows function as it can be done as a bitwise circular shift or, as it is done in this case, by simply remapping the registers. The add_round_key function is also very straightforward requiring a single XOR per column. The sub_bytes and mix_columns, however, required investigation of different design options.

Sub_bytes

The sub_bytes function can be implemented in one of two ways — using an s-box lookup table or calculating the value. Both approaches were implemented and tested to determine which would perform best.

The s-box approach requires the use of a lookup table, which requires multiple memory reads. This approach can be optimized by moving the table to shared memory as was done in this case. However, the use of lookup tables is discouraged as it increases the susceptibility of the implementation to side channel attacks [15]. Figure 8 shows the code to implement the s-box approach in CUDA C.

```
uint8_t __host__ __device__ subByte(uint8_t b, uint8_t *sbox){
      return sbox[b];
}
```

Figure 8: S-Box Sub_bytes Implementation

On the other hand, the calculation requires no memory lookups; it is done as a serious of basic arithmetic operations. Traditionally, the calculation is done using the extended Euclidian Algorithm to calculate the multiplicative inverse of a value and then performing an affine transformation to get the result. The problem with this technique is

in its use of the extended Euclidian Algorithm which is non-deterministic and will lead to an increase in branching.

An alternative method is to use composite field arithmetic to convert values from $GF(2^8)$ to $GF(((2^2)^2)^2)$ [16]. This is beneficial in two ways. First, operations in this field are simpler — addition is equivalent to XOR, and multiplication is equivalent to AND. Second, these computations are deterministic and, as such, will not result in branching. Both techniques were tested, and the s-box approach was actually found to be approximately ten times faster. Its better performance is likely because of the complexity of the implementation chosen for the calculation method, which required dozens of XOR, AND, and inverting operations. Alternative calculation implementations may exist which would perform better. As a result, the s-box approach is used in the final design.

Mix_columns

The other function which needed investigation was mix_columns. Performing the matrix multiplication would require a significant number of polynomial multiplications, which are both slow and non-deterministic. However, it is possible to take advantage of the simplicity of the polynomial multiplications required. Multiplying by 0x01 produces the original value, multiplying by 0x02 can be implemented as a bitwise left shift of one and an XOR with 0x1B if the most significant bit of the original value is a one [11], and multiplying by 0x03 can be implemented as multiplying by 0x02 and an XOR with the original value. Even with this improvement, mix_columns still remains the slowest part of the encryption consuming approximately half the time of the full algorithm. It is also the largest user of registers increasing register usage by more than 30% per thread.

4.2.2 Lookup table Design

This design condenses the functions of a round into four table looks ups and four XORs per column. The four necessary tables, referred to as T-boxes, are constructed as shown in (2). These tables are the results of performing the sub_bytes, and mix_columns operations of a round on every possible 8-bit value.

$$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} S[a_{0,j}] \\ S[a_{1,j-C1}] \\ S[a_{2,j-C2}] \\ S[a_{3,j-C3}] \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}$$

$$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = S\left[a_{0,j}\right] \begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} \oplus S\left[a_{1,j-C1}\right] \begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} \oplus S\left[a_{2,j-C2}\right] \begin{bmatrix} 01 \\ 03 \\ 02 \\ 01 \end{bmatrix} \oplus S\left[a_{3,j-C3}\right] \begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}$$

$$T_0[a] = \begin{bmatrix} S[a]\bullet 02 \\ S[a] \\ S[a] \\ S[a]\bullet 03 \end{bmatrix} \quad T_1[a] = \begin{bmatrix} S[a]\bullet 03 \\ S[a]\bullet 02 \\ S[a] \\ S[a] \end{bmatrix} \quad T_2[a] = \begin{bmatrix} S[a] \\ S[a]\bullet 03 \\ S[a]\bullet 02 \\ S[a] \end{bmatrix} \quad T_3[a] = \begin{bmatrix} S[a] \\ S[a] \\ S[a]\bullet 03 \\ S[a]\bullet 02 \end{bmatrix}$$

$$e_j = T_0[a_{0,j}] \oplus T_1[a_{1,j-C1}] \oplus T_2[a_{2,j-C2}] \oplus T_3[a_{3,j-C3}] \oplus k_j \qquad (2)$$

A round consists of using the four bytes of a column as indices into the T-boxes. The shift_rows operation is included in the selection of the bytes of each column as illustrated in Figure 9. Every four bytes of a single pattern represent the four bytes of a column. The results of these lookups are XORed along with the corresponding word of the round key to produce the new column.

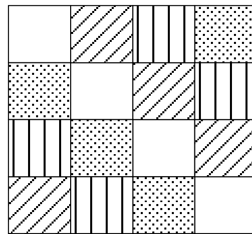

Figure 9: Column Byte Selection

These tables are constructed to function for the intermediate rounds, not the first and last rounds. These two rounds are performed a single function at a time in the same manner as the direct design and, in fact, use the optimized functions from that design.

To improve performance, the t-boxes, the s-box and the expanded key are moved from global memory to shared memory. All other operations are performed in registers.

Similar to the s-box of the direct design, the t-boxes increase the susceptibility of the implementation to side channel attacks [15]. However, they also provide potential for significant speedups. Whether security or throughput is more important is left up to the user.

4.2.3 Implementation

The direct design was found to outperform the LUT design by approximately 10% and was consequently used as the final design for testing. Appendix A provides the code for this implementation. Future modification to both implementations may change performance. Possible points of investigation include alternate sub_bytes and mix_columns implementations for the direct design. The LUT design could benefit from varying the memory type used as well as alignment in memory.

In addition to ciphers, hashing algorithms are often used for encryption. The next chapter describes the SHA-2 hashing algorithm as well as its GPU implementation.

Chapter 5 SHA-2

In 1993, NIST introduced the first version of the Secure Hash Algorithm (SHA) — SHA-0. Shortly thereafter, in 1995, SHA-0 was superseded by SHA-1 in order to improve its security. In 2001, four additional hash functions (SHA-224, SHA-256, SHA-384, and SHA-512), collectively known as SHA-2, were added to improve security further [10]. The SHA-2 family is currently considered one of the most secure hash algorithms, which makes it a good candidate for performance analysis.

5.1. Algorithm Overview

The SHA-2 functions utilize a single underlying algorithm with varying block sizes, word sizes, number of rounds (*Nr*), and message digest sizes [10][11]. These values can be seen in Table 1. Additionally, SHA-224 and SHA-256 use different word rotation functions and constants than SHA-384 and SHA-512 due to their different word sizes.

Table 1: SHA-2 Properties

Algorithm	Block Size (bits)	Word size (bits)	Number of rounds (*Nr*)	Message Digest Size (bits)
SHA-224	512	32	64	224
SHA-256	512	32	64	256
SHA-384	1024	64	80	384
SHA-512	1024	64	80	512

The underlying algorithm consists of two stages: preprocessing and hash computation. An overview of this algorithm can be seen in Figure 10.

```
SHA2(plaintext, digest)
        DM = preprocess(plaintext)
        for(N=sizeof(M)){
                W = message_schedule(M);

                A = M0; B = M1; C = M2; D = M3;
                E = M4; F = M5; G= M6; H = M4;

                for(t < Nr){
                        T1 = H + Σ1(E) + Ch(E,F,G) + Kt + Wt;
                        T2 = Σ0(A) + Maj(A,B,C);
                        H = G; G = F; F = E;
                        E = D + T1;
                        D = C; C = B; B= A;
                        A = T1 + T2;
                }

                M0 = A + M0; M1 = B + M1;
                M2 = C + M2; M3 = D + M3;
                M4 = E + M4; M5 = F + M5;
                M6 = G + M6; M7 = H + M7;
        }
}
```

Figure 10: SHA-2 Algorithm

Preprocessing prepares the message for hashing. First, the binary version of the message is padded with a 1 followed by *k* 0s and *n* bits containing the binary representation of the size, in bits, of the message. *k* is the number of bits required to make the message a multiple of the block size, and *n* is double the word size. Finally, the message is parsed into an array *M* of *N* blocks based on the block size of the chosen function.

Once preprocessing is complete, the hash computation can begin. The message is incorporated into the hash via the message schedule, W, which is calculated based on (3).

$$W_t = \begin{cases} M_t & 0 \le t < 16 \\ \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16} & 16 \le t < number_of_rounds \end{cases} \quad (3)$$

Next, the working variables *a, b, c, d, e, f, g*, and *h* are set to the hash values *H* of the previous block. In the case of the first round, the working variables are initialized to constant values based on the chosen function. The round function, seen in (4), is then executed *Nr* times.

$$
\begin{aligned}
&T_1 = H + \Sigma_1(A) + Ch(E, F, G) + K_t + W_t \\
&T_2 = \Sigma_0(A) + Maj(A, B, C) \\
&H = G \\
&G = F \\
&F = E \\
&E = D + T_1 \\
&D = C \\
&C = B \\
&B = A \\
&A = T_1 + T_2
\end{aligned}
\tag{4}
$$

Once the round function is complete, the working variables are XORed with the hash values of the previous round to produce the hash values for the current round. The hash computation step is repeated for each block of the message. The digest is the final result, *H*, of the last block processed.

5.2. GPU Design and Implementation

SHA-2 is a very serial algorithm and thus lacks any significant parallelism. Therefore, SHA-2 is not a good candidate for parallel speedup from SIMD GPU processing. Nevertheless, it may be beneficial to offload the hashing process from the CPU to the GPU, which could produce an overall speedup from the heterogeneous MIMD CPU and GPU processing because the CPU would be available for other tasks.

SHA-2 was implemented as shown in Figure 10. The code implementing the algorithm in CUDA C is shown in Appendix B. Since no significant parallelism exists,

the algorithm is implemented serially. Thus, no optimization for the CUDA architecture was performed. If multiple simultaneous hashes were desired, each independent hash computation could be done in parallel with CUDA. If enough such parallel hashes were performed, a benefit may be seen.

Although SHA-2 is still widely used, a competition is currently going on to find its replacement, which will be known as SHA-3. The next chapter discusses Keccak, one of the SHA-3 candidates.

Chapter 6 Keccak

As part of an effort to advance beyond current security methods, the NIST announced a competition in 2007 to find a successor to the current generation of hash algorithms [17]. A total of 61 algorithms were suggested for the first round of the competition. They were narrowed down to 14 in 2009. The final result of the competition will be known at the end of 2012. The algorithm selected will most likely replace the current generation of hashing algorithms. Thus, it is important to look at the performance of this future class of algorithms.

For the purpose of this thesis, Keccak [18] was selected as the representative SHA-3 algorithm. The SHA-3 candidates vary widely in their underlying structures, which makes it difficult to select one that perfectly represents all the candidates. Keccak was selected purely out of interest and the lack of published research into it. The only study of Keccak currently available is into its susceptibility to cube attacks [19].

6.1. Algorithm Overview

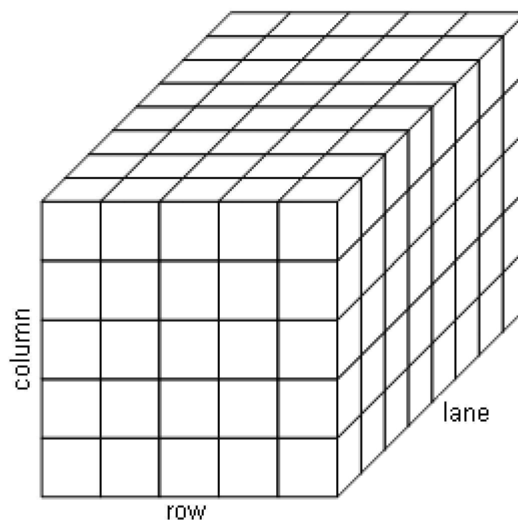

Figure 11: Keccak State [20]

Keccak is based on the sponge construction proposed by Bertoni, Daemen, Peeters, and Assche in [18]. The sponge construction consists of two steps: absorbing and squeezing. It operates on a state consisting of b=r+c bits where b is the state size, r (known as bitrate) is the predetermined block size, and c (known as capacity) is used to establish the security of the algorithm. The Keccak state is arranged in a 5x5 array of 64-bit lanes as show in Figure 11. In order to perform absorption and squeezing, the sponge construction requires a function, f, which is used to interleave the data. f can be any transformation or permutation but should be chosen carefully as security of the overall sponge is highly influenced by the function selected. The absorption phase absorbs the input message one r-bit block at a time. It incorporates each block into the state by XORing it with the state and then scrambling the result using f. Absorption continues until all blocks of the message have been absorbed at which point squeezing may begin. Squeezing, like absorption, uses the function, f, to scramble the data further. The number of squeezing steps is independent of all previous steps and can be varied based on desired output size. The sponge construction process explained above is concisely represented in Figure 12.

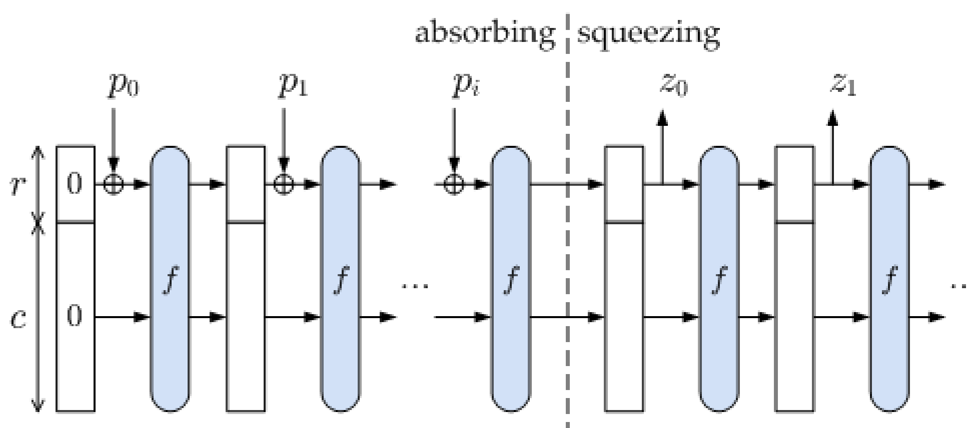

Figure 12: Sponge Construction [21]

Since Keccak is based on the sponge construction, it follows the same procedure of absorbing and squeezing utilizing a specific function *f*. The function selected is a permutation block cipher similar to AES but with some security improvements based on research done since the inception of AES [22][18]. The core of the function is represented as a series of equations shown in (5). *Rot* refers to a left-shift of some number of bits, *R* defines the shift value for each lane, and *RC* is a round constant. Seven variations of this function exist based on the desired state size, *b*. Acceptable values are *b*= 25 x 2*l*, where *l* is in the range 0 to 6. The complete function consists of 12+2*l* rounds of (5).

$$
\begin{array}{ll}
 & C[x] = A[x,0] \oplus A[x,1] \oplus A[x,2] \oplus A[x,3] \oplus A[x,4] \\
\theta: & D[x] = C[x-1] \oplus \mathrm{rot}(C[x+1],1) \\
 & A[x,y] = A[x,y] \oplus D[x] \\
\rho \text{ and } \pi: & B[y,2x+3y] = rot(A[x,y],R[x,y]) \\
\chi: & A[x,y] = B[x,y] \oplus (\sim B[x+1,y] + B[x+2,y]) \\
\iota: & A[0,0] = A[0,0] \oplus RC
\end{array} \qquad (5)
$$

The final Keccak specification lists four versions of Keccak with varying bitrate and capacity values. These are Keccak-224, Keccak-256, Keccak-384, and Keccak-512. Their respective bitrate and capacity can be found in Table 2. The number associated with each name corresponds to the output digest size in bits.

Table 2: Keccak Properties

	State Size (*b*)	Bitrate (*r*)	Capacity (*c*)	Output Length (*o*)
Keccak-224	1600	1152	448	224
Keccak-256	1600	1088	512	256
Keccak-384	1600	832	768	384
Keccak-512	1600	576	1024	512

6.2. GPU Design and Implementation

Keccak does not exhibit any coarse grained parallelism but does have some significant fine grained parallelism [18]. Every substitution and permutation of the round

function within a round can be done independent of the rest. Full occupancy is not attainable with this limited parallelism, but a CUDA implementation should still be able to produce speedups as high as 25x if all 25 lane calculations can be performed simultaneously. This is the approach attempted by this thesis. The CUDA C implementation can be seen in Appendix C.

Like with the previous algorithms, the data are first padded and then transferred to the GPU along with the round constants (*RC*) and rotation constants (*R*). The kernel is then launched with 25 threads arranged in a 5x5 grid to match the lane configuration of the state. Each thread is associated with a lane in the state based on its (x, y) coordinate in the 5x5 grid. Next, the round constants are moved to shared memory, with each thread moving a single value. The state array is then declared as a 5x5 subset of 64-bit words in shared memory, and each thread initializes its respective lane to 0.

Once setup is complete, the absorption process begins. This process is serial in nature since each block of the input message must be absorbed and processed by the function *f* in sequential order. However, each block consists of 25 64-bit words which can be read in parallel. This parallelism is accomplished by having the first 25 threads read the corresponding words into their respective lanes.

The function *f* is also serial in nature as it is a series of rounds applying the round function found in (5). However, the bulk of the parallelism found in Keccak is located within the round function. Initially, Keccak was implemented in an almost serial nature. Each thread calculated the result for its respective lane; however, many redundant calculations were used to minimize inter-thread communication. This technique was able to place all state variables into registers utilizing no shared memory. Ultimately, the time

saved by not taking advantage of shared memory was outweighed by the time used to perform redundant calculations. Consequently, the state was placed in shared memory and most redundant calculations were removed. θ, ρ and π, and χ are each still calculated in parallel. To achieve this parallelism, each thread calculates the output of the steps θ, ρ and π, and χ for its respective lane and places the result in shared memory. The weakness of this technique is its necessity for synchronization after every step of f. The shared memory technique proved to be approximately 3x faster.

Once absorption is completed for every block of input data, squeezing begins. Since the output for each version of Keccak is less than its respective bitrate, squeezing simply requires writing the first o bits of the output of the absorption phase to produce the final digest. This operation is parallelized by having the first $o/64$ threads write their respective lanes to the digest. Finally, the kernel returns, and the CPU transfers the digest from GPU memory to CPU memory.

Once all three algorithms were implemented, they were tested for performance and typical system loading effects. The testing techniques are discussed in the next chapter.

Chapter 7 Test Methodology

The test system utilized consists of a dual core AMD Athlon 5600+ CPU and an NVIDIA GeForce GTX 285 GPU. The GTX 285 contains 240 processing cores and 1GB of memory. Although this particular model is a higher end GPU, it is indicative of future trends in the GPU market and is still representative of CUDA enabled GPUs. Each algorithm is analyzed in terms of throughput, scalability, and resource utilization. Additionally, the effects of varying the CPU and GPU loads are assessed.

7.1. Algorithm GPU Performance

The throughput of each algorithm is measured in two ways — total throughput and GPU throughput. Total throughput is determined by measuring the total time required to encrypt or hash a dataset, including the time required to transfer data between the CPU and the GPU. GPU throughput is determined by measuring only the encryption or hashing time of the GPU. In order to obtain a value truly representative of the design, each measurement is taken one thousand times and the average throughput is calculated. These measurements are taken on datasets of various sizes ranging from 1KB to 32MB to determine the scalability of the algorithms. The total GPU time is currently more important as it reflects the total time required to go from input data to final output. In the future, this relationship will shift once GPUs and CPUs are combined and the need to transfer data is no longer required. This thesis focuses on improving GPU time as the transfer time is a secondary characteristic which will not be a factor in future designs.

7.2. Typical System Loading Effects

To determine the effects of offloading encryption and hashing from the CPU to the GPU, an assessment of the CPU and GPU resources is made. Using the total throughput time and the GPU throughput time, the CPU time is calculated. This CPU time is then compared to the CPU times of non-CUDA versions of each encryption and hashing algorithm to determine the effects on the CPU.

The non-CUDA CPU times were obtained from the European Network for Excellence in Cryptology (ECRYPT) [23]. This is an organization which collects performance data on a number of ciphers and hashing algorithms for a large variety of modern CPUs. The data obtained for this thesis is from an AMD64 processor running at 3000 MHz. This particular processor was chosen due to its high performance. Although the accuracy of these results is widely accepted, there are large variances in performance in some cases, which are suspect.

To measure the effects of non-encryption related CPU and GPU loads, CPU and GPU loads are simulated using custom applications. The CPU load application is a simple infinite loop with varying sleep times to achieve the desired 20%, 40%, and 60% loads. The GPU load application is a set of rotating tigers. Varying the number of tigers adjusts the GPU load to the desired 25% (200 tigers), 50% (480 tigers), and 75% (2000 tigers). Figure 13 shows this application alongside the measured GPU load of 50% with 480 tigers.

The loads of these applications were determined using Microsoft Perfmon for the CPU and TechPowerUP GPU-Z version 4.4 [24] for the GPU. These are industry proven tools which provide an estimate of the average load over a given period of time. Both

applications were run in parallel with the test code. They produced files containing CPU loads and GPU loads, respectively. These loads were recorded every second; the average load over the execution period for each test was calculated based on the values found in these files. Throughput measurements were made for loads ranging between 0% and 60% on the CPU and 0% to 75% on the GPU. The throughputs calculated under each load were then compared to the unloaded values.

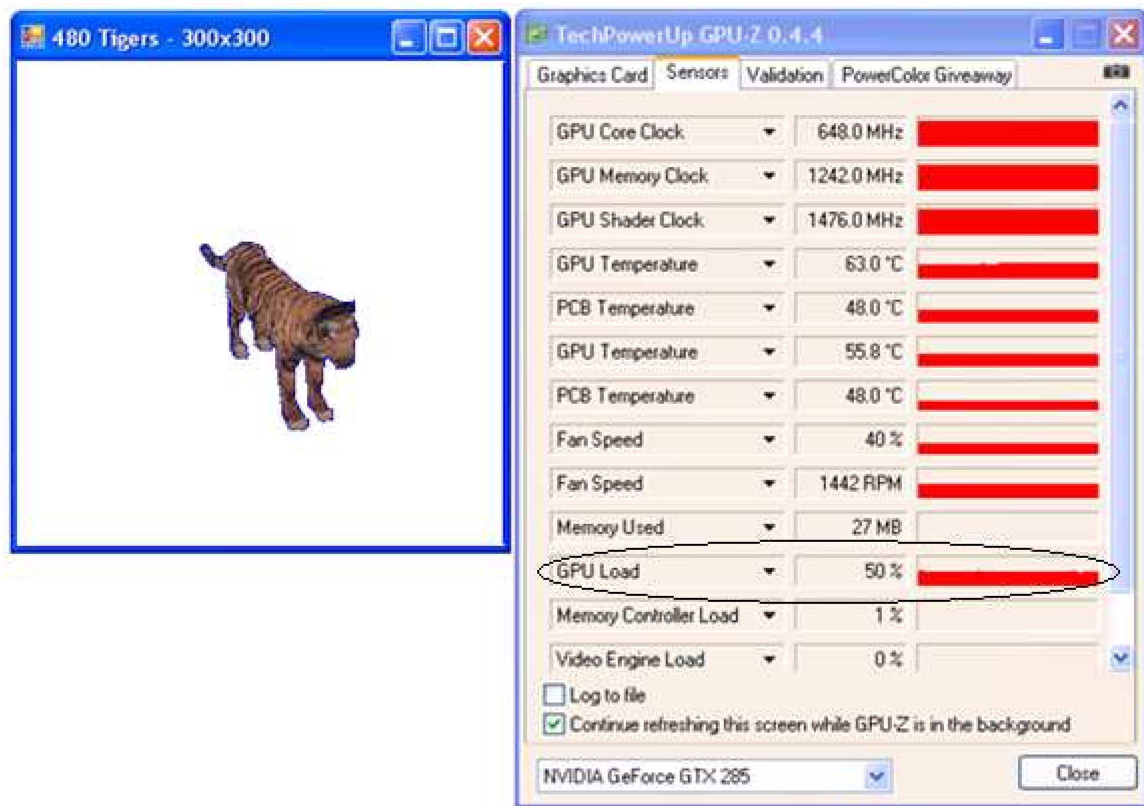

Figure 13: GPU Load Application

Chapter 8 Results and Analysis

Before results could be obtained, the algorithms had to be verified for proper functionality. The documentation available for each algorithm includes a set of standard test vectors as well as their respective results. These values can be found in [12], [25], and [26] for AES, SHA-2, and Keccak, respectively. Each algorithm was used to encrypt or hash the provided test vectors and the resulting output was compared to the provided result. In all cases, these values were identical signifying a properly functioning algorithm. Once the algorithms were verified, they were tested to ascertain their performance.

8.1. Single Kernel GPU Performance

8.1.1 AES

In order to obtain the best performance results, the proper CUDA configuration had to be determined. This determination required performing the AES encryption on numerous data sizes while varying the threads per block: 64, 128, 256, and 512 threads per block were tested. The resulting total throughputs were calculated and graphed as shown in Figure 14, Figure 15, and Figure 16 for AES-128, AES-192, and AES-256, respectively. For all three versions of AES, the four configurations of threads per block performed comparably up to 512 KB. For data sizes greater than 512 KB, the 64 threads per bock configuration outperformed the rest by as much as 11.4%. Hence, the 64 threads per block configuration was selected as the configuration for obtaining GPU AES performance results.

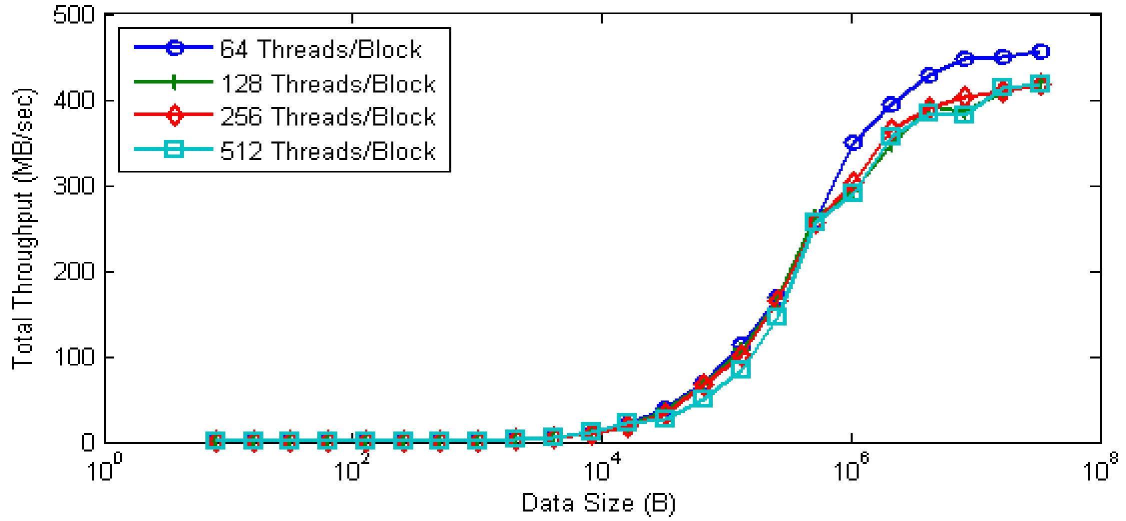

Figure 14: AES-128 Threads/Block Comparison

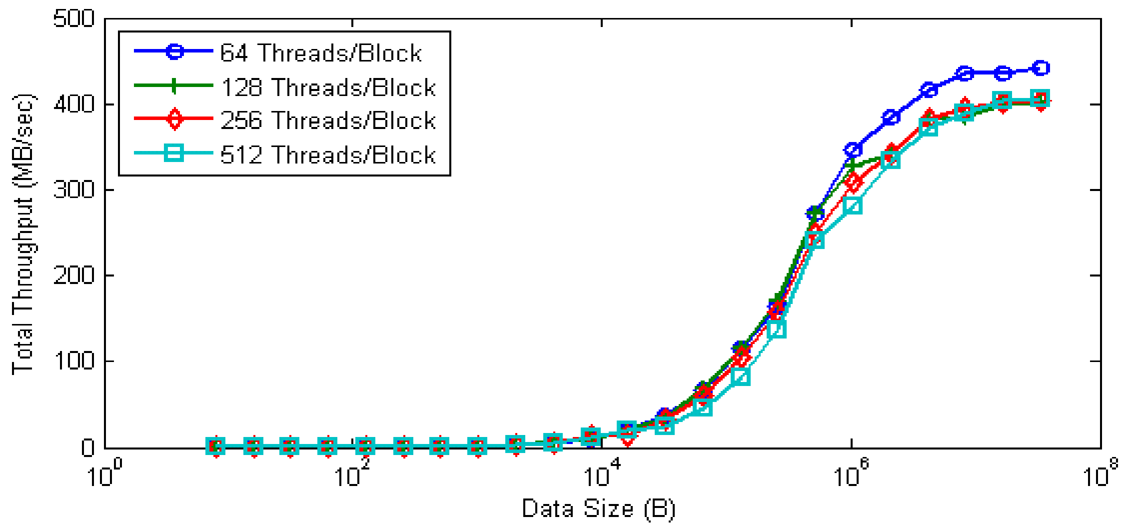

Figure 15: AES-192 Threads/Block Comparison

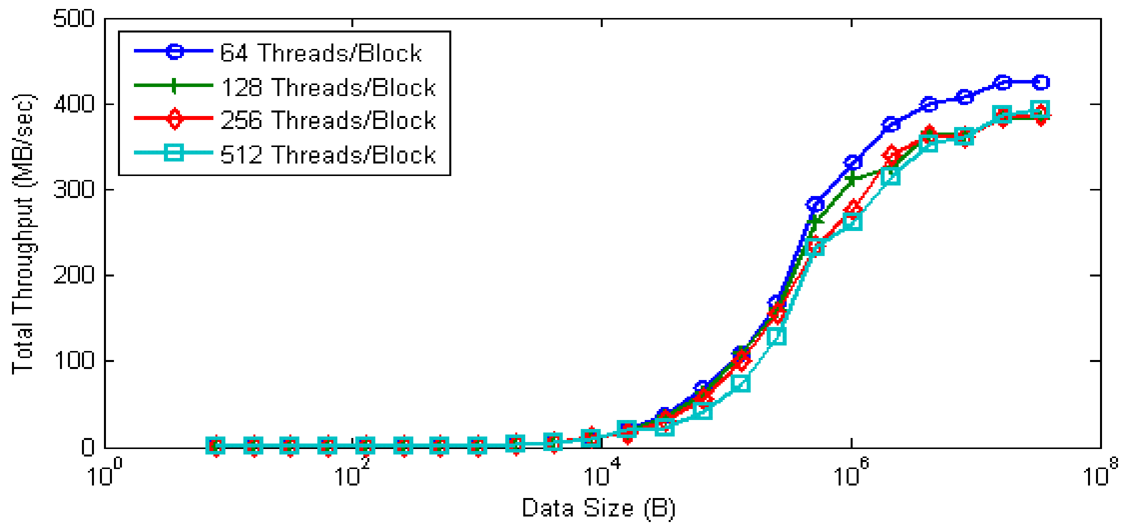

Figure 16: AES-256 Threads/Block Comparison

Using the 64 threads per block configuration, the GPU throughputs and total throughputs were calculated. These values are graphed in Figure 17, Figure 18, and Figure 19 for AES-128, AES-192, and AES-256, respectively. They are graphed alongside CPU throughputs which were obtained from [23]. The total throughput was lower than the CPU throughput for datasets smaller than 512 KB for AES-256 and smaller than 1 MB for AES-128 and AES-192. For larger datasets, the GPU throughput overtakes the CPU throughput approaching approximately 1.9x, 2.3x, and 2.6x faster than the CPU on 32 MB datasets for AES-128, AES-192, and AES-256, respectively. These speedups can be seen in Figure 20. This trend is to be expected of CUDA devices which perform best with large datasets. As discussed earlier, high occupancy is a driving factor of performance in CUDA GPUs. For AES, occupancy increases as dataset size increases with full occupancy being achieved by 512 KB.

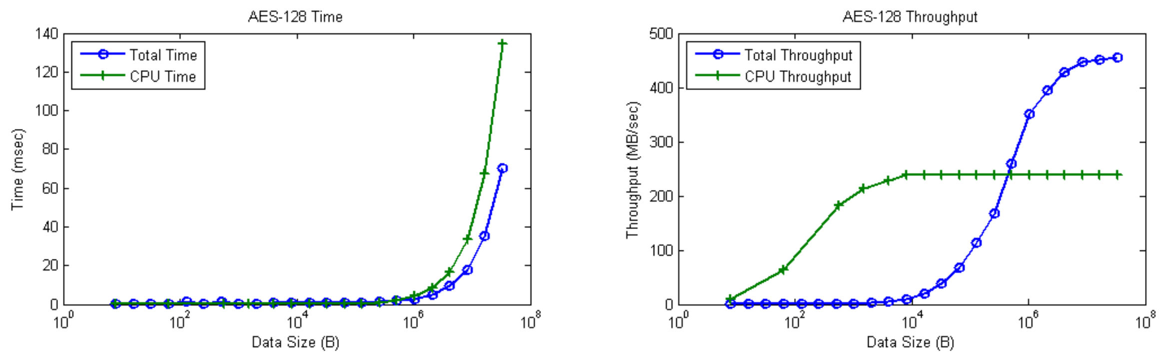

Figure 17: AES-128 Performance

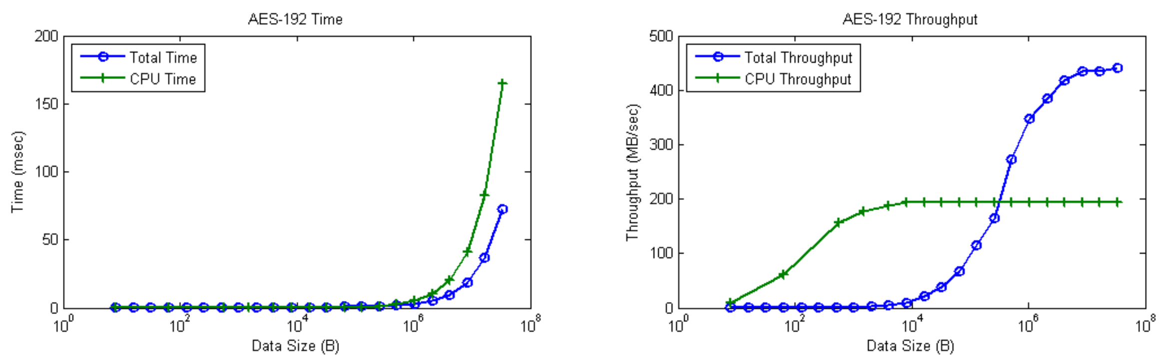

Figure 18: AES-192 Performance

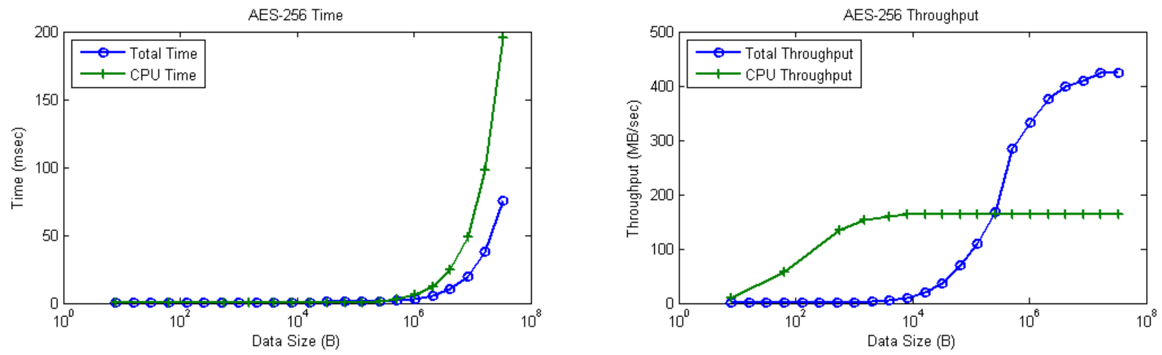

Figure 19: AES-256 Performance

A major bottleneck in the AES performance results is the overhead of data transmission between the CPU and GPU. When only considering the GPU performance of AES encryption, speedups as high as 6x over CPU implementations were seen.

The speedups achieved are somewhat slower than those achieved by previous results. Manavski saw speedups of 5.9x for AES-128 and 5.4x for AES-256 [13]. However, the speedups realized by this thesis are sufficiently comparable for analysis of CPU and GPU load effects.

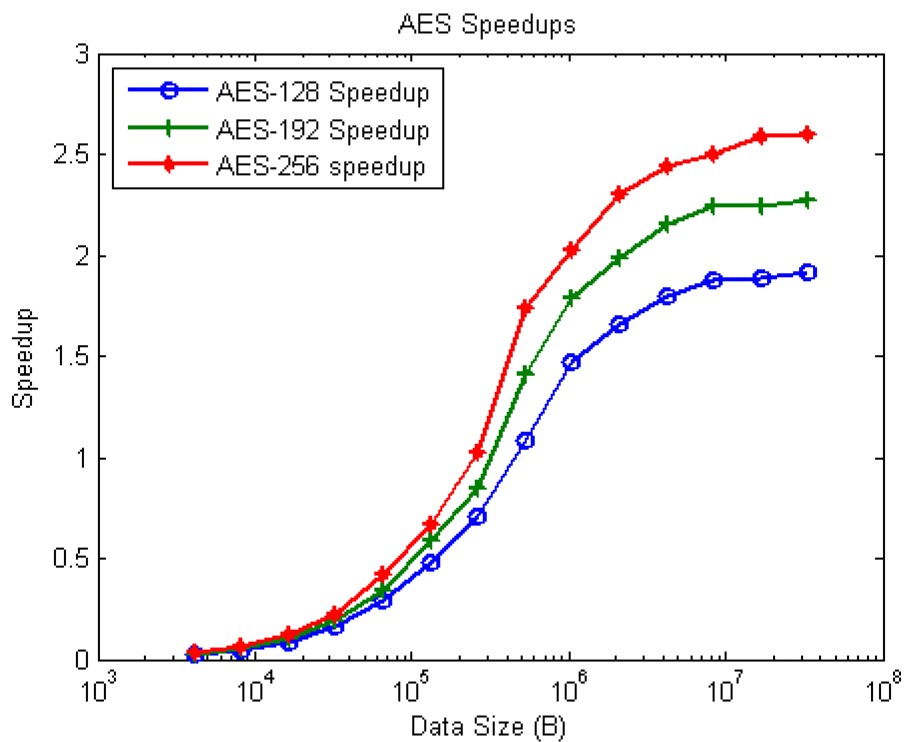

Figure 20: AES Speedups

8.1.2 SHA-2

Unlike AES, SHA-2 was not parallelizable and ran only in a single thread. As a result, GPU computation of SHA-2 was not expected to exhibit the performance increases seen with AES. Figure 21 and Figure 22 illustrate both the GPU performance and total performance versus the CPU results from EBACS for SHA-256, and SHA-512, respectively. The CPU outperforms the GPU by a factor of 400 in the case of SHA-224 and SHA-256, and a factor of 510 for SHA-384 and SHA-512. Maximum performance is achieved at approximately 1KB of data in all four cases. This is the point at which the encryption time is large enough to mask the effects of transferring data between the CPU and GPU.

The large performance degradation can be explained by architectural differences between a CPU and a CUDA enabled GPU. CUDA devices are designed for parallel processing. A single processing core of a CUDA device is vastly inferior when compared head to head with a GPU. The power of CUDA lies in the collective ability of all the cores on the GPU to take advantage of parallelism. Unfortunately, in the case of SHA-2, parallelism is nearly non-existent resulting in this abysmal performance degradation.

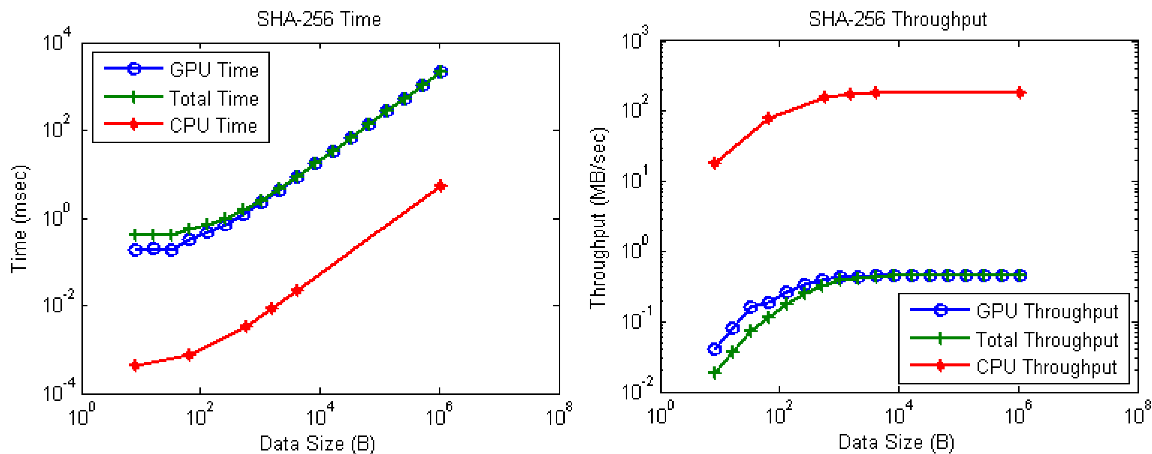

Figure 21: SHA-256 Performance

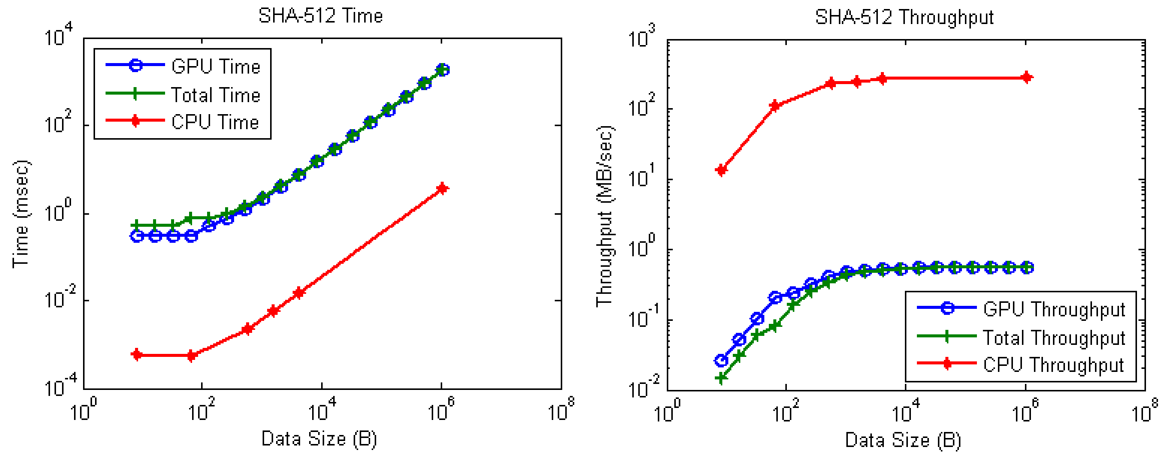

Figure 22: SHA-512 Performance

8.1.3 Keccak

Similar to SHA-2, Keccak does not exhibit large amounts of parallelism. The implementation chosen by this thesis utilized 25 threads, one for each lane of the state. Once again, this lack of parallelism led to degradation in performance when implemented on the GPU. CPU results could be found only for Keccak-256; thus, that is the only comparison available. The CPU version of Keccak-256 outperformed the GPU version by a factor of 100. These results are illustrated in Figure 23 for Keccak-256. GPU implementation of Keccak-224, Keccak-384, and Keccak-512 are expected show similar performance relative to a CPU.

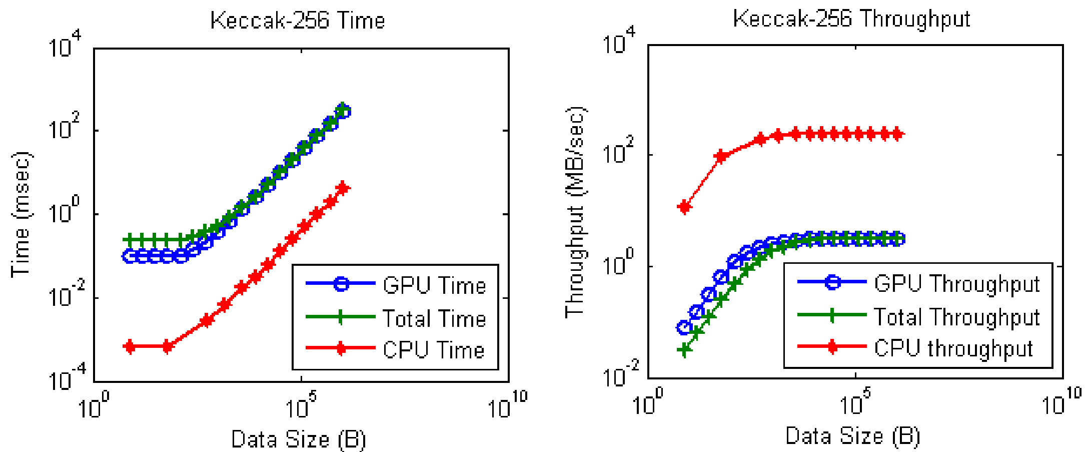

Figure 23: Keccak-256 Performance

8.2. Multi-Kernel GPU Performance

An additional test to analyze the effects of encrypting multiple datasets in parallel on a GPU was performed for AES. The speedups produced are highly dependent on occupancy, or, indirectly, dataset size. For smaller datasets, a single kernel may not be able to produce full occupancy; thus it may benefit from executing multiple kernels. Larger datasets, which do produce full occupancy, will see only small benefits as a result of the pipeling that occurs when executing multiple kernels

The AES encryption was performed 2–4 times in parallel by launching 2–4 kernels simultaneously. The results of these tests can be seen and Figure 24 for AES-256. Simultaneous encryption produced speedups of 1.3x–1.4x with two kernels, 1.55x–1.65x with three kernels, and 1.6x-1.8x with four kernels for datasets smaller than 512 KB. This translates to speedups between 3.6x and 4.7x over CPU implementations of AES. For datasets larger than 512 KB, using multiple kernels had a much smaller speedup of 1x–1.2x. This reduction occurs because of the natural full occupancy at these dataset sizes.

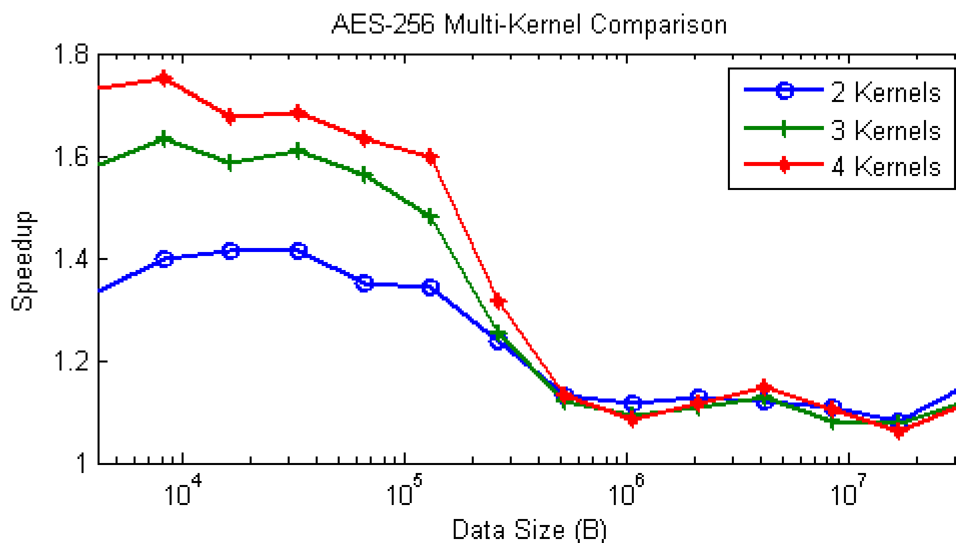

Figure 24: AES-256 Multi-Kernel Comparison

Implementing SHA-2 and Keccak in a multi-kernel way may actually be more beneficial. As neither SHA-2 nor Keccak produce full occupancy, much of the GPU is left unused. With a modified version of the current implementations, 240 simultaneous executions of SHA-2 or 30 of Keccak could be performed simultaneously on a GTX 285 regardless of dataset size.

8.3. Typical System Loading Effects

8.3.1 Offloading Effects

To determine the effects of offloading, the effective CPU time of each GPU implementation had to be calculated. The effective CPU time is the difference between the total time and the GPU time. In cases where GPU time could not be accurately measured, the total time is used as an approximation; this is the case for AES

The effective CPU time was then used to calculate the percentage of CPU time saved by using a GPU implementation. Like previous results, a performance benefit from GPU offloading of cryptographic processing was not seen until large enough datasets were used. For datasets of 256 KB and larger, all versions of the three algorithms saw time savings from GPU offloading. The largest improvement was seen by AES-256, which reduced CPU time by 60%. All implementations saw time savings of at least 20%, and the times savings increased with dataset size up to 50% time saving with a dataset size of 1 MB. These results are illustrated in Figure 25, Figure 26, and Figure 27 for AES, SHA-2, and Keccak, respectively.

These results are highly dependent on both the CUDA implementation and the CPU throughputs compared. In the case of algorithms, such as AES, which exhibit

performance improvements from GPU implementation relative to CPU implementation, the throughput of the CUDA implementation is the primary factor; the percentage of time saved can be calculated directly from the speedup as $1 - \frac{1}{speedup}$. However, for algorithms where a GPU implementation does not reduce execution time, the comparison becomes somewhat more arbitrary. The only conclusion that can be drawn for these cases is that some percentage of CPU time will be saved, although how much cannot be determined for all cases in a systematic way.

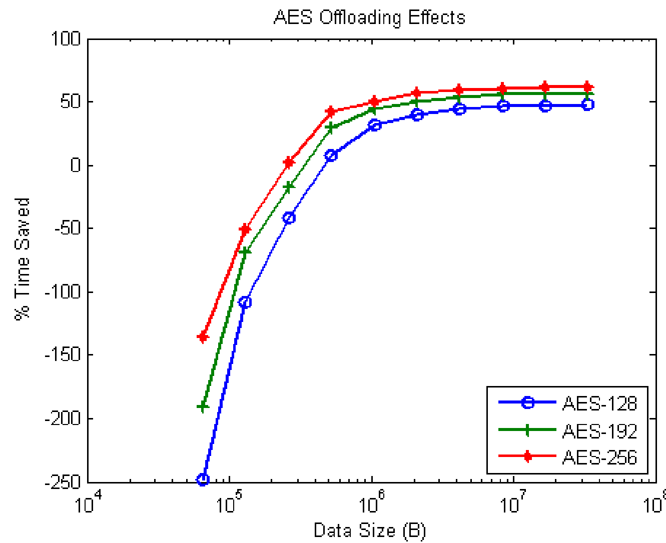

Figure 25: AES Offloading Effects

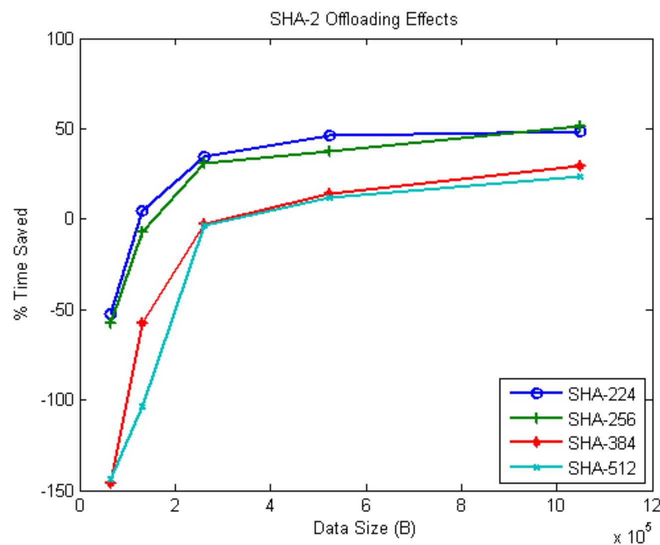

Figure 26: SHA-2 Offloading Effects

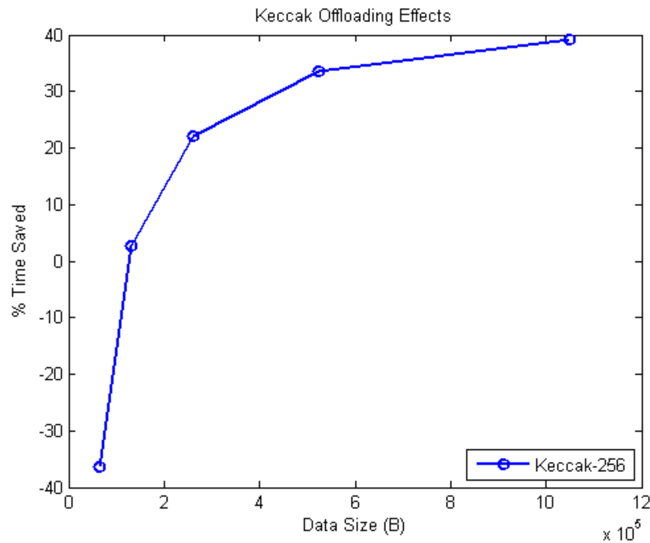

Figure 27: Keccak Offloading Effects

8.3.2 CPU Load Effects

For GPU processing, the effects of additional CPU loads are highly dependent on the total time of the algorithm. The primary use of the CPU is for transferring data and initializing the kernel. Consequently, adding a CPU load has no effect on GPU processing time. However, total time can be adversely affected.

In cases where total time is low, such as AES, a CPU load may have a significant negative effect. The effects are illustrated in Figure 28 for AES-256. With a 20% load, an average degradation of 4% is experienced for datasets larger than 1 MB. Average degradations of 12% and 22% are experienced with 40% and 60% loads, respectively, for datasets larger than 1 MB. The precise effects on datasets smaller than 1 MB are difficult to measure as they are generally encrypted quite fast and are prone to experiencing large percentage increases and decreases with small variations in performance. However, the effects on smaller datasets are generally minimal in terms of time. Similar trends were seen for AES-128 and AES-192 (these results can be seen in Appendix D). The effects of CPU loads are expected to increase as total time of the algorithm decreases.

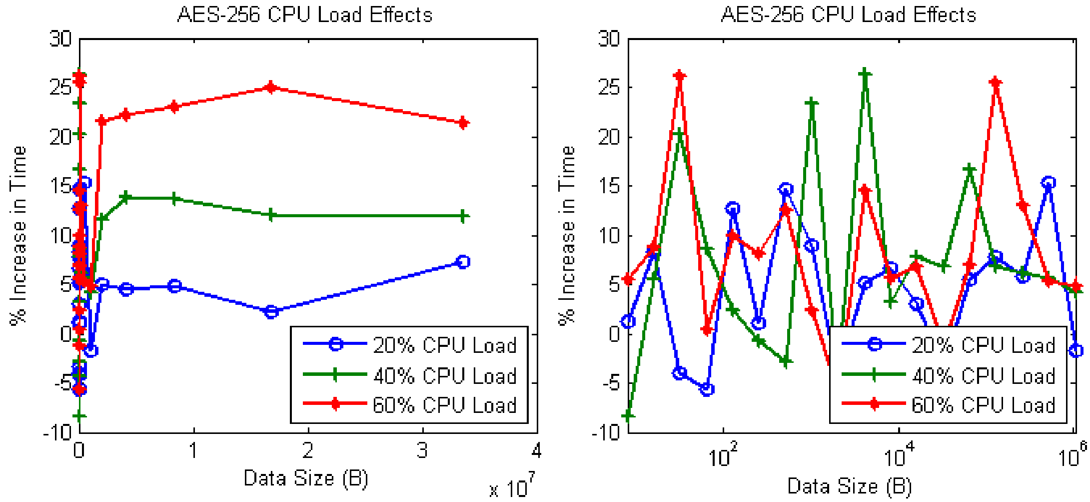

Figure 28: AES-256 CPU Load Effects

In cases, where total time is high, such as for SHA-2 and Keccak, a CPU load has a negligible impact because the majority of the total time is from GPU processing. CPU loading effects are illustrated in Figure 29 and Figure 30 for SHA-512 and Keccak-512, respectively. For all versions of both SHA-2 and Keccak, 20%, 40%, and 60% loads produced no degradation in time greater than 1.5% for files larger than 32 KB. Again, the effects on datasets smaller than 32KB are difficult to measure but are minimal in terms of time. Similar trends were seen for SHA-224, SHA-256, SHA-384, Keccak-224, Keccak-256, and Keccak-384. (These results can be seen in Appendix D).

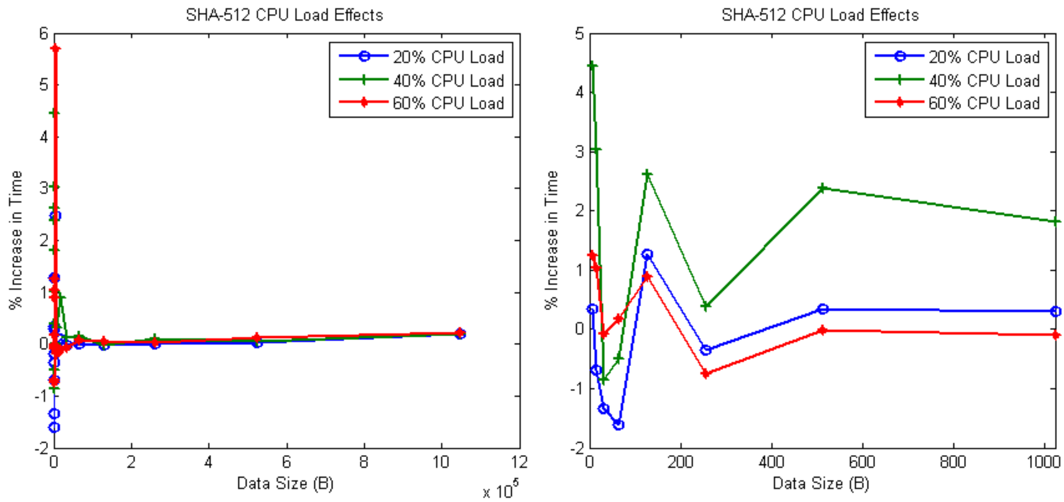

Figure 29: SHA-512 CPU Load Effects

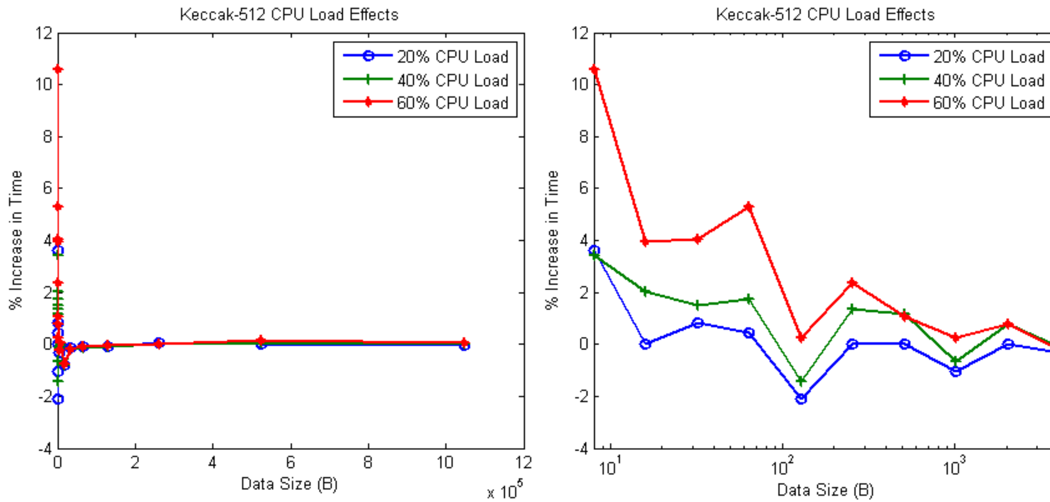

Figure 30: Keccak-512 CPU Load Effects

8.3.3 GPU Load Effects

Like the CPU load, the effects of introducing a GPU load are highly dependent on the time of the algorithm. However, they are quite different in nature. A GPU load consumes some portion of the resources available on the GPU and therefore makes them unavailable for encryption. Additionally, the application producing the load must typically also transfer data between the CPU and GPU. This extra data transfer requirement increases the load on the already slow PCI Express bus.

The primary effect of GPU utilization is on total time. Since GPU utilization affects both the GPU and CPU, it produces a greater increase in total execution time than does purely a CPU load. The overall trend, however, is similar to that of CPU load effects.

As observed with CPU loading, the GPU loading results are highly dependent on total time of the algorithm. With slower algorithms, like SHA-2, as the dataset size increases, the increase in time approaches 0%. For Keccak, which is moderately faster than SHA-2, but slower than AES, the increase in time approaches 2% as dataset size

increases. With faster algorithms like AES, on the other hand, as the dataset size increases, the increase in time does not approach 0%. In the case of AES-256, the increase in time approaches 25%, 27%, and 36% for loads of 25%, 50%, and 75%, respectively. Smaller datasets are not capable of masking the effects of the load well and thus are more affected with greatly increased execution times up to 2000%. These results are illustrated in Figure 31, Figure 32, and Figure 33 for AES-256, SHA-512, and Keccak-512, respectively. Results for other versions of AES, SHA-2 and Keccak can be found in Appendix D.

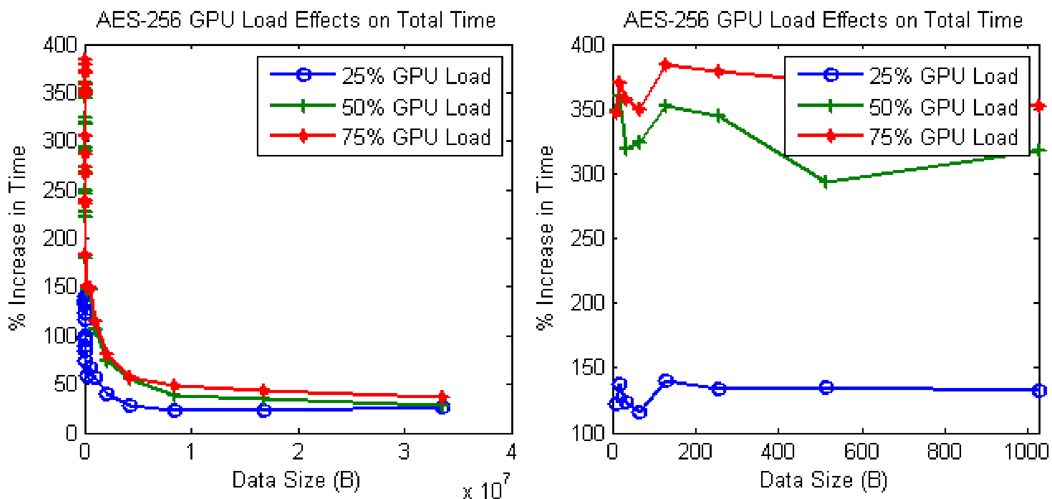

Figure 31: AES-256 GPU Load Effects on Total Time

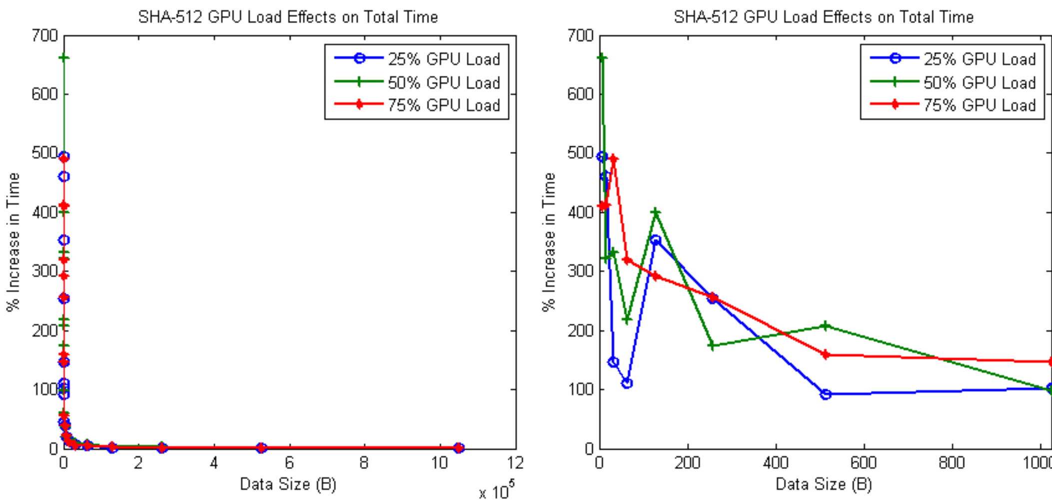

Figure 32: SHA-512 GPU Load Effects on Total Time

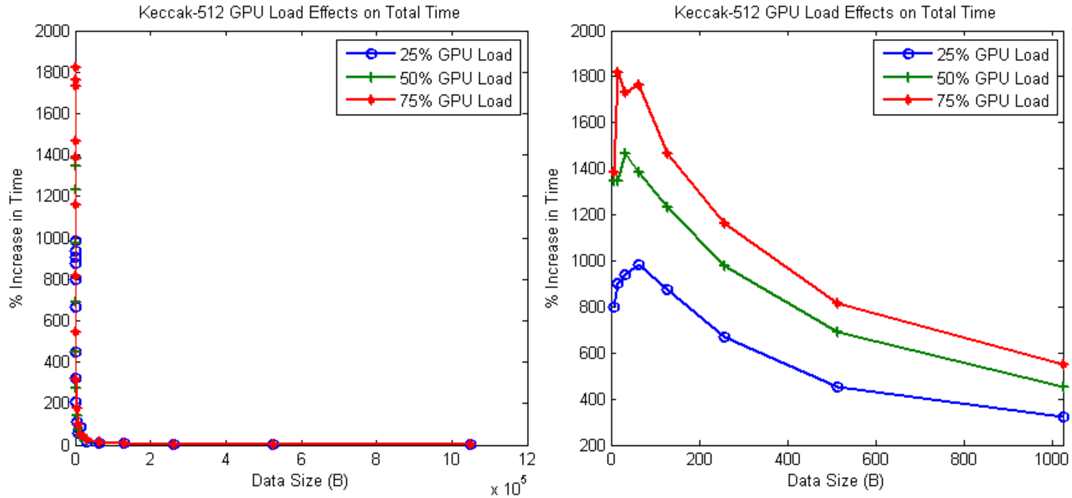

Figure 33: Keccak-512 GPU Load Effects on Total Time

The effects of GPU utilization on GPU time are less noticeable than those on total time. Datasets smaller than 1 MB are more affected since they are encrypted in very short times which cannot mask the overhead as well as larger datasets. They experience an increase in GPU time of 2–4% for SHA-2, and 6–11% for Keccak. As the size of the dataset increases, the overhead is better masked, decreasing the percentage to 0–1% beyond 1 MB for AES and 0–1% beyond 32 KB for SHA-2 and Keccak. These results are illustrated in Figure 34 and Figure 35 for SHA-512 and Keccak-512, respectively.

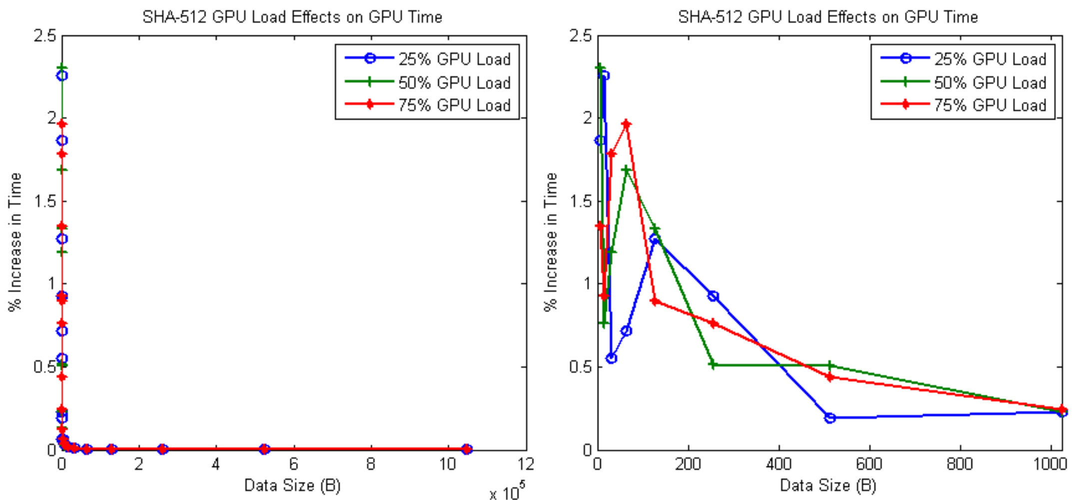

Figure 34: SHA-512 GPU Load Effects on GPU Time

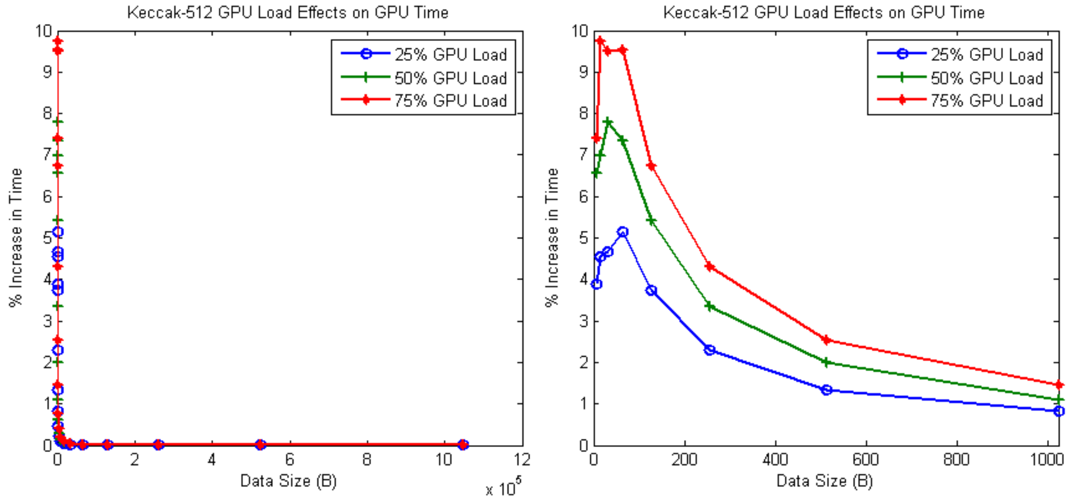

Figure 35: Keccak-512 GPU Load Effects on GPU Time

Chapter 9 Conclusions

CUDA enabled GPU's are very powerful tools, which offer massively parallel processing capabilities. However, only certain highly parallel algorithms can take advantage of their full potential. This thesis investigated three encryption algorithms to understand how they behave on a GPU in typical system environment. With regard to these three cryptographic algorithms, the main contributions of this thesis are investigation and characterization of three distinct aspects of their GPU performance:

- Single-kernel performance of encryption and hashing algorithms on a GPU
- Multi-kernel performance of encryption and hashing algorithms on a GPU
- Effects on GPU encryption performance in typical system configurations

The single-kernel CUDA implementation of AES outperformed a CPU implementation by as much as 2.6x. SHA-2 and Keccak, on the other hand, do not exhibit enough parallelism to take advantage of the multiple GPU cores, and consequently, suffer in performance when implemented on CUDA GPUs. In addition to potentially improving performance, offloading encryption from a CPU to a GPU can free CPU time for other possibly more important tasks. Reduction in CPU time of 40–60%, 22–52%, and approximately 39% were seen for AES, SHA-2, and Keccak respectively.

The multi-kernel CUDA implementations of AES were 1.4x–1.8x faster than the single-kernel implementation, or 3.6x–4.7x faster than CPU implementations. SHA-2 and Keccak were not tested, but are expected to have equal, or most likely, greater improvements from a multi-kernel implementation.

To simulate a typical system environment, various CPU and GPU loads were introduced, and their effects on encryption performance were measured. CPU loads were

found to have no effect on GPU time, but increased the total time of AES by as much as 22%. SHA-2 and Keccak saw minimal increases in time because the load was masked by the total encryption time. GPU loads had a similar effect on total time, although to a greater degree. The total time of AES increased as much as 36%. Again, SHA-2 and Keccak saw minimal increases.

There are several avenues for further research to expand these findings. This thesis showed that offloading can be beneficial but is dependent on the speed of the algorithm being investigated. It would be of value to characterize this behavior. Additionally, this thesis showed that both CPU and GPU loads can have negative effects on CUDA performance. However, only one form of CPU load and one form of GPU load was investigated. It may be of value to apply this same process to various other types of CPU and GPU loads (e.g., benchmarks, applications producing data for encryption, etc.).

Bibliography

[1] Alan Kaminsky and Stanislaw P. Radziszowski, "A Case for Parallelizable Hash," 2009.

[2] Shanxin Qu, Guochu Shou, Yihong Hu, Zhigang Guo, and Zongjue Qian, "High Throughput, Pipelined Implementation of AES on FPGA," in *International Symposium on Information Engineering and Electronic Commerce*, 2009.

[3] Sumio Morioka and Akashi Satoh, "A 10-Gbps Full-AES Crypto Design With a Twisted," *IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS*, 2004.

[4] NVIDIA Co., "NVIDIA CUDA Best Practices Guide Version 2.3," Santa Clara, California, 2009.

[5] NVIDIA Co., "NVIDIA CUDA Programming Guide Version 2.3.1," Santa Clara, California, 2009.

[6] NVIDIA Co. CUDA Zone. [Online]. http://www.nvidia.com/cuda

[7] AMD. (2010, August) ATI Stream Technology. [Online]. http://www.amd.com/US/PRODUCTS/TECHNOLOGIES/STREAM-TECHNOLOGY/Pages/stream-technology.aspx

[8] NVIDIA Co., "NVIDIA CUDA Reference Manual Version 2.3," Santa Clara, California, July 2009.

[9] AMD. (2010, July) The AMD Fusion Family of APUs. [Online]. http://sites.amd.com/us/fusion/APU/Pages/fusion.aspx

[10] National Institute of Standards and Technology. (2010, July) Computer Security Resource Center. [Online]. http://csrc.nist.gov/

[11] National Institute of Standards and Technology, "Advanced Encryption Standard (FIPS-197)," 2001.

[12] Morris Dworkin, "Recommendation for Block Cipher Modes of Operation," National Institute of Standards and Technology, 2001.

[13] Svetlin A Manavski, "CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography," in *IEEE International Conference on Signal Processing and Communications (ICSPC 2007)*, Dubai, 2007.

[14] Andrea Di Biagio, Alessandro Barenghi, Giovanni Agosta, and Gerardo Pelosi, "Design of a Parallel AES for Graphics Hardware Using the CUDA Framework," 2009.

[15] Joseph Bonneau and Ilya Mironov, "Cache-Collision Timing Attacks Against AES,".

[16] Satoh Akashi, Morioka Sumio, Takano Kohji, and Munetoh Seiji, "A Compact Rijndael Hardware Architecture with S-Box Optimization,".

[17] National Institute of Standards and Technology. (2010, July) NIST.gov - Computer Security Division. [Online]. http://csrc.nist.gov/groups/ST/hash/sha-3/

[18] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche, "Keccak Sponge Function Family Main Document," 2009.

[19] Joel Lathrop, "Cube Attacks on Cryptographic Hash Functions," RIT, 2009.

[20] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. (2010, July) The Keccak Sponge Function Family. [Online]. http://keccak.noekeon.org/

[21] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. (2010, February) Cryptographic Sponges. [Online]. http://sponge.noekeon.org/

[22] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche, "Keccak Specifications," 2009.

[23] (2010, July) eBACS: ECRYPT Benchmarking of Cryptographic Systems. [Online]. http://bench.cr.yp.to/

[24] TechPowerUp. (2010, July) GPU-Z Video Card GPU Information Utility. [Online]. http://www.techpowerup.com/gpuz/

[25] National Institute of Standards and Technology, "Secure Hash Standard (FIPS 180-2)," 2002.

[26] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. (2010, July) The Keccak Sponge Function Family. [Online]. http://keccak.noekeon.org/files.html

[27] National Institute of Standards and Technology, "Secure Hash Standard (FIPS 180-3)," 2008.

[28] Robert P McEvoy, Francis M Crowe, Colin C Murphy, and William P Marnane, "Optimisation of the SHA-2 Family of Hash Functions on FPGAs," in *Proceedings of the 2006 Emerging VLSI Technologies and Architectures*, 2006.

[29] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche, "Sponge Functions," 2007.

Appendix A AES Code

The CUDA C implementation of AES is presented below. The code references standard AES constants which can be found in [11].

```
#define NUM_ROUND_KEYS (Nb*(Nr+1))

/****************************************
 * AES encryption
 ****************************************/
uint8_t __host__ __device__ subByte(uint8_t b, uint8_t *sbox){
	return sbox[b];
}

uchar4 __host__ __device__ xor(uchar4 A, uchar4 B){
	uchar4 C;
	C.x = A.x ^ B.x;
	C.y = A.y ^ B.y;
	C.z = A.z ^ B.z;
	C.w = A.w ^ B.w;
	return C;
}

uchar4 __host__ __device__ xor(uchar4 A, uchar4 B, uint8_t C){
	uchar4 D;
	D.x = A.x ^ B.x ^ C;
	D.y = A.y ^ B.y;
	D.z = A.z ^ B.z;
	D.w = A.w ^ B.w;
	return D;
}

uchar4 __device__ xor(uchar4 A, uchar4 B, uchar4 C){
	uchar4 D;
	D.x = A.x ^ B.x ^ C.x;
	D.y = A.y ^ B.y ^ C.y;
	D.z = A.z ^ B.z ^ C.z;
	D.w = A.w ^ B.w ^ C.w;
	return D;
}

uchar4 __host__ __device__ subWord(uchar4 word, uint8_t *sbox){
	word.x = subByte(word.x, sbox);
	word.y = subByte(word.y, sbox);
	word.z = subByte(word.z, sbox);
	word.w = subByte(word.w, sbox);
	return word;
}

uchar4 __host__ __device__ rotWord(uchar4 word){
	uchar1 temp;
	temp.x = word.x;
	word.x = word.y;
```

```
        word.y = word.z;
        word.z = word.w;
        word.w = temp.x;
        return word;
}

void __device__ addRoundKey(uchar4 *state, const uchar4 *roundKey,
                              int round_num){
        state[0] = xor(state[0], roundKey[Nb*round_num+0]);
        state[1] = xor(state[1], roundKey[Nb*round_num+1]);
        state[2] = xor(state[2], roundKey[Nb*round_num+2]);
        state[3] = xor(state[3], roundKey[Nb*round_num+3]);
}

void __device__ subBytes(uchar4 *state, uint8_t *sbox){
        state[0] = subWord(state[0], sbox);
        state[1] = subWord(state[1], sbox);
        state[2] = subWord(state[2], sbox);
        state[3] = subWord(state[3], sbox);
}

void __device__ shiftRows(uchar4 *state){
        uchar4 temp;

         // Second row
        temp.x = state[0].y;
        temp.y = state[1].y;
        temp.z = state[2].y;
        temp.w = state[3].y;
        state[0].y = temp.y;
        state[1].y = temp.z;
        state[2].y = temp.w;
        state[3].y = temp.x;

        // Third row
        temp.x = state[0].z;
        temp.y = state[1].z;
        temp.z = state[2].z;
        temp.w = state[3].z;
        state[0].z = temp.z;
        state[1].z = temp.w;
        state[2].z = temp.x;
        state[3].z = temp.y;

        // Fourth row
        temp.x = state[0].w;
        temp.y = state[1].w;
        temp.z = state[2].w;
        temp.w = state[3].w;
        state[0].w = temp.w;
        state[1].w = temp.x;
        state[2].w = temp.y;
        state[3].w = temp.z;
}

#define xtime(x)  ((x<<1) ^ (((x>>7) & 1) * 0x1b))
void __device__ mixColumns(uchar4 *state){
```

```
	uchar4 Tmp, Tm;

	Tmp.x = Tmp.y = Tmp.z = Tmp.w
	      = state[0].x ^ state[0].y ^ state[0].z ^ state[0].w;
	Tm.x = state[0].y;      Tm.y = state[0].z;
	Tm.z = state[0].w;      Tm.w = state[0].x;
	Tm = xor(state[0], Tm);
	Tm.x = xtime(Tm.x);     Tm.y = xtime(Tm.y);
	Tm.z = xtime(Tm.z);     Tm.w = xtime(Tm.w);
	state[0] = xor(state[0], Tm, Tmp);

	Tmp.x = Tmp.y = Tmp.z = Tmp.w
	      = state[1].x ^ state[1].y ^ state[1].z ^ state[1].w;
	Tm.x = state[1].y;      Tm.y = state[1].z;
	Tm.z = state[1].w;      Tm.w = state[1].x;
	Tm = xor(state[1], Tm);
	Tm.x = xtime(Tm.x);     Tm.y = xtime(Tm.y);
	Tm.z = xtime(Tm.z);     Tm.w = xtime(Tm.w);
	state[1] = xor(state[1], Tm, Tmp);

	Tmp.x = Tmp.y = Tmp.z = Tmp.w
	      = state[2].x ^ state[2].y ^ state[2].z ^ state[2].w;
	Tm.x = state[2].y;      Tm.y = state[2].z;
	Tm.z = state[2].w;      Tm.w = state[2].x;
	Tm = xor(state[2], Tm);
	Tm.x = xtime(Tm.x);     Tm.y = xtime(Tm.y);
	Tm.z = xtime(Tm.z);     Tm.w = xtime(Tm.w);
	state[2] = xor(state[2], Tm, Tmp);

	Tmp.x = Tmp.y = Tmp.z = Tmp.w
	      = state[3].x ^ state[3].y ^ state[3].z ^ state[3].w;
	Tm.x = state[3].y;      Tm.y = state[3].z;
	Tm.z = state[3].w;      Tm.w = state[3].x;
	Tm = xor(state[3], Tm);
	Tm.x = xtime(Tm.x);     Tm.y = xtime(Tm.y);
	Tm.z = xtime(Tm.z);     Tm.w = xtime(Tm.w);
	state[3] = xor(state[3], Tm, Tmp);
}

void __global__ encrypt(const uchar4 *roundKeys_in, uint8_t *data,
uint32_t aes_blocks, uint8_t *sbox_in){
	uint32_t index = blockIdx.x*blockDim.x + threadIdx.x;

	__shared__ uint8_t sbox[256];
	if(threadIdx.x<256){
		int temp = ceil(256.0/blockDim.x);
		for(int i=0; i<temp; i++){
			int temp2 = threadIdx.x*temp+i;
			sbox[temp2] = sbox_in[temp2];
		}
	}

	__shared__ uchar4 roundKeys[NUM_ROUND_KEYS];
	if(index<NUM_ROUND_KEYS)
		roundKeys[threadIdx.x] = roundKeys_in[threadIdx.x];

	__syncthreads();
```

```
	if(index >= aes_blocks)
		return;

	uchar4 state[Nb];
#ifdef TEST_VECTOR
	uint32_t ctr = 0xfcfdfeff + index;
	state[0].x = 0xf0; state[0].y = 0xf1; state[0].z = 0xf2;
	state[0].w = 0xf3;
	state[1].x = 0xf4; state[1].y = 0xf5; state[1].z = 0xf6;
	state[1].w = 0xf7;
	state[2].x = 0xf8; state[2].y = 0xf9; state[2].z = 0xfa;
	state[2].w = 0xfb;
	state[3].x = (ctr>>24)&0xFF;
	state[3].y = (ctr>>16)&0xFF;
	state[3].z = (ctr>>8)&0xFF;
	state[3].w = ctr&0xFF;
#else
	state[0].x = 0; state[0].y = 0; state[0].z = 0; state[0].w = 0;
	state[1].x = 0; state[1].y = 0; state[1].z = 0; state[1].w = 0;
	state[2].x = 0; state[2].y = 0; state[2].z = 0; state[2].w = 0;
	state[3].x = (index>>24)&0xFF;
	state[3].y = (index>>16)&0xFF;
	state[3].z = (index>>8)&0xFF;
	state[3].w = index&0xFF;
#endif

	addRoundKey(state, roundKeys, 0);

	subBytes(state, sbox);  shiftRows(state); mixColumns(state);
	addRoundKey(state, roundKeys, 1);
	subBytes(state, sbox);  shiftRows(state); mixColumns(state);
	addRoundKey(state, roundKeys, 2);
	subBytes(state, sbox);  shiftRows(state); mixColumns(state);
	addRoundKey(state, roundKeys, 3);
	subBytes(state, sbox);  shiftRows(state); mixColumns(state);
	addRoundKey(state, roundKeys, 4);
	subBytes(state, sbox);  shiftRows(state); mixColumns(state);
	addRoundKey(state, roundKeys, 5);
	subBytes(state, sbox);  shiftRows(state); mixColumns(state);
	addRoundKey(state, roundKeys, 6);
	subBytes(state, sbox);  shiftRows(state); mixColumns(state);
	addRoundKey(state, roundKeys, 7);
	subBytes(state, sbox);  shiftRows(state); mixColumns(state);
	addRoundKey(state, roundKeys, 8);
	subBytes(state, sbox);  shiftRows(state); mixColumns(state);
	addRoundKey(state, roundKeys, 9);
#if defined(AES_192) || defined(AES_256)
	subBytes(state, sbox);  shiftRows(state); mixColumns(state);
	addRoundKey(state, roundKeys, 10);
	subBytes(state, sbox);  shiftRows(state); mixColumns(state);
	addRoundKey(state, roundKeys, 11);
#endif
#if defined(AES_256)
	subBytes(state, sbox);  shiftRows(state); mixColumns(state);
	addRoundKey(state, roundKeys, 12);
```

```
      subBytes(state, sbox);  shiftRows(state); mixColumns(state);
      addRoundKey(state, roundKeys, 13);
#endif

      subBytes(state, sbox);
      shiftRows(state);
      addRoundKey(state, roundKeys, Nr);

      uint32_t pos = (index<<4);
      data[pos] = state[0].x;
      data[pos+1] = state[0].y;
      data[pos+2] = state[0].z;
      data[pos+3] = state[0].w;
      data[pos+4] = state[1].x;
      data[pos+5] = state[1].y;
      data[pos+6] = state[1].z;
      data[pos+7] = state[1].w;
      data[pos+8] = state[2].x;
      data[pos+9] = state[2].y;
      data[pos+10] = state[2].z;
      data[pos+11] = state[2].w;
      data[pos+12] = state[3].x;
      data[pos+13] = state[3].y;
      data[pos+14] = state[3].z;
      data[pos+15] = state[3].w;
}
```

Appendix B SHA-2 Code

The CUDA C implementation of SHA-2 is presented below. The code references standard SHA-2 constants which can be found in [27].

```
#define CH(X,Y,Z)        ((X & Y) ^ (~X & Z))
#define MAJ(X,Y,Z)       ((X & Y) ^ (X & Z) ^ (Y & Z))
#if defined(SHA_224) || defined(SHA_256)
      #define S0(X)      (rot(X,2) ^ rot(X,13) ^ rot(X,22))
      #define S1(X)      (rot(X,6) ^ rot(X,11) ^ rot(X,25))
      #define s0(X)      (rot(X,7) ^ rot(X,18) ^ (X >> 3))
      #define s1(X)      (rot(X,17) ^ rot(X,19) ^ (X >> 10))
#elif defined(SHA_384) || defined(SHA_512)
      #define S0(X)      (rot(X,28) ^ rot(X,34) ^ rot(X,39))
      #define S1(X)      (rot(X,14) ^ rot(X,18) ^ rot(X,41))
      #define s0(X)      (rot(X,1) ^ rot(X,8) ^ (X >> 7))
      #define s1(X)      (rot(X,19) ^ rot(X,61) ^ (X >> 6))
#endif

/****************************************
 * SHA-2 encryption
 ****************************************/
uint32_t __device__ rev(uint32_t w){
      w = (w>>16) | (w<<16);
      w = ((w & 0xff00ff00UL) >> 8) | ((w & 0x00ff00ffUL) << 8);
      return w;
}

uint64_t __device__ rev(uint64_t w){
      w = (w >> 32) | (w << 32); \
      w = ((w & 0xff00ff00ff00ff00ULL) >> 8)
            | ((w & 0x00ff00ff00ff00ffULL) << 8);
      w = ((w & 0xffff0000ffff0000ULL) >> 16)
            | ((w & 0x0000ffff0000ffffULL) << 16);
      return w;
}

uint32_t __device__ rot(uint32_t w, uint8_t num){
      w = w>>num | w<<(32-num);
      return w;
}

uint64_t __device__ rot(uint64_t w, uint8_t num){
      w = w>>num | w<<(64-num);
      return w;
}

void __global__ sha2_hash(uint8_t *data, uint32_t data_size,
                              uint8_t *hash){
      uint32_t num_sha2_blocks = data_size/BLOCK_SIZE;

#if WORD_SIZE==4
      uint32_t w[64];
```

```
	uint32_t a,b,c,d,e,f,g,h,T1,T2;
#else
	uint64_t w[64];
	uint64_t a,b,c,d,e,f,g,h,T1,T2;
#endif;
	for(int r=0; r<num_sha2_blocks; r++){
		for(int i=0; i<16; i++){
#if WORD_SIZE==4
			w[i] = rev(*((uint32_t*)data+r*BLOCK_SIZE/WORD_SIZE+i));
#else
			w[i] = rev(*((uint64_t*)data+r*BLOCK_SIZE/WORD_SIZE+i));
#endif
		}
		for(int i=16; i<Nr; i++){
			w[i] = s1(w[i-2]) + w[i-7] + s0(w[i-15]) + w[i-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];

		for(int i=0; i<Nr; i++){
			T1 = h + S1(e) + CH(e,f,g) + k[i] + w[i];
			T2 = S0(a) + MAJ(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;
	}

	memcpy(hash, H, HASH_SIZE);
}
```

Appendix C Keccak Code

The CUDA C implementation of Keccak is presented below. The code references standard Keccak constants which can be found in [22].

```
#define DATA_BLOCK_SIZE (R/W)
#define BLOCK_SIZE      (B/W)
#define HASH_SIZE       (C/2/8)

/****************************************
 * Keccak encryption
 ****************************************/
uint2 __device__ not(uint2 word){
      uint2 word2;
      word2.x = ~word.x;
      word2.y = ~word.y;
      return word2;
}

uint2 __device__ and(uint2 word1, uint2 word2){
      uint2 word3;
      word3.x = word1.x & word2.x;
      word3.y = word1.y & word2.y;
      return word3;
}

uint2 __device__ xor(uint2 word1, uint2 word2){
      uint2 word3;
      word3.x = word1.x ^ word2.x;
      word3.y = word1.y ^ word2.y;
      return word3;
}

uint2 __device__ rot(uint2 w, uint8_t num){
    uint32_t hi_shift, hi_rotated;
    uint32_t lo_shift, lo_rotated;
    uint8_t n;

    if(num>32)
        n = num - 32;
    else
        n = num;

    hi_shift = w.y << n;
    hi_rotated = w.y >> (32 - n);
    lo_shift = w.x << n;
    lo_rotated = w.x >> (32 - n);

    if(num>32){
        w.x = hi_shift | lo_rotated;
        w.y = lo_shift | hi_rotated;
    } else {
        w.y = hi_shift | lo_rotated;
```

```
        w.x = lo_shift | hi_rotated;
    }

    return w;
}
void __global__ keccak_hash(uint8_t *data, uint32_t data_size,
                        uint8_t *hash, uint2 *RC_in, uint8_t *rotc){
      uint8_t row = threadIdx.x;
      uint8_t col = threadIdx.y;
      uint32_t index = 5*row+col;
      uint8_t t1, t2;

      uint32_t num_keccak_blocks = data_size/(DATA_BLOCK_SIZE*8);

      uint2 temp1, temp2, temp3, temp4, temp5;

      __shared__ uint2 RC[Nr];
      if(index<Nr)
            RC[index] = RC_in[index];

      __shared__ uint2 state[5][5];
      state[row][col].x = 0;
      state[row][col].y = 0;

      __syncthreads();

      //absorbing phase
      for(int i=0; i<num_keccak_blocks; i++){
            if(index < DATA_BLOCK_SIZE)
                  state[row][col] = xor(state[row][col],
                        *((uint2*)data+DATA_BLOCK_SIZE*i+index));
            __syncthreads();

            for(int i=0; i<Nr; i++){
                  //theta
                  t1 = col==0 ? 4 : col-1;
                  t2 = col==4 ? 0 : col+1;
                  temp1 = xor(xor(xor(xor(state[0][t1],state[1][t1]),
                              state[2][t1]),state[3][t1]),state[4][t1]);
                  temp2 = xor(xor(xor(xor(state[0][t2],state[1][t2]),
                              state[2][t2]),state[3][t2]),state[4][t2]);
                  temp3 = xor(temp1, rot(temp2,1));
                  temp4 = xor(state[row][col], temp3);

                  //rho and pi
                  temp4 = rot(temp4, rotc[index]);
                  __syncthreads();
                  t1 = (2*col+3*row)%5;
                  state[t1][row] = temp4;
                  __syncthreads();

                  //chi
                  t1 = col==4 ? 0 : col+1;
                  t2 = col==4 ? 1 : (col==3 ? 0 : col+2);
                  if(index==0)
                        temp5 = xor(xor(state[row][col],
                                    and(not(state[row][t1]),
```

```
                                        state[row][t2])), RC[i]);
            else
                  temp5 = xor(state[row][col],
                              and(not(state[row][t1]),
                                    state[row][t2]));
            __syncthreads();
            state[row][col] = temp5;
            __syncthreads();
      }
   }

   //squeezing phase
   memcpy(hash, (uint8_t*)state, HASH_SIZE);
}
```

Appendix D Graphs of Performance Data

To demonstrate and analyze performance trends, Chapter 8 includes plots of only a representative sample of collected performance data. This appendix plots additional performance data that may be of interest and provides more detailed insight.

Figure 36 and Figure 37 show the performance of SHA-224 and SHA-256, respectively. Measured total performance and GPU performance are graphed alongside CPU results from EBACs.

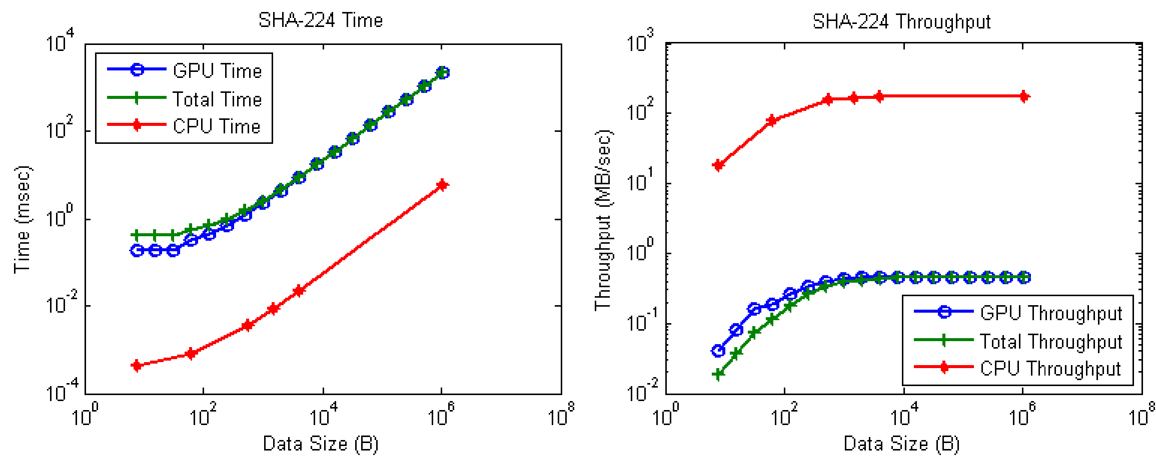

Figure 36: SHA-224 Performance

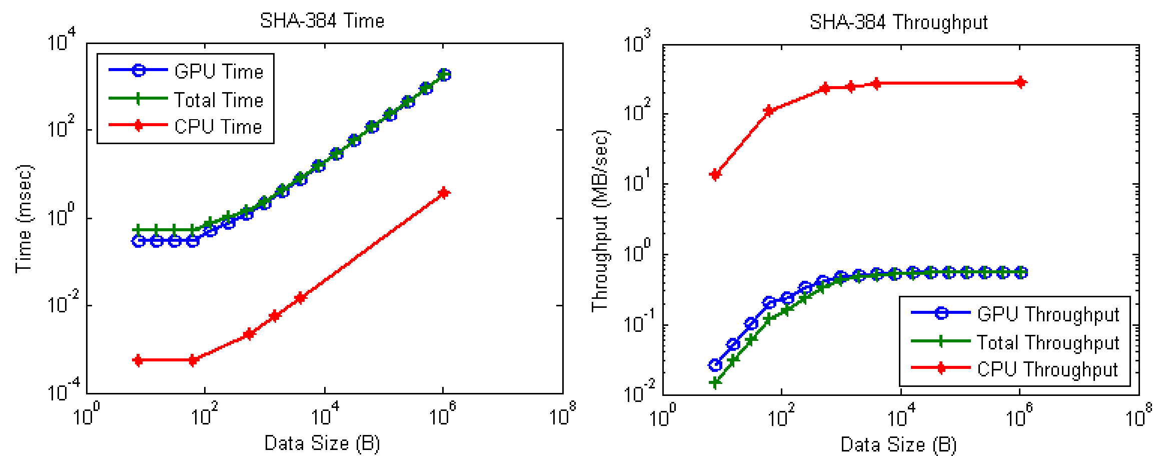

Figure 37: SHA-384 Performance

Figure 38, Figure 39, and Figure 40 show performance results for Keccak-224, Keccak-256, and Keccak-384, respectively. Again, measured total performance and GPU

performance values are graphed. CPU performance values could not be found for these versions of Keccak.

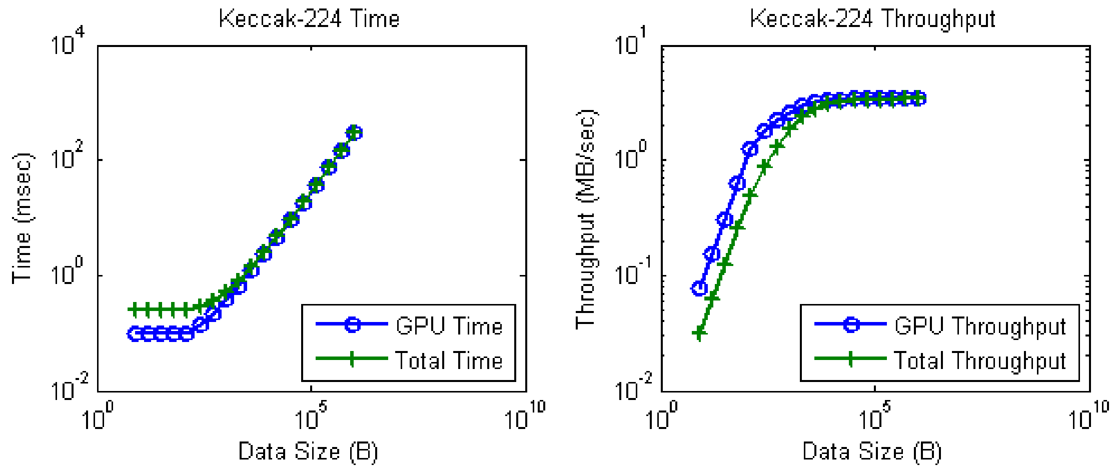

Figure 38: Keccak-224 Performance

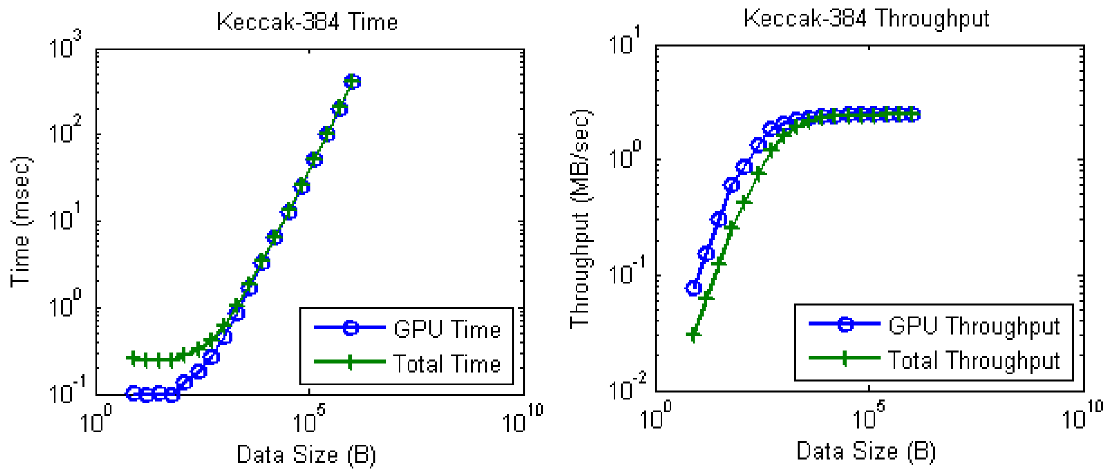

Figure 39: Keccak-384 Performance

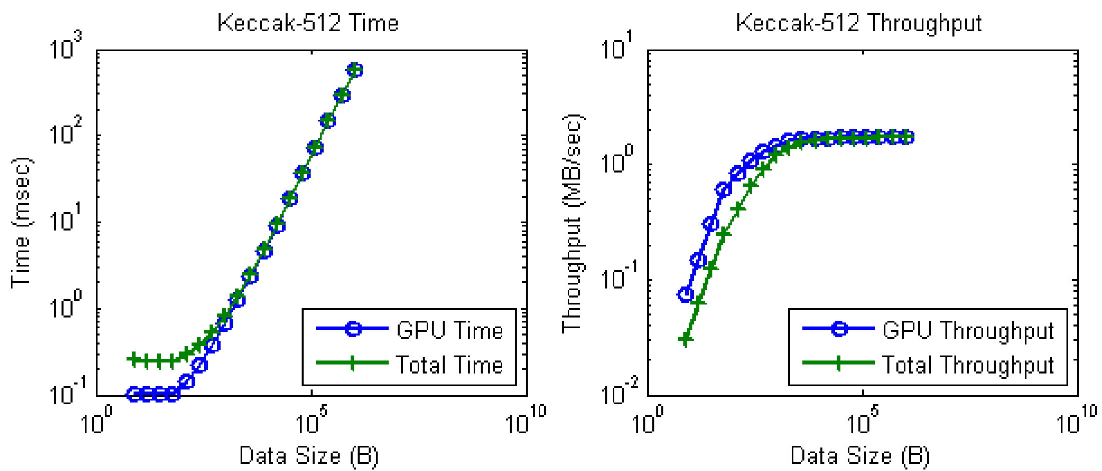

Figure 40: Keccak-512 Performance

Figure 41 and Figure 42 show the speedups produced using a multi-kernel design over a single kernel design for AES-128 and AES-192, respectively.

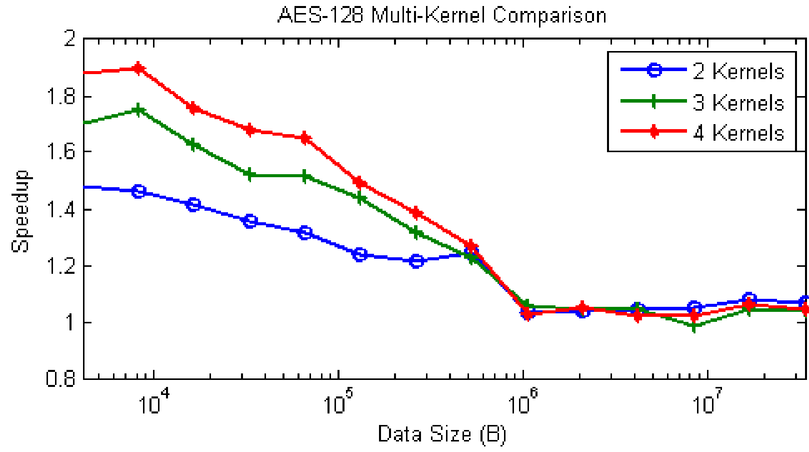

Figure 41: AES-128 Multi-Kernel Comparison

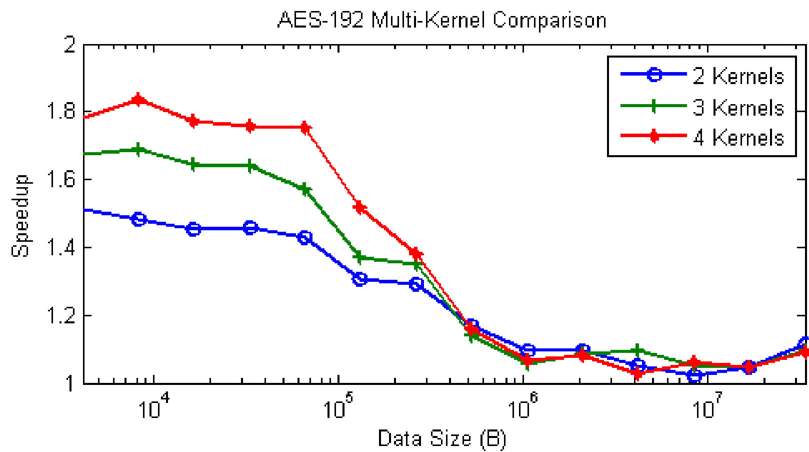

Figure 42: AES-192 Multi-Kernel Comparison

Figure 43, Figure 44, and Figure 45 show CPU load effects for all versions of AES, SHA-2, and Keccak, respectively.

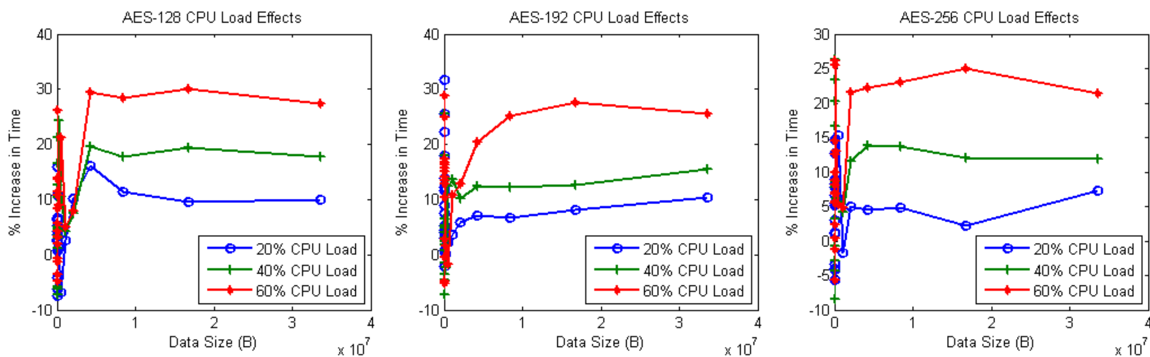

Figure 43: AES CPU Load Effects

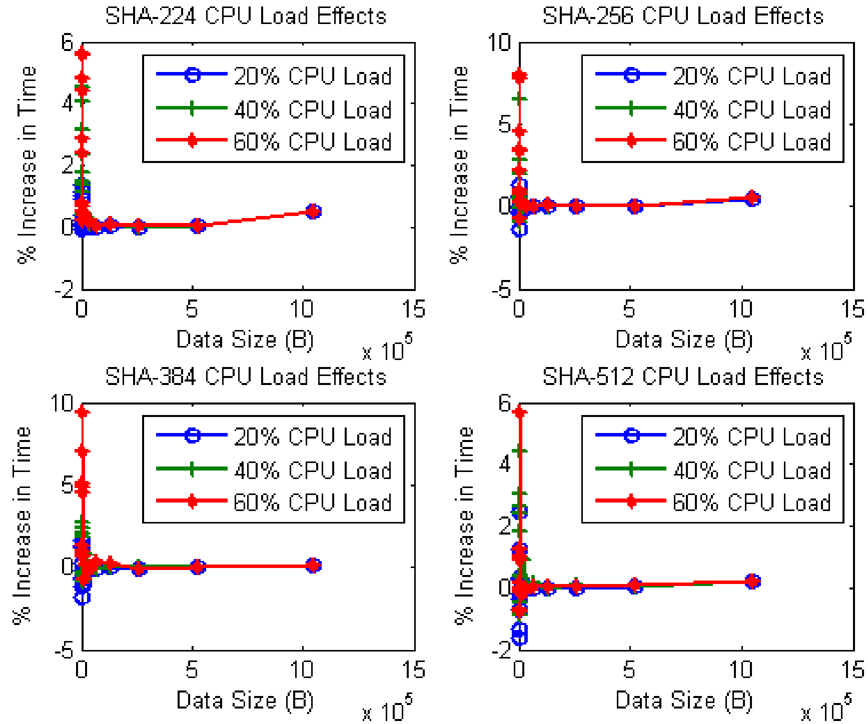

Figure 44: SHA-2 CPU Load Effects

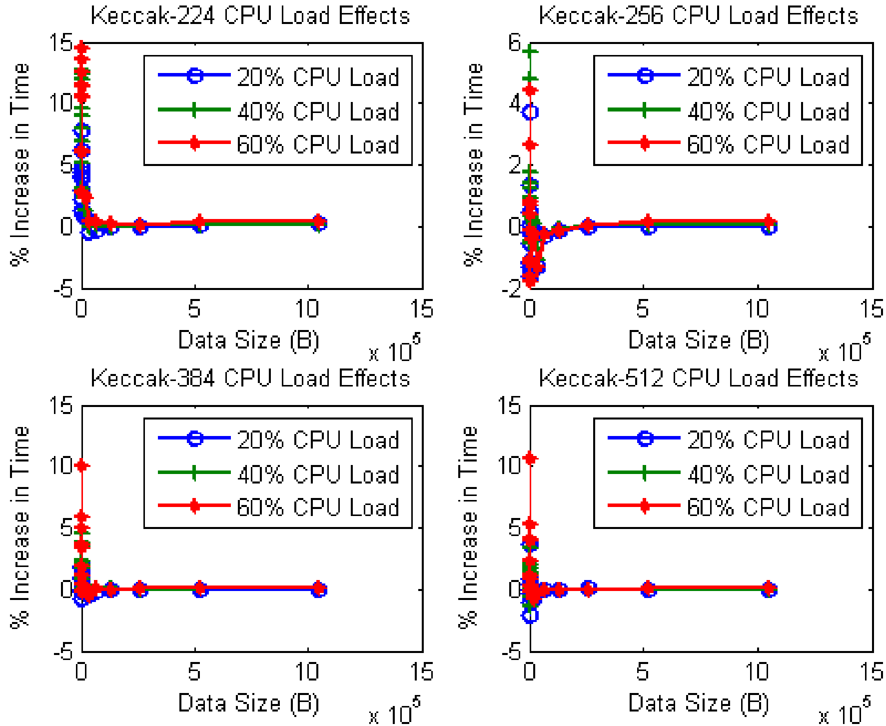

Figure 45: Keccak CPU Load Effects

Figure 46 and Figure 47 show GPU load effects for all versions of SHA-2 and Keccak, respectively.

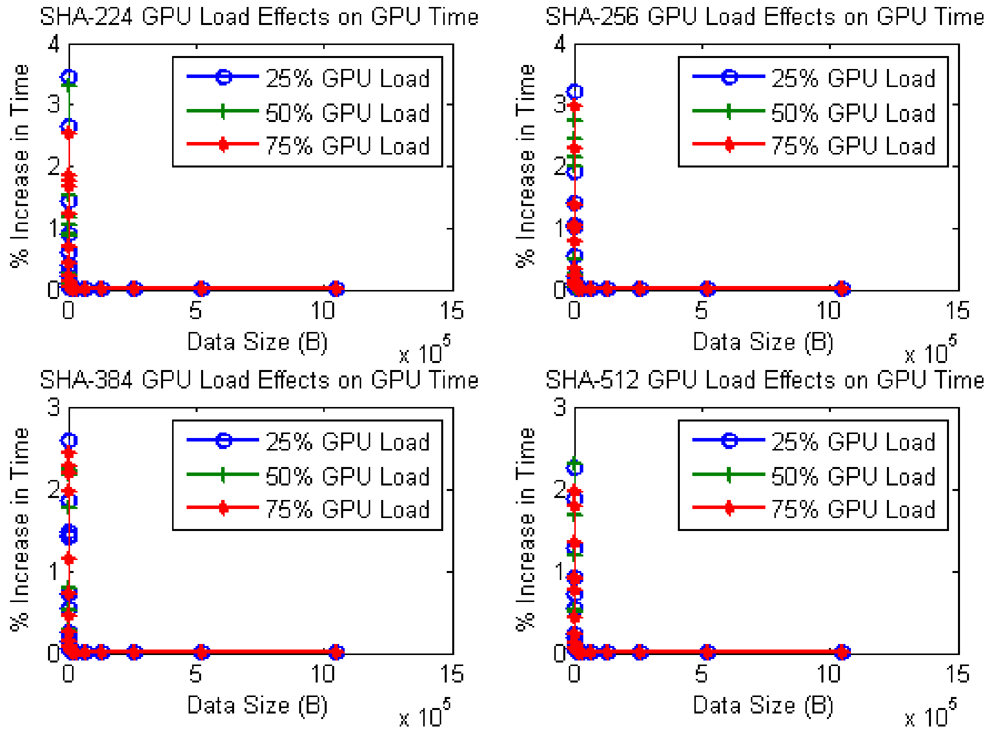

Figure 46: SHA-2 GPU Load Effects on GPU Time

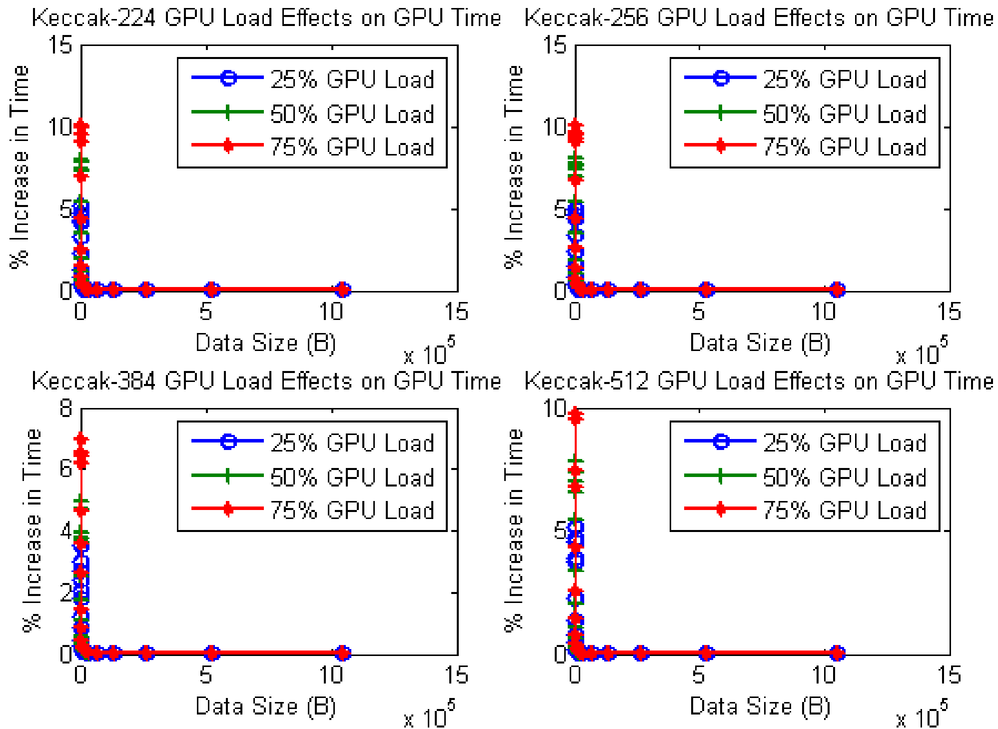

Figure 47: Keccak GPU Load Effects on GPU Time

Figure 48, Figure 49, and Figure 50 show GPU load effects on total time for all versions of AES, SHA-2, and Keccak, respectively.

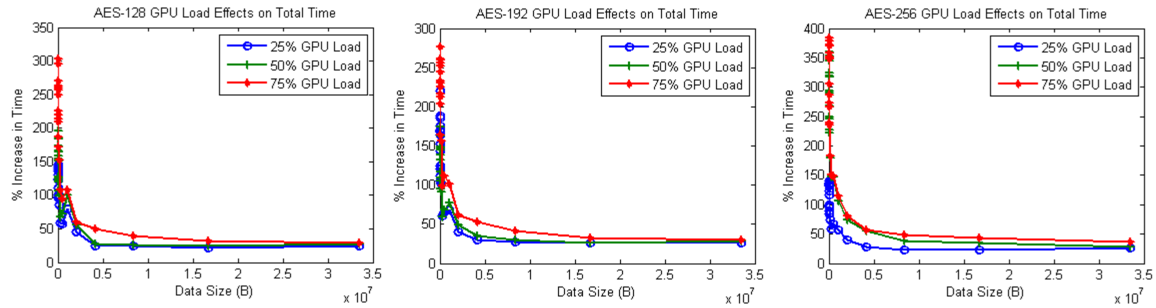

Figure 48: AES GPU Load Effects on Total Time

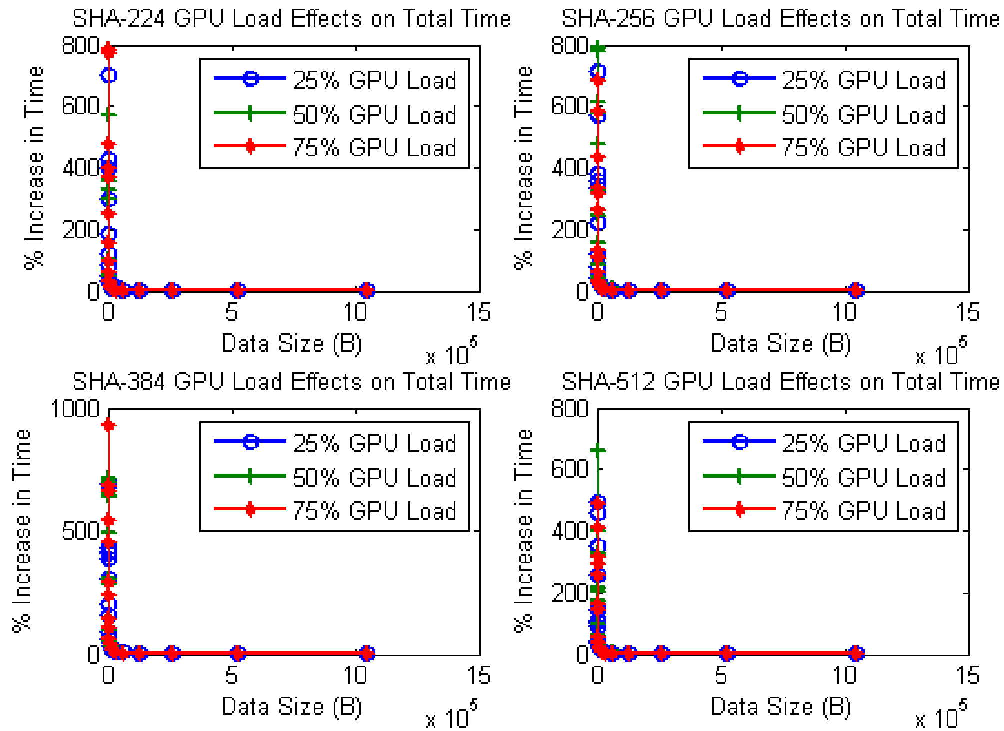

Figure 49: SHA-2 GPU Load Effects on Total Time

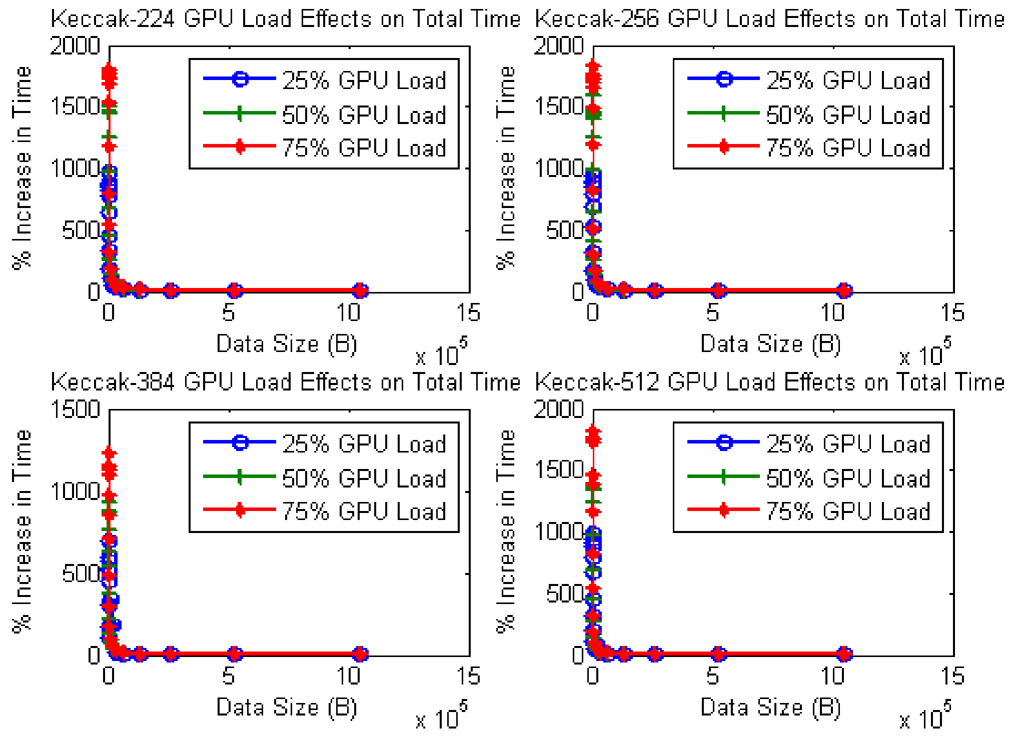

Figure 50: Keccak GPU Load Effects on Total Time