AES Cryptosystem Acceleration Using Graphics Processing Units

by

Ethan Willoner

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

B.SC. COMPUTER SCIENCE HONOURS

in

Irving K. Barber School of Arts and Sciences

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Okanagan)

April 2016

© Ethan Willoner, 2016

Abstract

As the need for processing large amounts of data grows, software developers and researchers work with new ways to accelerate how this data is processed. The Graphics Processing Unit (GPU) was originally designed for the purpose of allowing video games to render graphics more efficiently while freeing resources on the CPU. It was realized that the massively parallel properties of these devices naturally lend themselves to processing large amounts of data when provided with a way to write applications which used GPUs as general compute devices rather than dedicated graphics processors. This thesis sought to continue the work of other researchers in applying these general purpose computing capabilities of GPUs to cryptographic systems, specifically the Advanced Encryption Standard (AES) cryptosystem, which is the current standard. It was hypothesized that by offloading the operations of the cryptosystem to the GPU that large gains in performance to the whole system could be made. Specifically, the goal was to show the benefit of utilizing the GPU for AES operations performed by an open source database such as Postgres. However, experimental results demonstrate that there are only very minor performance gains for GPU-based AES encryption for databases, and the technique is not applicable to many workloads. While previous papers have stated that GPUs are excellent devices for accelerating encryption, these papers have largely ignored how their AES implementations for the GPU compare in various workloads to hardware accelerated alternatives. Modern Intel and AMD CPUs provide developers with the AES-NI hardware accelerated instruction set for performing AES, which is utilized in libraries such as OpenSSL to provide higher performance than an implementation done in software. This paper demonstrates that when an AES implementation utilizes AES-NI, it is far more performant to utilize the CPU than it is to offload the processing to the GPU for typical workloads.

Table of Contents

Abstract . . . . **ii**

Table of Contents . . . . **iii**

List of Tables . . . . **v**

List of Figures . . . . **vi**

Chapter 1: Introduction **1**

Chapter 2: Compute Unified Device Architecture **3**
- 2.1 Overview 3
- 2.2 Threads 4
- 2.3 Shared Memory 8

Chapter 3: Advanced Encryption Standard **10**
- 3.1 Overview 10
- 3.2 Modes of Operation 10
- 3.3 T-Box Optimization 12
- 3.4 AES-NI and OpenSSL 13

Chapter 4: CUDA Implementation of AES and Optimizations **15**
- 4.1 Overview 15
- 4.2 The Rijndael Algorithm 15
- 4.3 Parallelization 18
- 4.4 T-Table Lookups in Shared Memory 21
- 4.5 Asynchronous Memory Copies and Kernel Launches 23

Chapter 5: Benchmarking and Results Analysis **26**
- 5.1 Test Machine and Environment 26
- 5.2 AES CBC Mode of Operation with Paged Data 27

5.3 AES CBC Mode of Operation 33
5.4 AES CTR Mode of Operation 34
5.5 Applications Analysis and Further Optimizations 39

Chapter 6: Conclusion 47
6.1 Future Work 47

Bibliography 50

List of Tables

Table 5.1 System specifications for the test platform. 26
Table 5.2 GPU Implementation with Streams vs OpenSSL performing Encryption. All speeds are in MBps. 30
Table 5.3 GPU Implementation with Streams vs OpenSSL performing Decryption. All speeds are in MBps. 33
Table 5.4 Table of raw processor throughput for the GPU implementation of AES CBC mode with paged data and data copy time excluded. 45

List of Figures

Figure 2.1 The thread layout of a standard CUDA kernel launch [5]. 6
Figure 2.2 A fully coalesced memory access (a) and a sequential but misaligned memory access (b) [4]. 7
Figure 2.3 A strided memory access, where each thread will seek to access every second block in global memory, otherwise known as a stride of two [4]. 7
Figure 2.4 An excerpt of code which shows how the application loads AES round keys into shared memory. 8

Figure 3.1 The standard AES mode of operation (Electronic Code Book mode (ECB)) for encryption (a) and decryption (b) 11
Figure 3.2 A classic demonstration of different modes of operation on the Tux image. This shows how encrypting every block of a data set still leaves patterns exposed. 12
Figure 3.3 The modified algorithm for Cipher Block Chaining (CBC) mode for encryption (a) and decryption (b) . 12

Figure 4.1 Key-Block-Round Combinations [1]. 16
Figure 4.2 The SubBytes, ShiftRows, and MixColumns operations of the AES algorithm. 16
Figure 4.3 The Rijndael Cipher Function written in C pseudocode. 17
Figure 4.4 By treating the 16 byte blocks of data as 32 bit unsigned integers, the application can achieve properly coalesced memory accesses. 19
Figure 4.5 Macro used to change the endianness of data being retrieved from global memory. 20
Figure 4.6 An excerpt of code which shows how the application loads lookup tables into shared memory. 22

Figure 4.7 An example of the frequent usage of t-tables, used to justify storing them in shared memory. 22
Figure 4.8 A demonstration of how to reduce runtime by overlapping copies and kernel executions [6]. 23
Figure 4.9 A visualization of overlapping memory copies and kernel executions, generated with the Nvidia Visual Profiler (NVVP). 24

Figure 5.1 AES CBC Encryption(a) and Decryption(b) with Paged Data 28
Figure 5.2 AES CBC Encryption(a) and Decryption(b) with Paged Data 29
Figure 5.3 The standard deviation of throughput for the AES CBC implementation and OpenSSL with Paged Data 31
Figure 5.4 The standard deviation of throughput for the AES CBC implementation and OpenSSL with Paged Data 34
Figure 5.5 The Counter (CTR) mode for encryption (a) and decryption (b) 35
Figure 5.6 AES CTR Encryption(a) and Decryption(b) 36
Figure 5.7 AES CTR Encryption(a) and Decryption(b) with Paged Data 37
Figure 5.8 The standard deviation of throughput for the AES CTR implementation and OpenSSL with Paged Data 38
Figure 5.9 AES CBC Decryption with Copy Time Excluded . . 41
Figure 5.10 AES CBC Encryption(a) and Decryption(b) with Paged Data and Copy Time Excluded 42
Figure 5.11 AES CTR Encryption(a) and Decryption(b) with Paged Data and Copy Time Excluded 43

Acknowledgements

I would like to thank Dr. Ramon Lawrence for providing a great chance to grow my development skills and improve my university education by allowing me to work on a very interesting project in an area with which I am greatly interested. I would also like to thank Scott Fazackerley and Steven McAvoy for performing the research which would become the inspiration for this work.

Terms and Definitions

AES

Advanced Encryption Standard

AES-NI

AES New Instructions aim to improve the speed of encryption as performed by the CPU by providing hardware accelerated instructions.

CPU

Central Processing Unit

CUDA

Nvidia's Compute Unified Device Architecture. This is a set of libraries and a runtime for massively parallel computations performed on a GPU.

Device

In the context of this report, Device is interchangeable with GPU.

FIPS

Federal Information Processing Standards are standards developed or adopted by the United States federal government for computer systems, government agencies, and contractors.

Galois Field

Otherwise known as a finite field, this is the mathematical basis for the AES cryptosystem.

GPU

Graphics Processing Unit. This term is used loosely to describe a processor and related architecture which operates as a co-processor to the CPU in a massively parallel fashion.

GPGPU

General Purpose GPU

GPU Memory/Device Memory

This refers to the memory physically located on the GPU device, similar to RAM on the host.

Kernel

In this context, refers to the threads, memory, and runtime used by an application which launches and processes data on a GPU.

Modes of Operation

Block cipher algorithms can be modified to provide protections from various different classes of attacks and attempts at profiling the encrypted data. The mode of operation indicates which of these changes are being used with the AES cryptosystem.

Electronic Code Book - ECB

The most basic mode of operation; This mode encrypts each block separately.

Cipher Block Chaining - CBC

This mode of operation utilizes previously encrypted blocks to further obfuscate the encrypted data.

Counter - CTR

This mode of operation utilizes a counter, or a nonce, *XORed* with the plaintext or cipher text to produce output.

NIST

The National Institute of Standards and Technology is a United States government agency tasked with soliciting, verifying, and approving technology, measurement, and standards.

Rijndael

Commonly known as AES, this is the symmetric block cipher algorithm submitted to NIST by Joan Daemen and Vincent Rijmen in consideration for the Advanced Encryption Standard.

Streaming Multiprocessor

This is the physical processor which CUDA schedules threads to run on. GPUs have many of these.

S-box

Otherwise known as a substitution box. This is a lookup table of precomputed values. The values of a plaintext input are substituted with the values found in this lookup table based on a pre-determined algorithm.

Chapter 1

Introduction

Graphics Processing Units are quickly becoming commonplace in numerous areas of computing sciences due to their ability to accelerate processing of data in a highly parallelizable fashion. Both Nvidia and AMD manufacture GPUs specifically for use in high performance workloads, which can include numerous applications, from Computer Aided Design and 3D modeling, to Deep Learning, to Video Processing, to Virtual Desktops, and many others. Both of these manufacturers support the OpenCL computing language; However for the applications of this paper, the system explored shall be Nvidia's Compute Unified Device Architecture parallel computing runtime and programming model. Similar to the PG-Strom extension to the PostgreSQL database [7] which is designed to off-load parallelizable database operations, this research aimed to explore the benefits to utilizing GPUs with databases, for the purposes of accelerating the encryption and decryption of retrieved and stored data, specifically the Advanced Encryption Standard cryptosystem. Additionally, this research seeks to compare the performance benefits of offloading data to the GPU versus utilizing the CPU by comparing benchmarks against the highly performant OpenSSL crypto library, which utilizes the hardware accelerated AES-NI instruction set present on all modern CPUs.

Building off of papers published by previous researchers[8][10][11][12][14][15][16][18], the goal was to implement and benchmark a custom AES implementation which utilized CUDA, and to investigate its benefits when applied to the data formatting and pipeline of a database. Working from the data published in the aforementioned papers, the implementation would utilize optimizations selected to allow the implementation to realize the greatest gains in performance. Additionally, there is a desire to provide an open source library so that others could take advantage of the cryptographic acceleration. Lastly, while an initial goal, the task of integrating this cryptographic library with the Postgres open source database was not realized. The implementation presented, however, should serve as an excellent starting point for others to utilize in this endeavor.

Over the course of this paper, the different components of GPUs and the CUDA runtime will be explored in depth, and what features and optimizations must be utilized to achieve the highest performance possible for the CUDA implementation of AES. Also covered will be an introduction to the AES cryptosystem and its major components, as well as optimizations which can be applied to it, and different modes of operation which aid in serving different purposes and protecting from different kinds of attacks and cryptanalysis. OpenSSL and its use of hardware acceleration through the x86 AES-NI instruction set will be discussed in detail as well. How the implementation can apply AES to databases and other data models will be covered. Lastly, this paper will cover the topic of why previous researchers in the field of GPU accelerated AES have come to incorrect conclusions about its usefulness, and why ignoring AES-NI is an issue for benchmarking different implementations of AES.

Chapter 2

Compute Unified Device Architecture

2.1 Overview

The first GPUs were designed as graphics accelerators, but it took researchers and developers very little time to start leveraging GPUs thanks to their impressive floating point performance, and the term General Purpose GPU (GPGPU) programming became more prevalent. Originally, utilizing GPUs for the purposes of general purpose computing required an intimate knowledge of the hardware as well as the APIs provided to utilize it, such as OpenGL or Direct3D [13]. However in 2006, Nvidia introduced CUDA as a way for developers to be able to leverage GPUs for general purpose uses with a purpose built API. CUDA is a platform, runtime, and programming model designed for the purpose of accelerating the processing of data which can exploit the massive parallelism of GPUs. CUDA allows C, C++, and Fortran code to be executed directly on the GPU, and as such, any language which can utilize bindings into these languages can by extension leverage CUDA as well, allowing for others such as Python, R, Matlab and C# to leverage GPGPU programming.

While it is common for traditional CPUs to have anywhere from a single processing core to thirty-six in the high end server market, GPUs utilize a very different architecture. The Nvidia Tesla M2050, launched in 2010, contains 448 processor cores, and the GTX 780 launched in 2013 with 2304 processor cores. These cores are spread out on the GPU amongst a number of isolated chips, called streaming multiprocessors. These hundreds or thousands of GPU cores are utilized far differently from CPU cores, making up for their significantly lower performance through sheer numbers. As such, utilizing a GPU for computations which are not highly parallel in nature will result in significant performance degradation when compared to performing the same computations on a CPU.

With this increase in available processing pipelines, comes a number of trade-offs however. These GPU processors have a significantly lower core clock speed, and contain limitations on some programming methods such as recursion. Programming on a GPU forces the developer to solve problems in a much more narrowly defined paradigm as compared to CPUs, and as such, it is important to understand the problem that needs to be solved intimately before attempting to leverage CUDA to solve it. Parallelism is not defined by processes or traditional working threads, and this must be taken into account when designing a device kernel. It is also important to understand the limitations of GPUs which are inherent to their design as an external compute device. Not only is there overhead incurred in initializing a GPU Kernel, but the programmer must also account for the time it takes to transfer data between the host memory and GPU memory. CUDA allows the programmer fine tuned control over most of these aspects, but understanding how the underlying runtime works is key to producing the most efficient algorithms.

2.2 Threads

Conventional threads which run on a CPU are typically divergent in behavior, and allow for many different operations to occur simultaneously in each individual thread. These threads which are scheduled by the operating system share the same memory space, but each have their own stack and state. Threads are commonly scheduled by the operating system by utilizing context switching, where each thread gets a certain amount of processing time, and then its state and CPU registers are saved, and the next thread is loaded. As such, a typical thread is free to process what it wants when it is scheduled, and performance should not be affected. Threads often work together and share data by making use of locking mechanisms such as semaphores and mutexes in shared data structures. Rob Pike explains that concurrency is not the same as parallelism [3], and while CPU threads can be parallel, they lack certain properties that make CUDA truly massively parallel. Likewise, CUDA threads can be concurrent, but at the expense of almost any previously held performance benefits, making CPUs far more suited to the task of concurrency.

CUDA threads run on the GPU are by their nature highly parallel, and

as such have a number of differences to CPU threads. Threads are not singularly scheduled, but rather are scheduled in thread groups, called *warps*. Each warp is made up of 32 threads which are executed at the same time on the streaming multiprocessor of the GPU [13]. While a programmer can schedule single threads, it is not advised as without the benefits of parallelism, GPUs are actually quite slow. Each warp is non-divergent, and as such, should some threads be required to branch and some not, each separate divergence shall be executed serially, one after another. In the simple case of the $if-else$ construct, should threads 0-15 branch on if, and threads 16-31 on $else$, the CUDA runtime will run the instructions of 0-15 in parallel, and then reschedule threads 16-31 and run them after the first 16 threads, also in parallel. This serialization of parallelism is expensive, and as such it is wise to design an algorithm so that should one thread branch then they all branch, to avoid these performance penalties.

This method of scheduling warps also allows for the GPU to take full advantage of the caching of instructions in the streaming multiprocessors registers, as well as data caching based on locality. When an access in GPU memory is made, its surrounding elements are cached as well. This way, kernels which access many data elements from common regions of device memory avoid the latency of retrieving them, and instead receive a boost in fetch speed by retrieving them from streaming multiprocessor cache. This ties into *coalesced* memory accesses, where half-warps accessing consecutive data aligned to 128 bytes incur only a single L1 cache transaction to fetch data. If the data is not aligned or consecutive, then this will take multiple access transactions [4].

While warps describe how threads are actually scheduled on the streaming multiprocessor, they are too small of a unit to describe how groups of threads work together as part of a larger computation. This requires the introduction of the concept of a *grid* and a *thread block*. A grid consists of one or more thread blocks, and each thread block contains of one or more threads. All threads within the grid execute the same kernel on the device. The programmer can specify the dimensions of both the grid and the blocks within it. The dimensions dictate how many blocks per grid, and how many threads per block. These dimensions are specified with x, y, z variables, although these values cannot be arbitrary. All CUDA compatible devices have a maximum of 1024 threads per block, so it is up to the programmer to form the dimensions of the thread block to best suite the problem. Grids themselves also have limits to their dimensions as well, although on newer

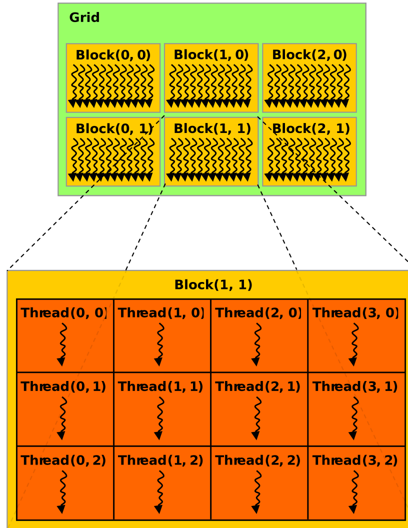


Figure 2.1: The thread layout of a standard CUDA kernel launch [5].

devices this limit is 2^{31}-1 x 65535 thread blocks and therefore can largely be ignored. Thread blocks are what get scheduled to run on the streaming multiprocessor, not individual threads or warps, and the number of thread blocks which can execute on a given multiprocessor is also limited to 16 or 32 blocks on newer architectures, and 8 on older devices. Figure 2.1 demonstrates the layout of a typical device kernel launch, with a number of blocks being launched within a grid, and each block with a number of threads. Notice that the logical layout of the blocks and threads allows for unique indexing of individual threads, which is utilized extensively in any CUDA application to designate chunks of data to threads.

Memory Accesses

When programming with CUDA, the programmer must also consider memory access patterns and their effect on the run time of the CUDA kernel. CUDA performs what is called memory coalescing, which packages as many global memory reads or writes as possible into a single memory transaction. To reap the full benefits of coalesced memory, the application must access contiguous regions of memory from the warp or half warp, aligned to

(a) (b)

Figure 2.2: A fully coalesced memory access (a) and a sequential but misaligned memory access (b) [4].

128 bytes. This can be seen in Figure 2.2(a), where threads from a warp access a single 128 byte section of global memory which occurs in a single memory transaction. Figure 2.2(b) demonstrates why coalesced memory accesses are so important, as it is clearly shown that an unaligned memory access has the unfortunate consequence of producing two memory transactions to access a similar amount of memory. By using a non-optimal memory access pattern, the application has effectively doubled the time required to access memory. While the effects of this are minimal for small warps with limited memory accesses, it is quite clear how failing to access memory in a sequential and byte aligned fashion can quickly amount to large delays in the GPU processing pipeline. It is also a common pattern for a kernel to access multiple sequential regions of memory, or the regions of memory at every n-th position, from a single thread, which is referred to as a strided access pattern, seen in Figure 2.3. This type of access pattern is incredibly expensive as it severely degraded the global memory bandwidth, and as such should be avoided unless absolutely necessary. This type of access pattern is much more suited to shared memory, which is covered next.

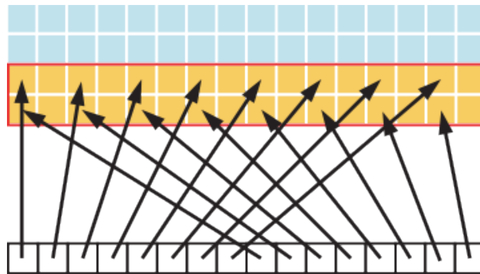


Figure 2.3: A strided memory access, where each thread will seek to access every second block in global memory, otherwise known as a stride of two [4].

2.3 Shared Memory

Shared memory is an important mechanism for achieving the most from a CUDA application by means of allowing data to be exchanged and shared between CUDA threads with much less overhead than global GPU memory. Shared memory is physically located on each streaming multiprocessor which allows for much higher throughput than global memory, with lower latency (Nvidia claims 100x lower), and reduces utilization of global memory bandwidth. These three key benefits help to increase overall performance of CUDA kernels [17], and as such makes it an attractive option to utilize for strided memory access cases, or as a scratch-pad for frequently altered data to prevent global memory accesses. Shared memory is not a solution to all problems though, as it is a limited resource on each streaming multiprocessor, and increased utilization can reduce the performance of other thread blocks running on the multiprocessor, as well as reduce the number of thread blocks being executed at any given time. Shared memory also forces the programmer to understand exactly how they intend to use the memory, and how much, as the shared memory allocation must be specified at kernel launch time. Modern CUDA enabled devices have between 64KB and 112KB of shared memory available on each multiprocessor, with a maximum of 48KB allocated to a single thread block.

```
1  __shared__ uint8_t Rks_s[256];
2
3  if (threadIdx.x <= 256) {
4      uint32_t i = ceil(256.0 / blockDim.x);
5      for (uint32_t j = 0; j < i; j++) {
6          uint32_t idx = threadIdx.x * i + j;
7          Rks_s[idx] = ctx->Rks_d[idx];
8      }
9  }
10 __syncthreads();
```

Figure 2.4: An excerpt of code which shows how the application loads AES round keys into shared memory.

Figure 2.4 contains an example of how a programmer might utilize shared memory in their application, such as an implementation of the AES cryptosystem. *Rks_s* defines a fixed array of shared memory with type uint8_t

(unsigned 8-bit data type) which will hold all of the encryption or decryption keys needed for the kernel to operate on the input data. The kernel utilizes threads running on the multiprocessor to load the data from global memory (Rks_d) into the shared memory buffer (Rks_s). Once this is completed, the application synchronizes all the threads to ensure that all memory loads are completed before the application proceeds.

Figure 2.4 is a common utilization of shared memory. The data stored in Rks_s will be accessed frequently by each thread which is executing in the CUDA kernel, and therefore the global memory bandwidth usage will be decreased, and memory accesses to this data will be significantly faster than if accessed in global memory. Additionally, this shared memory allocation is not so large as to be an inhibiting factor to the performance of other thread blocks which may simultaneously be executing on the streaming multiprocessor.

Chapter 3

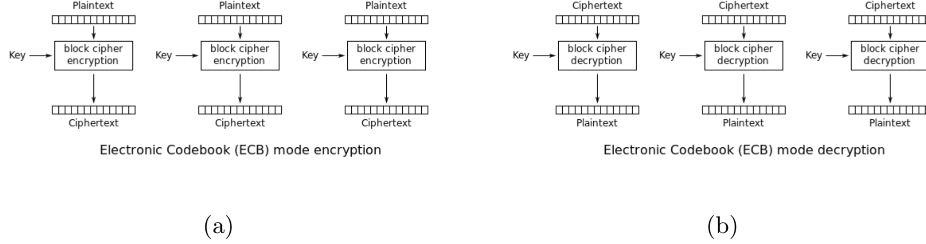
Advanced Encryption Standard

3.1 Overview

AES, or the Advanced Encryption Standard, is a symmetric block cipher chosen in 2000 by NIST and specified in FIPS publication 197 [1] to serve as a standard cryptosystem, with submissions being judged based on security, both current and expected future, ease of implementation, performance and ability to be implemented on low end devices with limited resources, as well as strength against cryptanalysis, and several others. Rijndael is the algorithm submitted by Joan Daemen and Vincent Rijmen which was approved for AES. It supports key sizes of 128, 192, and 256 bits with a block size of 128 bits, and performs 10, 12, or 14 rounds of the cipher depending on key size. While not implemented or standardized, it is possible for Rijndael to be expanded both to larger block and key sizes. As such, the AES cipher encrypts or decrypts 128 bits of data at a time, to produce ciphertext and plaintext respectfully. The 128, 192, or 256 bit key is not used for every round of the block cipher, but instead is expanded into what is called a *key schedule*. The *key expansion* generates a schedule of bytes equal to $Nb(Nr+1)$ where Nb is the size of the key in bytes, and Nr is the number of rounds the algorithm performs. The generation utilizes substitution tables (S-box) and a cyclic byte rotation $XORed$ with the previously generated key to fill the key schedule [1].

3.2 Modes of Operation

Despite the high security nature of AES, it is not immune to all types of attacks, such as the classic attack shown in Figure 4.2 which allows for Tux the penguin to remain visible when rendered as an image even after being encrypted. This is a common weakness in any cryptosystem which

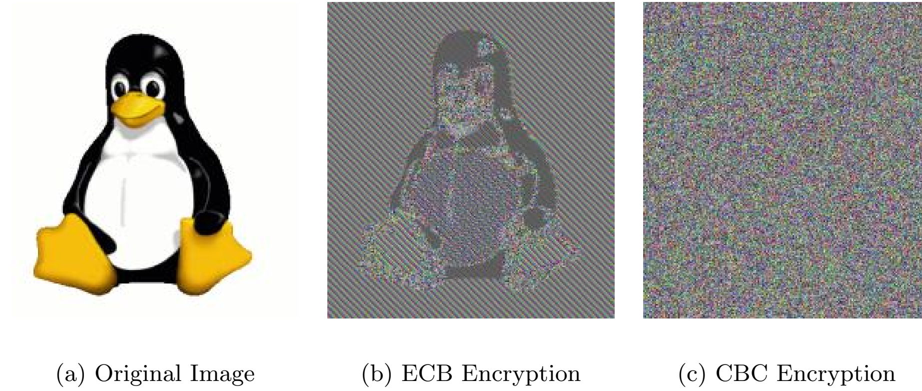


(a) (b)

Figure 3.1: The standard AES mode of operation (Electronic Code Book mode (ECB)) for encryption (a) and decryption (b)

utilizes the same key for any given input. To counteract this, variations to the standard Rijndael algorithm are created which add additional sources of entropy to the encryption process. One such example is Cipher Block Chaining (CBC) mode, which $XORs$ the first block with an Initialization Vector (IV) and subsequent blocks with the previously generated ciphertext block, as shown in Figure 3.2 (c). Additional modes of operation for AES are defined in NIST 800-38A [2], which outlines several others including Cipher Feedback (CFB) mode, Output Feedback (OFB) mode, and Counter (CTR) mode. AES can also be implemented with a Hash-based Message Authentication Code (HMAC) in the Galois Counter Mode (GCM) which provides the encrypted data with integrity in addition to security.

By examining ECB mode in Figure 3.1, it quickly becomes clear how this mode of operation can be parallelized. In both the encryption and decryption diagrams, it is shown that each block is isolated and is not reliant on the surrounding blocks. This means that a GPU implementation can perform encryption or decryption on any block, in any order, on as grand a scale of parallelism as desired. However, in Figure 3.3 it can be seen that parallelism is not quite as simple as previously. For encryption (a), each block requires the ciphertext output of the previous block of data as input. This serializes the algorithm, and removes any capabilities for parallelism. Upon examination of the decryption process though, parallelism is revealed to be possible. The decryption of each block requires the ciphertext of the previous block, which does not need to be generated. Therefore, parallelization of the CBC decryption process can take place, and this paper will see how this is achieved in a later chapter.

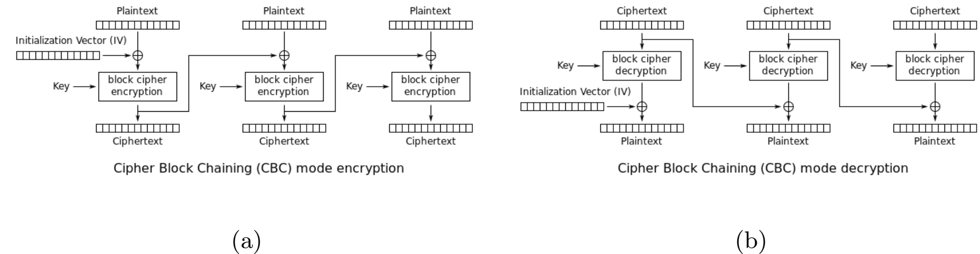


(a) Original Image (b) ECB Encryption (c) CBC Encryption

Figure 3.2: A classic demonstration of different modes of operation on the Tux image. This shows how encrypting every block of a data set still leaves patterns exposed.

3.3 T-Box Optimization

The Rijndael cipher relies on a number of operations in each round to produce its ciphertext, and the same holds true for decrypting the ciphertext back into plaintext. This includes, for each round, performing a byte substitution with the S-Box, a cyclic row shift on a 4x4 matrix of the 16 byte block, and a polynomial transformation in the Galois Field 2^8. At the end of each round, the result of these operations is *XORed* with the round key, and the next round commences. All these operations, while constant in running time, are expensive to compute. Originally proposed in the Rijndael



(a) (b)

Figure 3.3: The modified algorithm for Cipher Block Chaining (CBC) mode for encryption (a) and decryption (b)

submission to NIST is the replacement of a number of these operations with a set of four pre-computed lookup tables for both encryption and decryption, called $T - tables$, or $T - Boxes$ [9], the basis for which can be found in Section 5.2.1 of the Rijndael proposal. These lookup operations replace the computations required for each round of AES, at the expense of needing to load and store an additional 4096 bytes for either encryption or decryption in memory. For modern computing systems, these resource requirements are trivial, and therefore most non-embedded systems can receive significant performance gains by implementing a T-Box solution. Smaller embedded systems with less available resources can also benefit from this concept, by instead storing a single 1024 byte table. This table is a stand-in for the other 3 tables, at the cost of an additional 3 per round of the cipher, for every 4 bytes in the block. This method drastically decreases memory usage, a very finite resource in embedded systems, while still helping to improve the efficiency of Rijndael.

3.4 AES-NI and OpenSSL

In 2008 Intel and AMD proposed an extension to the x86 instruction set architecture, known as the Advanced Encryption Standard New Instructions, or AES-NI. This instruction set aimed to increase the throughput on modern CPUs of the encryption and decryption of data with the Rijndael cipher by providing hardware acceleration for AES and exposing this functionality to the programmer by way of the new assembly instructions. The prevalence of AES utilization in applications for both desktop and server is incredibly high. It is utilized in full disk encryption applications such as FileVault, Bitlocker and TrueCrypt for many users of Mac OSX and Windows operating systems and in web browsers such as Firefox and Chrome for secure internet sessions over SSL and TLS. AES is utilized further in the enterprise and server space by way of encrypted databases, secure Virtual Private Networks, traffic inspection tools, HTTPS enabled web servers and API endpoints, and numerous other applications which provide high volume services. Intel claims that the AES-NI hardware acceleration increases performance of the AES algorithm by 3 to 10x that of a purely software implementation, increasing not only throughput and therefore reducing latency, but also freeing processing cycles for usage by more complex algorithms, more clients, or more featureful applications. Adding to the throughput potential, Intel supports instruction pipelining for the core instructions of the AES algorithm, allowing for parallel execution of block encryption or

decryption when multiple blocks are present.

This paper shall be focusing on the OpenSSL cryptographic library and its usage of AES-NI. OpenSSL is one of, if not the most used cryptographic libraries in the world, and can be considered a de facto industry standard. It is almost universally installed on Mac OSX and Linux systems, and is utilized by countless desktop, server, and embedded systems across all major operating systems. As demonstrated with the recent Heartbleed vulnerability, security flaws in OpenSSL affect huge swathes of internet connected computers due to the number of deployments. OpenSSL utilizes AES-NI for hardware acceleration of the AES cryptosystem on supported platforms, which includes most CPU's from AMD and Intel since 2010, with a few exceptions on lower end offerings. Therefore, to compare the performance of the AES implementation in CUDA with the optimized implementation found in OpenSSL which leverages AES-NI is a reasonable benchmark for the performance benefits of the implementation. Any system which processes large amounts of data is surely utilizing newer CPU architectures to leverage increases in instruction per clock performance, core count and clock speeds, improvements in memory bandwidth, and other technological progressions as well as AES-NI.

Chapter 4

CUDA Implementation of AES and Optimizations

4.1 Overview

At the core of this project is the implementation of Rijndael, the cryptosystem more commonly referred to as AES. This section will take a high level look at how an AES implementation generates the set of keys, known as a key schedule, from the initial 128, 192, or 256 bit key, as well as how it is utilized to encrypt and decrypt data. Also covered will be the various modes of operation of AES, which aim to provide security again different kinds of attacks. Lastly, this chapter will cover a number of optimizations in which the standard layout of AES is altered to fully harness the computational power of CUDA and GPUs.

4.2 The Rijndael Algorithm

The first step to encrypting information with AES is to generate a key schedule from the initial key using a key expansion routine. Key generation techniques are beyond the scope of this research, and as such are not covered. When appropriate, pre-generated keys are used from FIPS-197 or NIST-800-38A, or have been generated using the OpenSSL library. The size of the key schedule is dependent on the size of the initial key, with each key in the key schedule being used in a given round of the AES algorithm. Figure 4.1 shows the number of 4-byte words associated with each key length, and the number of rounds that correspond to it. The block size is constant for all key sizes.

After the key schedule is generated, the *cipher*, or encryption, function can be executed on a 16 byte block of input. Figure 4.3 contains pseudocode for the process which encrypts a block of input. First the application declares a 16 byte buffer called the ‘state’, which will then have a copy of the

	Key Length (Nk words)	Block Size (Nb words)	Number of Rounds (Nr)
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Figure 4.1: Key-Block-Round Combinations [1].

input block placed in it. The state, and in fact all blocks in Rijndael, are represented as a 4×4 *column-major* matrix. Next the AddRoundKey function is performed on the state with the 0th round key, found in Rks, which is actually just the original encryption key, from which the rest of the keys in the schedule are derived. AddRoundKey $XORs$ each column of the state matrix with the corresponding columns in a matrix representation of the round key.

(a) (b) (c)

Figure 4.2: The SubBytes, ShiftRows, and MixColumns operations of the AES algorithm.

A series of four operations takes places for each of the $Nr - 1$ rounds in the Rijndael algorithm. First, the algorithm performs a non-linear byte substitution utilizing the substitution box, or sbox, which is a pre-computed lookup table, demonstrated in Figure 4.2(a). Next, the ShiftRows operation is applied, which performs a cyclic shift of each row of the state matrix, as seen in Figure 4.2(b). Following this the mix columns operation takes place, shown in Figure 4.2(c), which multiplies each column of the state matrix with a polynomial, which is then reduced modulus $x^4 + 1$. All operations on columns are treated as polynomials in the Galois Field $GF(2^8)$. These 4 steps are repeated for $Nr - 1$ rounds, at which point the final round is performed in identical fashion, except for the MixColumns transformation which is omitted. To perform decryption, otherwise known as the *inverse*

```
void Cipher(Rks, input, output, sbox)
{
    uint8_t state[BLOCK_SIZE];
    state = input;
    AddRoundKey(state, Rks, 0);
    for (i = 1; i < Nr; i++) {
        SubBytes(state, sbox);
        ShiftRows(state);
        MixColumns(state);
        AddRoundKey(state, Rks, i);
    }
    SubBytes(state, sbox);
    ShiftRows(state);
    AddRoundKey(state, Rks, i);
    output = state;
}
```

Figure 4.3: The Rijndael Cipher Function written in C pseudocode.

cipher, the algorithms implement the inverses of the four core Rijndael operations. Algorithmically, the decryption function is similar, with a few modifications to the order of operations as well as the inverse of MixColumns requiring a different set of polynomial operations.

As covered in Section 3.3, this implementation is, while correct, not incredibly performant nor efficient. Instead, the algorithm replaces the SubBytes, ShiftRows, and MixColumns operations with a series of lookup tables, contextually referred to as $t-boxes$ or $t-tables$. These lookup tables greatly increase the performance of the Rijndael cipher at the expense of memory. Briefly covered were two possible implementations of the t-box optimization, either by utilizing four tables, or utilizing a single table which requires the application to cyclically rotate the values of each location upon retrieval. While the latter is incredibly useful for use in smaller embedded systems, the size of a GPUs available global memory is simply so large that these tables do not have an impact on device performance or memory availability. Thus, the highest performance algorithm for t-table lookups can be implemented. However, for the absolute best performance, the t-boxes must be loaded into memory which performs even better than global memory, at the expense of it being a limited resource. This is covered later in Section 4.4.

4.3 Parallelization

In Section 2.2 the paper investigated just how CUDA exposes concurrency to the developer. How these concepts are applied to AES are now going to be covered, and how the massive parallelism of GPUs can be exploited through CUDA for a very performant implementation. AES is, at its core a block cipher, which means that it operates on data of a fixed size, in this case 16 bytes. While data may occasionally come in single blocks, it is far more common to be operating on data in the kilo, mega, or gigabyte realms. This is good, because CUDA excels at running hundreds of thousands or millions of threads concurrently while operating on or utilizing large data sets. As discussed in Section 3.2, the ECB mode of AES lends itself well to parallelization, as each 16 byte block is self contained and relies on no other blocks. This means that for any input data, the implementation can launch the encryption or decryption operation on the GPU, with a single AES block assigned to each thread. It really is that simple, and with the incredibly high thread limit of the CUDA runtime, it can be assured that the implementation will run out of GPU memory long before ever reaching the limit on the number of launched CUDA threads. This processing model is most recently laid out by Patchappen et al. [16], and the earlier works of Iwai et al. [11] confirm that this is indeed the most efficient way of utilizing the GPU as opposed to 8 bytes per thread, or 1 byte per thread in each AES block.

Complications arise when trying to implement cipher block chaining, or CBC mode, for AES. As shown earlier in Figure 3.3(a), the encryption operations actually require that the ciphertext of the previous AES block be *XORed* with the plaintext of current block before it is encrypted. This means that each block has a dependence on the previous block, and by extension ensures that the encryption process is in no way parallelizable. Decryption on the other hand, is still parallelizable. Figure 3.3(b) shows that the output of the decryption processes must be *XORed* with the ciphertext input of the previous block. Since this is data that is available, decryption of data encrypted in CBC mode is perfectly viable, and is implemented identically to ECB mode, one block per thread. It should be noted that to utilize the GPU for this purpose, the implementation requires twice the space that would be utilized for ECB mode. Because no block in ECB mode relies on another, the global memory used to store the plaintext or ciphertext can also be used as an equivalent output buffer. CBC mode does not allow for this because all of the input data must be accessible, and as

```
// Cast the global memory to an array of 32bit ints, this will
    help with coalesced accesses
uint32_t* state = (uint32_t*)&ctx->Output_d[aesBlockIdx];

s0 = byteswap(state[0]);
s1 = byteswap(state[1]);
s2 = byteswap(state[2]);
s3 = byteswap(state[3]);
```

Figure 4.4: By treating the 16 byte blocks of data as 32 bit unsigned integers, the application can achieve properly coalesced memory accesses.

such the application must allocate an output buffer of identical size to that of the input memory.

There is another consideration to take into account when parallelizing AES, and that is the size of the ciphertext or plaintext and the context in which this size applies. Specifically, databases store data not in long continuous chunks, but rather they store data in pages of a fixed size, which allows for easier selective retrieval of data. When dealing with data which is encrypted with CBC mode, this changes the paradigm of what is considered parallelism. Our implementation still cannot parallelize encrypting a single page of data, as there still exists the block level dependence on previous blocks which must first be encrypted. However, this does not mean that the algorithm cannot parallelize decrypting multiple pages at the same time. Instead, the application assigns a single CUDA thread to the task of encrypting a page of data, which is not performant for a single page, but on the order of thousands or hundreds of thousands of pages scales much better. This model of working with pages of data also scales well to decryption, which can already be parallelized, so the model of one thread per block can be followed with some additional logic added into the implementation to handle the page format.

To maximize the performance of this AES implementation, the application should also utilize coalesced memory accesses whenever possible(the concept of which was introduced in Section 2.2). To do this requires working around some limitations of this particular implementation of AES. Specifically, the implementation utilizes byte arrays to store and retrieve the data. This representation is incredibly easy to work with and is intuitive as AES

```
#define byteswap(x)
(((x >> 24) & 0x000000ff) | ((x >> 8) & 0x0000ff00) |
((x << 8) & 0x00ff0000) | ((x << 24) & 0xff000000))
```

Figure 4.5: Macro used to change the endianness of data being retrieved from global memory.

utilizes the notion of a 16 byte block. It is, however, somewhat troublesome for achieving optimum memory throughput, as CUDA does not perform coalesced accesses for any data type which is less than 32 bits [17]. This causes an issue, as it means that all 16 bytes in each thread must be accessed individually in their own memory transaction. This is very expensive, and creates a bottleneck in the application, with the Nvidia Visual Profiler reporting only a 36.5% efficiency when loading each block from global memory onto the streaming multiprocessor. This issue is solved by casting the block of data from a byte array to an integer array, and then accessing the data as 32 bit unsigned integers, as seen in Figure 4.4. This works very well because there are exactly four 32 bit unsigned integers out of 16 bytes. Our CUDA implementation not only utilizes the optimization for loading data from global memory, but also to improve copy times from the streaming multiprocessor to global memory when the encryption or decryption for a given block is done.

When implementing this optimization it is required to understand endianness and handle it appropriately, especially for this application which casts arrays of bytes into integers. Assume there is a byte array containing the values [0x00, 0x01, 0x02, 0x03] in hexadecimal. When this is casted to an unsigned integer, the integer becomes the hexadecimal value of 0x03020100. The first memory location in the byte array, which stores hex value 0x00, has gone from being the most significant byte to the least significant byte. This shows that the data representation of the machine is litte-endian. This is the opposite of what the T-box implementation of AES expects, which expects the most significant byte of the byte array to correspond to the most significant byte of the unsigned integer. Therefore the ordering of the bytes stored in the integers must be swapped. This can be done by utilizing a simple macro, shown in Figure 4.5 to swap the byte order around in the variable. As such, in the Figure 4.4 example, it can be seen that the bytes of each 32 bit unsigned integer from the byte array are swapped to create an

unsigned integer which is then assigned to a state variable, stored in s0-3. Similarly, it is necessary to run the same macro on the unsigned integers representing the state when utilizing the same concept to cast the integers into a byte array representation to store in global memory.

This creative use of the type system that C exposes allows the implementation to completely remove the reported bottlenecks caused by inefficient global memory loads, and fully harness the memory bandwidth of the GPU. Overall, the speed-up when implementing this was observed to be anywhere from 15-25% for the actual GPU data processing, as this does not effect memory copy times. This shows how very important it is to consider all possible bottle necks in a system and how they can best be accounted for and worked around if necessary. This speed-up is realized even though there is added execution time overhead due to the byte-swap operations.

4.4 T-Table Lookups in Shared Memory

As mentioned in Section 2.3, this CUDA implementation of AES utilizes shared memory, located on the streaming multiprocessor executing a given thread block. It aids in reducing global memory accesses to smaller amounts of commonly used data, leaving more available bandwidth for accessing segments of memory which are too large to load into shared memory. This is utilized primarily for lookup tables and round keys. Below it is shown how the implementation utilizes shared memory in the encryption function of the CBC mode of operation. The algorithm allocates a 256 byte element buffer *sbox_s* for storing the sbox, as well as four 1024KB buffers for the T-Box lookup tables. This utilizes 4352KB per thread block for the various lookup tables, with an additional an additional 256 bytes for round keys (shown in Section 2.3), for a total of 4608KB per thread block of shared memory utilization.

This certainly adds some complexity to the application, as not only does the kernel have to allocate this memory and load data into it, but it must add checks to ensure that the bounds of the shared memory region are not exceeded (line 5) as well as accounting for the situation where there are not enough threads for each of them to load a single chunk of memory (line 7 and the proceeding for-loop). However, this added complexity is highly rewarding due to the sheer number of memory accesses performed on these regions.

```
__shared__ uint8_t sbox_s[SBOX_SIZE];
__shared__ uint32_t te0_s[SBOX_SIZE], te1_s[SBOX_SIZE],
                    te2_s[SBOX_SIZE], te3_s[SBOX_SIZE];

if (threadIdx.x < SBOX_SIZE) {
    uint8_t i = ceil((float)SBOX_SIZE / blockDim.x);
    for (uint8_t j = 0; j < i; j++) {
        uint8_t idx = threadIdx.x * i + j;
        sbox_s[idx] = ctx->sbox_d[idx];
        te0_s[idx] = ctx->te0_d[idx];
        te1_s[idx] = ctx->te1_d[idx];
        te2_s[idx] = ctx->te2_d[idx];
        te3_s[idx] = ctx->te3_d[idx];
    }
}
__syncthreads();
```

Figure 4.6: An excerpt of code which shows how the application loads lookup tables into shared memory.

```
t0 = k0 ^ te0_s[(uint8_t)(s0 >> 24)]
        ^ te1_s[(uint8_t)(s1 >> 16)]
        ^ te2_s[(uint8_t)(s2 >> 8)]
        ^ te3_s[(uint8_t)(s3)];
```

Figure 4.7: An example of the frequent usage of t-tables, used to justify storing them in shared memory.

Figure 4.7 is an example of the table lookups which replace the standard AES round operations, which performs an XOR of part of the round key, *k0*, with values extracted from the lookup table based on input data found in the variables *s0-3*. This example operation is performed four times, once for each 4 byte segment of data in the 16 byte AES block, and that set of operations is then performed either 10, 12, or 14 times per 16 byte block of data depending on key size. The data stored in s0-3 will change for every thread, and therefore it is not an ideal candidate for loading into shared memory. The data found in *te0-3_s* shared memory buffers however is accessed often but is static and as such is an excellent usage of the limited shared memory. The importance of shared memory is extensively covered in numerous previous

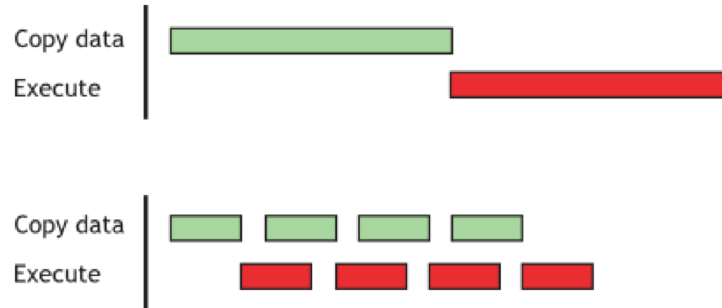


Figure 4.8: A demonstration of how to reduce runtime by overlapping copies and kernel executions [6].

papers on CUDA AES implementation, as it plays an important part not just in the AES algorithm, but in any GPU implementation [8][12][16].

4.5 Asynchronous Memory Copies and Kernel Launches

CUDA provides the developer with a number of functions so as to allow for concurrent execution of memory transfers to and from the GPU, and kernel executions on the GPU. These concurrent operations operate over what are known as *streams*, which are asynchronous channels for pipelining these concurrent operations in a particular order. Think of each stream as a *first-in-first-out* queue. When an operation is added to a particular stream, the CUDA runtime guarantees that the operations will be run in order, however it does not guarantee the order of operations between streams automatically, the programmer must utilize stream synchronization methods themselves.

For the purposes of this research, the choice was made to split the input plaintext or ciphertext into their own self contained entities for the purposes of utilizing CUDA Streams. The data is divided into programmer specified chunk sizes, and for each chunk the memory copies and kernel launch operations are placed into a different stream. The number of available streams is also chosen by the programmer, with the application choosing the streams from a set of streams in a round robin fashion. As a result of having multiple streams through which data can be copied to and from the GPU, the implementation can start to leverage and benefit from significant time saving features by way of overlapping data transfers with kernel executions.

The CUDA runtime allows for data to be copied to and from the GPU at the same time as a kernel is running on the GPU. Additionally, memory copies are full duplex, meaning that data can be copied bi-directionally to and from the GPU simultaneously. This is an excellent property of the PCIe bus which is exposed by CUDA, and as shown in Figure 4.9 allows for significant overlap of operations.

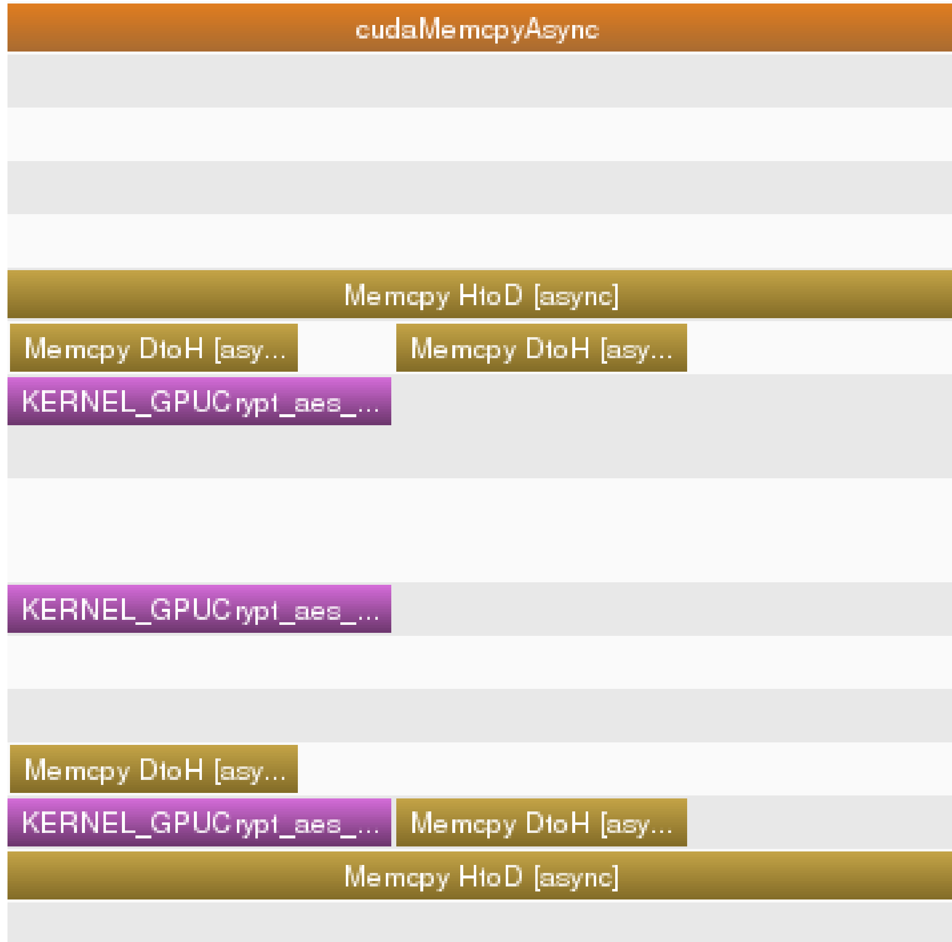


Figure 4.9: A visualization of overlapping memory copies and kernel executions, generated with the Nvidia Visual Profiler (NVVP).

This demonstrates how CUDA streams can be used to overlap copies to and from the device with data processing to reduce overall runtime. Figure 4.9 clearly shows the “Memcpy HtoD” and “Memcpy DtoH” (host to device

and device to host respectively) executing at the same time as the KERNEL function calls. More importantly, the overlap of significant data transfer begins to hide the runtime cost of copying data to or from the GPU by way of performing it at the same time as the kernel execution. This is a case of utilizing all available resources at all available times. When the kernel is running on the GPU, it is processing data that must already be on the GPU. This leaves the mechanisms for copying data to or from the GPU unused. This idea is used to produce empirical data which shows the sizes of data chunks that are most optimal for ensuring that the GPU multiprocessors are operating as often as possible, and that the PCIe bus is being saturated with data copies as often as possible.

Chapter 5

Benchmarking and Results Analysis

5.1 Test Machine and Environment

This chapter is an analysis and examination of the experimental results of comparing the GPU implementation with OpenSSL running on the CPU. As mentioned in Section 3.4, the CUDA AES solution is benchmarked against the OpenSSL cryptographic library, on computer systems which support the AES-NI hardware accelerated instruction set. OpenSSL is the definitive standard for an open source, commonly utilized and widely accepted library for cryptographic operations and as such is the best comparison for determining whether the implementation is capable of comparable performance. The implementation is tested using the computer system outlined in Table 5.1, which is a respectable, multi-core workstation computer with a high performance Nvidia gaming GPU running Debian Sid and utilizing CUDA 7.0, the latest available in the Debian repositories.

Components	**Test Machine**
CPU	AMD FX 8350 (8 Cores)
RAM	16GB DDR3 @ 1600Mhz
GPU	Nvidia GTX 780 3GB (2304 CUDA Cores)
Operating System	Debian Sid
CUDA Version	7.0

Table 5.1: System specifications for the test platform.

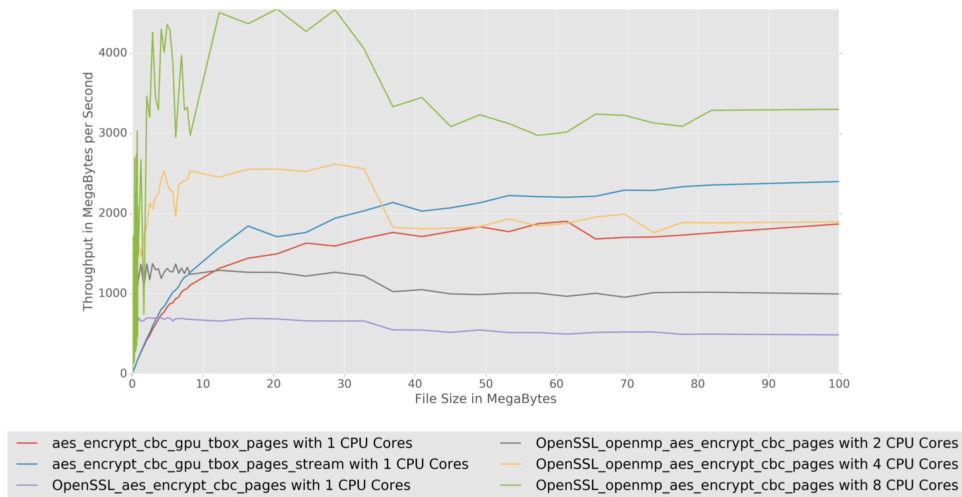
For the testing, the benchmarks collect a number of data points for a range of page sizes, and utilizing a number of different runtime configurations for the number of streams available and the number of pages copies in each stream. The OpenSSL benchmarks are also performed in a multi-core

environment, as it could be argued that utilizing more cores of a CPU is far easier to do that installing a GPU in a server, and using custom software to achieve the performance gains. All throughput results are the average of 20 runs of identical configuration at a given amount of paged data. The tests run in groups of 1-10, 10-100, 100-1000, 1000-10000 and 10000-50000 pages, and the number of data pages after 20 iterations is increased by 1, 5, 50, 500 and 5000 for each range respectively. Each page is a fixed 8192 byte page, although this library supports larger page sizes which could be tested in the future. Additionally the overhead for memory allocations is largely excluded from these benchmarks because in a production setting, this library would need to be modified to utilize manually managed pools of memory rather than allocating memory on each run, as allocations especially on the GPU can take very long amounts of time. It must also be noted that the data which is used for the testing is generated from the /dev/urandom pseudorandom generator from the Linux kernel. This ensures that all of the benchmarks are outputting results given the absolute worst case input data, as random data is much less likely to contain patterns which could be exploited by various caching schemes at the L1, L2, or L3 cache levels of the GPU.

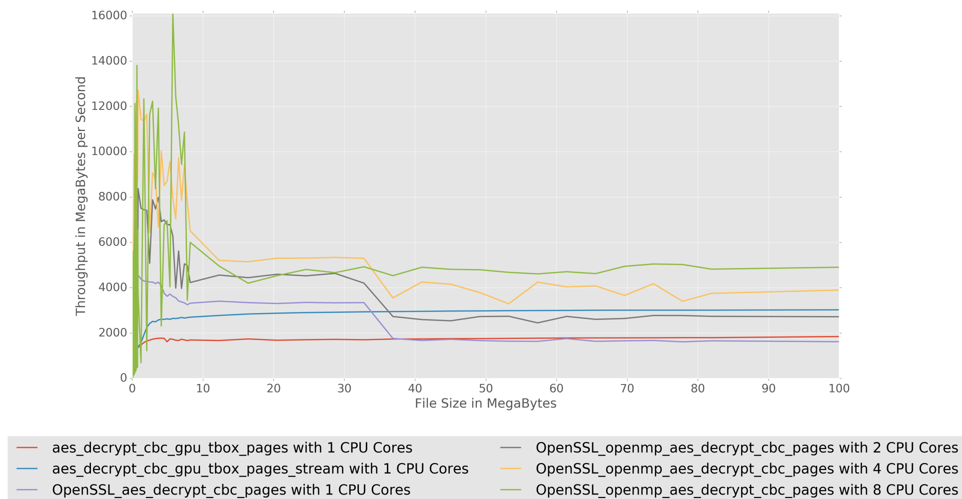
5.2 AES CBC Mode of Operation with Paged Data

Paged data is one of the core ideas of this paper, and specifically how it can be applied so the encryption of data utilizing the CBC mode of operation can be parallelized. Seen in Section 3.2, CBC mode is inherently not parallelizable, as each block relies on the output ciphertext of the previous block. But with paged data this possible by parallelizing not the block level decryption, but the page level decryption.

Observe in Figure 5.1(b), that the highly optimized AES implementation in CUDA arriving at the equivalent performance of 2 and 4 CPU Cores by about 50MBs worth of paged data, and well over one thousand megabytes per second faster than an implementation which does not utilize streams. Figure 5.1(a) paints an even more impressive picture, with the AES implementation holding its own against OpenSSL utilizing 8 CPU cores. It is however quite interesting to see that it is not performing exceptionally higher than the AES implementation which does not utilize data streams.

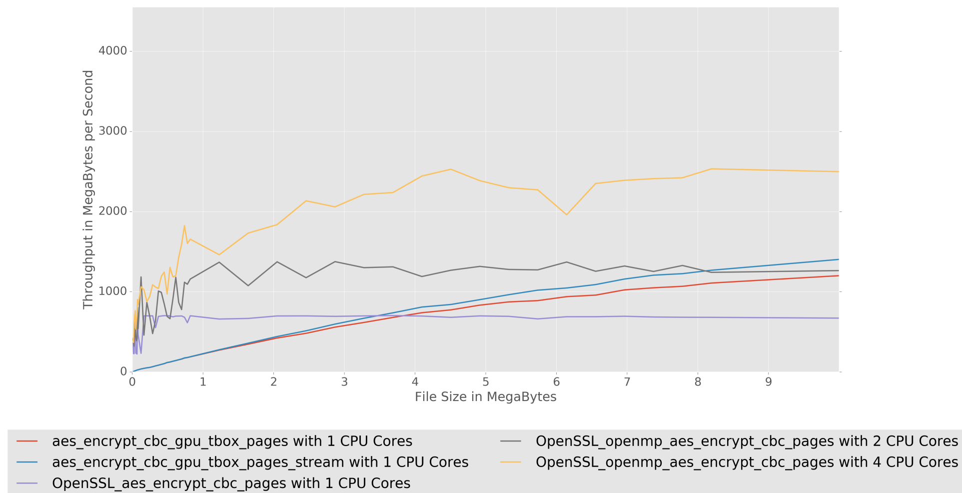


(a)

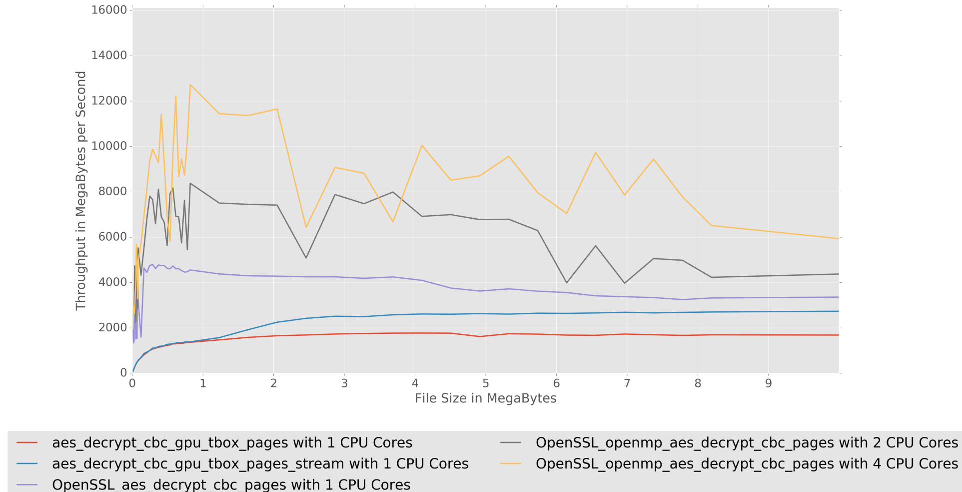


(b)

Figure 5.1: AES CBC Encryption(a) and Decryption(b) with Paged Data



(a)



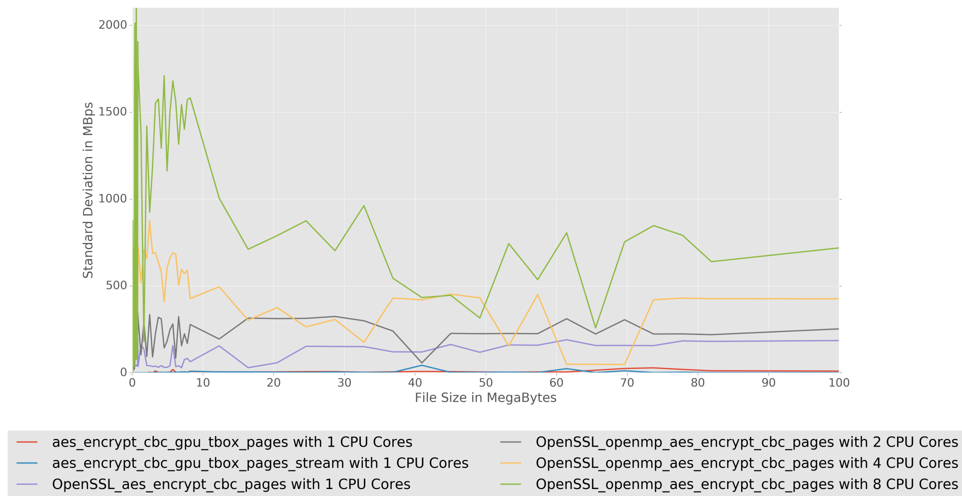
(b)

Figure 5.2: AES CBC Encryption(a) and Decryption(b) with Paged Data

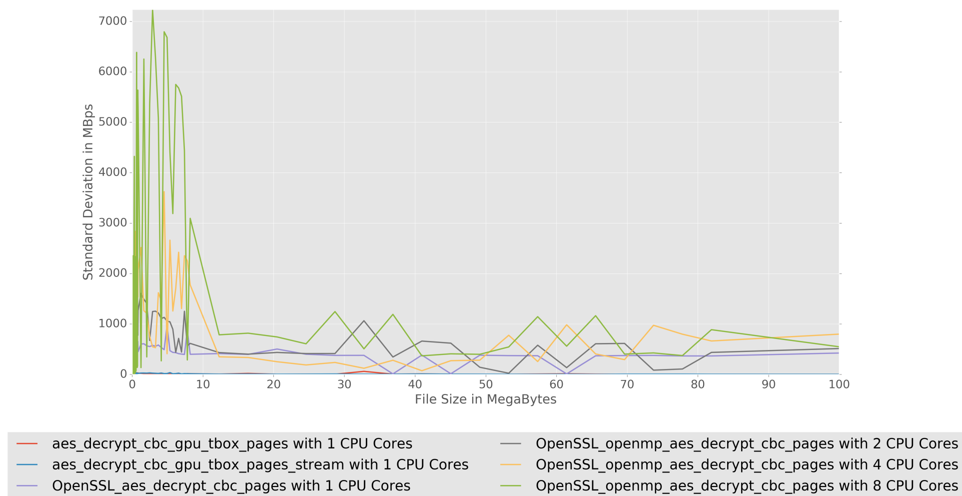
Figure 5.2 clearly shows what a number of other researchers have missed in the past, which is that utilizing the GPU for small amounts of data encryption is not, in fact, faster in any way than utilizing the CPU hardware acceleration. The GPU has a "warm-up" to its performance benefits. This makes sense due to the additional overhead of copying data to the device, even if it utilizes overlapping data transfers, along with overhead that comes from synchronization of threads while loading data into shared memory, and the time it takes to launch and terminate the kernel on the device itself. Recalling back to Section 2.2, it was also mentioned that GPU threads, individually, are much slower than a CPU and therefore utilize massive parallelization to overcome this. This concept holds true, because when dealing with smaller amounts of paged data, there is less data to parallelize, and therefore the effects of it will be lessened. It is also important to once again bring up the topic of CPU pipelining, as it ties in very nicely to the comparison between CPU and GPU implementation. For encryption with AES-NI, like the CUDA implementation, is going to inherently be serial for a given block, whereas the CPU can parallelize the launching of decryption AES-NI instructions at a low level. What this means is that with encryption, the CUDA implementation is matched with OpenSSL for parallelization limitations, whereas with decryption the GPU is competing with not only multi-core processing, but with parallelization in instruction pipelining with the AES-NI instructions.

Filesize (Bytes)	GPU (Mbps)	CPU w/ 1 Core	2 Cores	8 Cores
8192	3	421	402	5
16384	7	229	378	13
24576	10	229	323	717
122880	38	233	1186	1241
163840	46	699	460	1723
1228800	278	661	1370	2677
1638400	360	668	1077	747
12288000	1576	660	1294	4509
16384000	1846	694	1269	4370
327680000	2660	372	846	3167
368640000	2673	363	854	3283
409600000	2689	339	782	3114

Table 5.2: GPU Implementation with Streams vs OpenSSL performing Encryption. All speeds are in MBps.



(a)



(b)

Figure 5.3: The standard deviation of throughput for the AES CBC implementation and OpenSSL with Paged Data

It is also necessary to address what appears in the results to be wild thrashing from the multi-core OpenSSL benchmarks. From Figure 5.1 it should be very clear that the average runtime fluctuates greatly between the various amounts of data processed at a time. This is made explicitly clear in Figure 5.3, which charts the standard deviation of encryption and decryption time in both the GPU and OpenSSL tests. Our GPU implementation clearly has much lower variance in throughput between each run, whereas the CPU is highly erratic, due to the fact that despite the assistance of hardware acceleration from the CPU, scheduling of the low level instruction execution is subject to being suspended and resumed by the operation system kernel through context switching and other scheduling operations which produce non-deterministic fluctuations. This is an incredibly important property that very much needs to be highlighted, because this could be received very well in environments which demand low and consistent latency for their operations; Communications, financial systems, and real-time operating system applications immediately come to mind, but would necessitate being explored further. Table 5.2 contains a sampling of data from the output data set of the CBC paged data testing, and numerically shows the erratic nature of utilizing AES on the CPU. Table 5.3 Contains a sampling of data from the same data sizes, but for the decryption. This clearly shows how even with even a single CPU core, the CPU decryption runs considerably faster than the encryption, thanks to CPU pipelining of the AES-NI instructions.

Filesize (Bytes)	GPU (Mbps)	CPU w/ 1 Core	2 Cores	8 Cores
8192	80	2159	2234	6
16384	157	1338	2776	1418
24576	224	1428	2669	2293
122880	701	1599	4328	4468
163840	869	4635	5523	5626
1228800	1574	4382	7507	693
1638400	1925	4299	7449	12335
12288000	2779	3409	4562	4959
16384000	2843	3346	4447	4204
327680000	3071	1443	2553	4486
368640000	3071	1381	2548	4480
409600000	3074	1390	2690	4647

Table 5.3: GPU Implementation with Streams vs OpenSSL performing Decryption. All speeds are in MBps.

5.3 AES CBC Mode of Operation

Our library also implements AES CBC mode for arbitrary files which do not follow a paged data format. Encryption cannot be implemented at all, as it is completely single threaded, and as such is not only abysmally slow when running on a GPU, but for any data larger than about 8KB can actually cause the process to become uninterruptible by the operating system's scheduler, which can lock the entire system until the kernel watchdog terminates the process. Investigation was performed for this, and the issue seems to stems from the Nvidia graphics driver, but no acceptable fix was found outside of simply not running an application in such a serialized fashion. Decryption can be implemented however, as that is a very well understood parallelization. For the sake of continuity, the results are included here as Figure 5.4, which compares OpenSSL on a single core to the optimized implementation. Note that the application does not implement CUDA streams for overlapping data transfer, but it would be possible to implement without much additional effort. It also compares to the naive implementation of AES which does not include the t-box lookup table optimization, as an example of just how much more performant AES is when fully optimized.

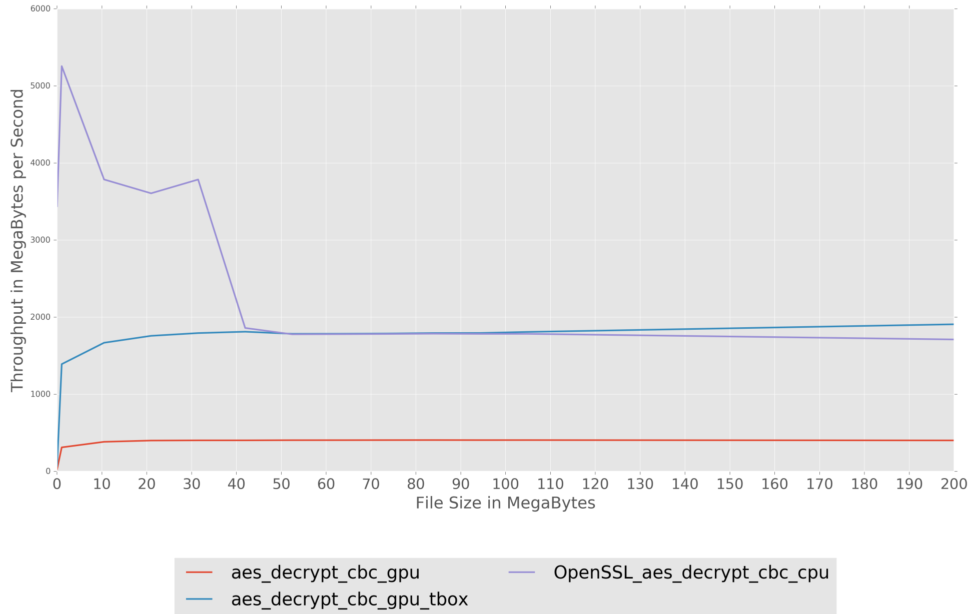
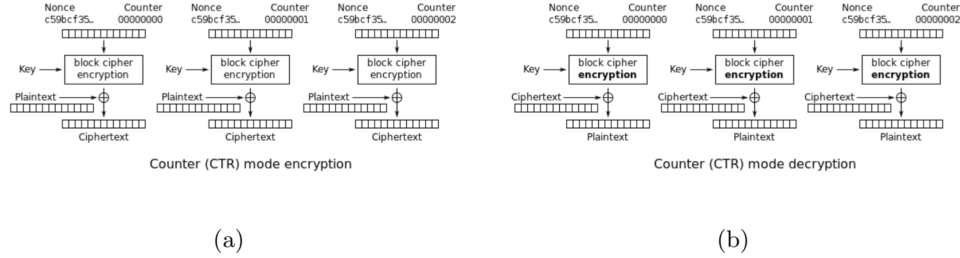


Figure 5.4: The standard deviation of throughput for the AES CBC implementation and OpenSSL with Paged Data

5.4 AES CTR Mode of Operation

While the Counter mode of operation (CTR) for AES was not covered earlier, it was none the less implemented as part of this research, although it was not a cornerstone. To briefly cover CTR mode, CTR utilizes an initialization vector called a nonce. A nonce is a starting value which is encrypted with the AES block cipher algorithm identically to ECB mode of operation. The output of this is then $XORed$ with a block of plaintext to retrieve the ciphertext, and similarly $XORed$ with a block of ciphertext to retrieve plaintext. However, the same nonce is not used continually, but is only used as a starting point. For each block after the initial block, the nonce is incremented by 1, and then encrypted and $XORed$ with the next block of plaintext or ciphertext. This is demonstrated in Figure 5.5, which depicts both encryption and decryption of plaintext and ciphertext utilizing CTR mode. Notice that both encryption and decryption use the *encryption* algorithm. Figure 5.6 demonstrates a basic implementation of CTR which operates on arbitrary block sizes, while Figure 5.7 demonstrates the possibilities for a paged data system. CTR mode is, alongside CBC, considered a



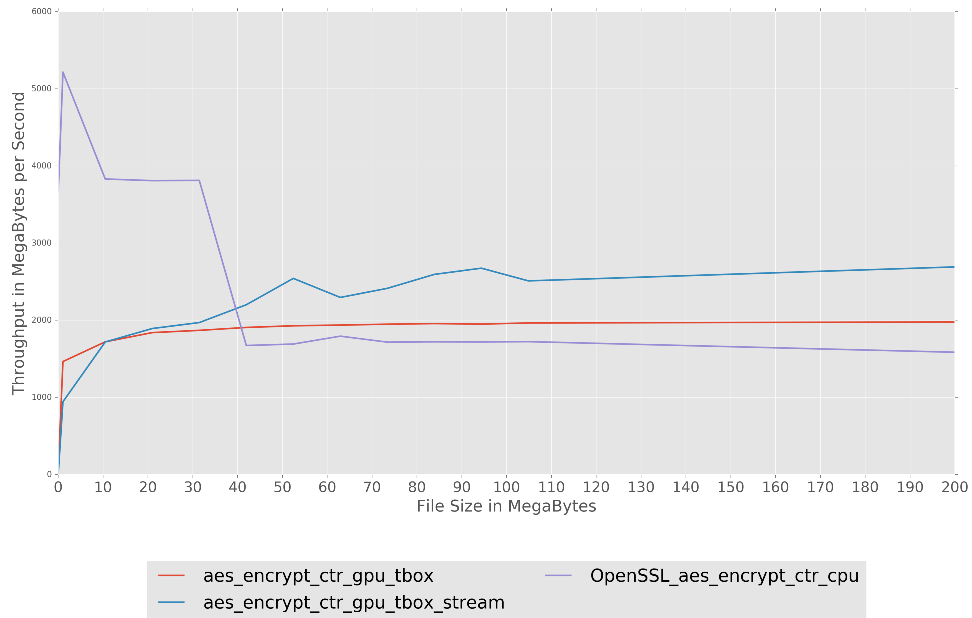
(a) (b)

Figure 5.5: The Counter (CTR) mode for encryption (a) and decryption (b)

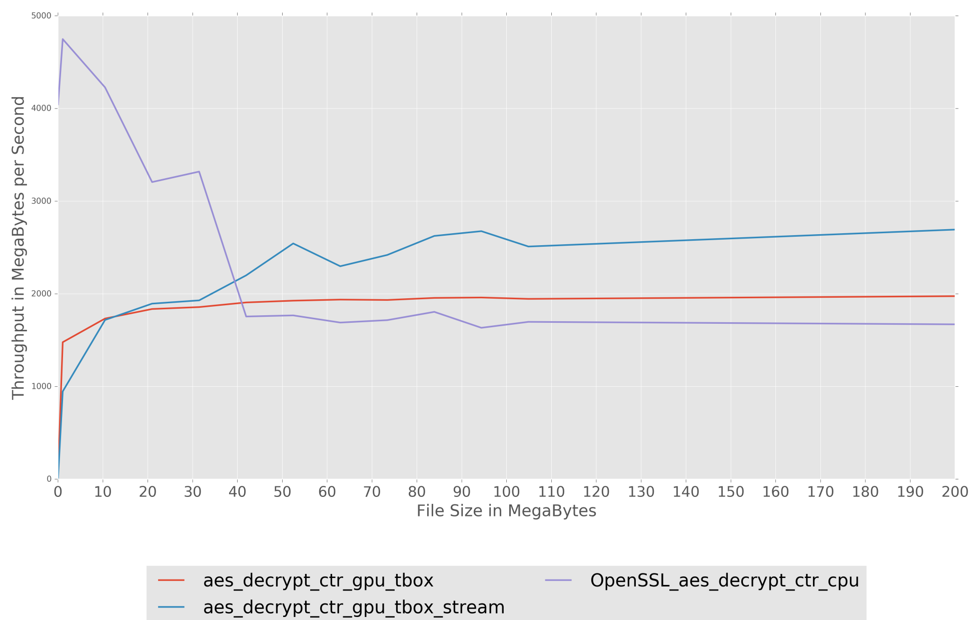
fairly resilient cipher system, and because of this “Counter” property, is very commonly in use as a stream cipher, which allows for the nonce to be transmitted along with the connection establishment handshake, at which point both the sender and receiver simply increment it for every block worth of data received. This makes CTR an excellent choice for encrypting network traffic and other communications. CTR mode is inherently parallelizable at all levels, because each block is operated on independently, sharing a common starting nonce.

Similar to CBC mode, Figure 5.7 shows that the AES-NI enabled OpenSSL takes an early lead, with CTR mode eventually settling in between 2 and 4 core OpenSSL performance at about 40MBs of paged data, both for encryption and decryption. This is expected as they are essentially identical algorithms just being called on different data. Also seen is the same erratic throughput from OpenSSL at the smaller data sizes, thanks to the operating system scheduler, and it is observed that the GPU maintains its consistency with incredibly low variance in throughput between runs. Once again however, the GPU is competing with not only multi-core parallelism but instruction level parallelism via pipelining.

Overall, what is presented in this section is quite clear. AES on the GPU is simply not viable in a data processing environment which uses small amounts of data, because OpenSSL’s hardware acceleration through AES-NI is simply too fast for the highly optimized CUDA implementation to compete. The overhead of copying memory and launching the GPU kernel is too great to overcome with smaller data sizes. However, in scenarios where a processing node must deal with large amounts of data which can be processed in batches, where this implementation excels, then GPUs are an

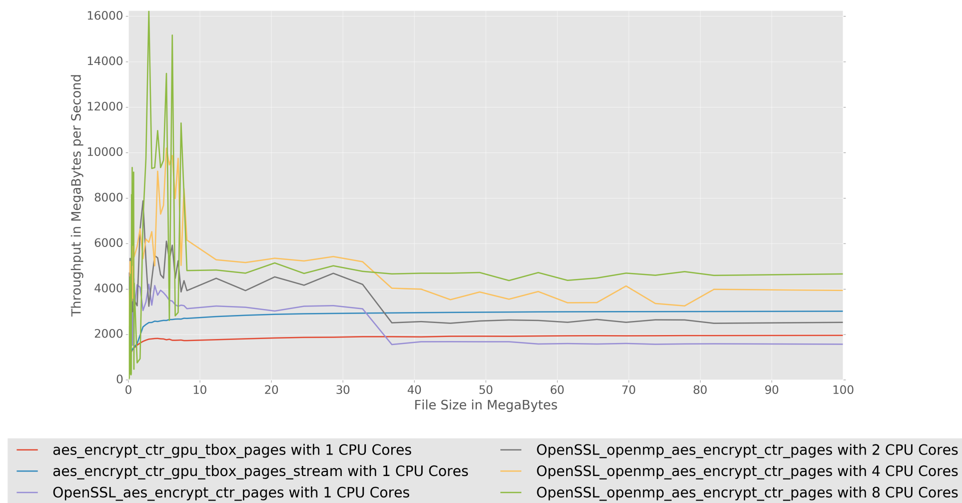


(a)

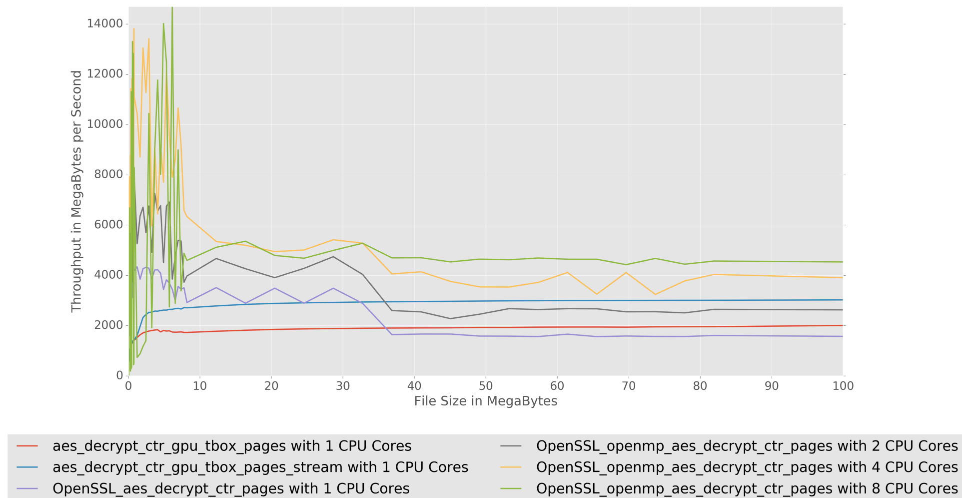


(b)

Figure 5.6: AES CTR Encryption(a) and Decryption(b)

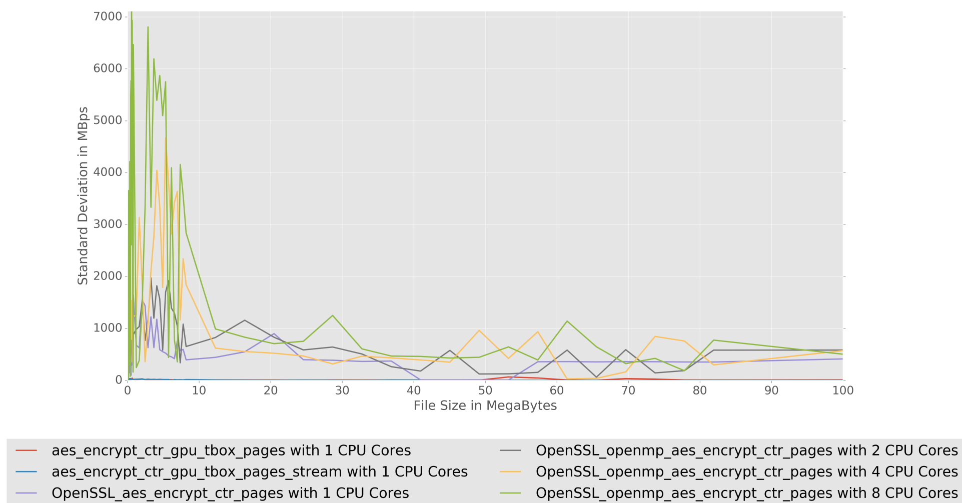


(a)

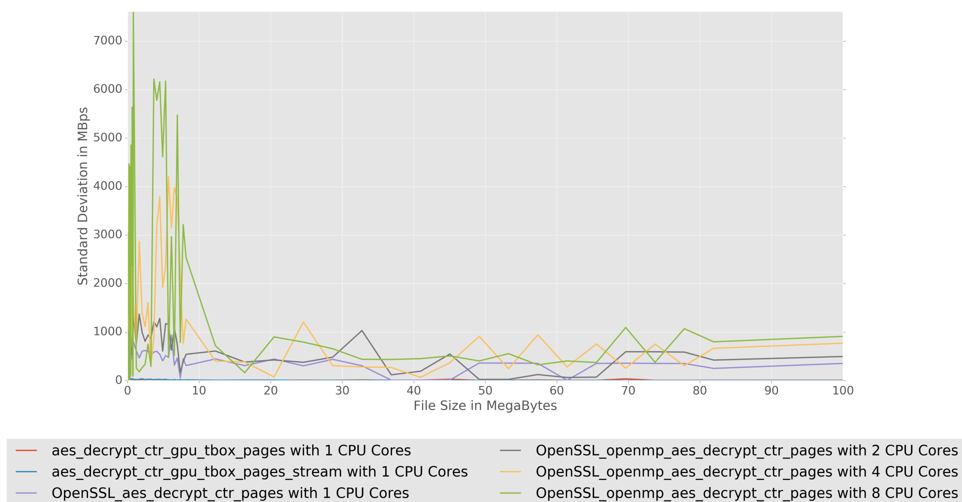


(b)

Figure 5.7: AES CTR Encryption(a) and Decryption(b) with Paged Data



(a)



(b)

Figure 5.8: The standard deviation of throughput for the AES CTR implementation and OpenSSL with Paged Data

incredibly viable solution for performance gains.

5.5 Applications Analysis and Further Optimizations

This section of the paper serves the purpose of expanding on some interesting points which were found in the data collected which are not necessarily obvious or intuitive at first glance. GPUs are complex devices, and their strengths and weaknesses are highly dependent on the data involved, what guarantees are needed by the programmer, the algorithms required to achieve goals, and so on. A programmer needs to "massage" an application into the most efficient and performant configuration. Expanding on what has been observed and researched is an important part of learning and looking to the future for new possibilities.

Data Processing without Copy-To-Device Penalty

The current set of benchmarks look at the time to copy data to the GPU, process it, and copy it back to the host as one complete system. The data resides on the host, and it needs to move to and from the GPU so that it can to do work on it, which incurs significant amounts of delay. In fact, in most cases the data is in transit for anywhere up to double the actual processing time. Instead of treating this as a problem that is inherent to the fact that the GPU is a co-processor attached to a computer, the programmer should treat the GPU as a computer itself. Direct Memory Access, or DMA, allows the programmer to treat the GPU in this way. DMA is an incredibly useful feature which allows for any device connected over PCIe, to directly copy data to another PCIe attached device by circumventing the CPU and directly accessing the memory controller. This reduces latency and increases memory throughput considerably. By utilizing such a mechanism, a few different goals can be achieved.

First, the application can now copy data directly from disk to the GPU. Instead of reading pages of data from a database, or indeed any arbitrary data from disk, into host memory and then to the device, the application can copy it straight to the GPU without incurring extra CPU load or delay. With the advent of NVMe storage devices, it is in fact not uncommon for a single flash storage device to be able to read at speeds of 4GB per second, with higher speeds possible with various striped storage configurations. An

application can also write back to disk at these speeds, as the CUDA runtime can copy data straight from GPU memory directly to disk via DMA as well. This concept and its benefits would be a very interesting research topic to pursue.

DMA is not restricted to data storage devices, as there are many uses for high speed devices which attach via PCIe. Infiniband high speed network interconnects are incredibly common in environments which require high speed and/or low latency connections for data communications. These cards attach to a computer or server via PCIe, and these are actually well understood and commonly implemented with CUDA by way of Nvidia's *GPUDirect RDMA*, which allows for direct communication between two PCI devices. This opens up a myriad of possible uses for the CUDA AES implementation which can bypass a significant amount of copy expenses. Assume there is a server which connects to a network over infiniband, which acts to proxy data, and at the same time encrypts or decrypts traffic depending on the direction it flows in the network, similar to how a VPN operates, but for the sake of simplicity assume arbitrary data. When that data arrives at the server, it must be copied from the PCI infiniband card to host memory, where it is processed with a highly variable amount of latency, and then copied from host memory onto either the same or another interconnect device to be passed on to its final destination or another hop in the data flow. Whether this data is processed by the CPU, or the GPU, it must be copied both onto and off of the intermediate node. Therefore, data can be copied directly to the GPU, the data operated on, and then copied from the GPU to another PCI device without any need for the CPU besides scheduling kernel launches. It can be confidently assumed that the time taken to copy from the network device to host memory is the same to copy to the GPU device, and similarly it can be assume the time to copy from the CPU to the network device is similar to copying from the GPU to the network device.

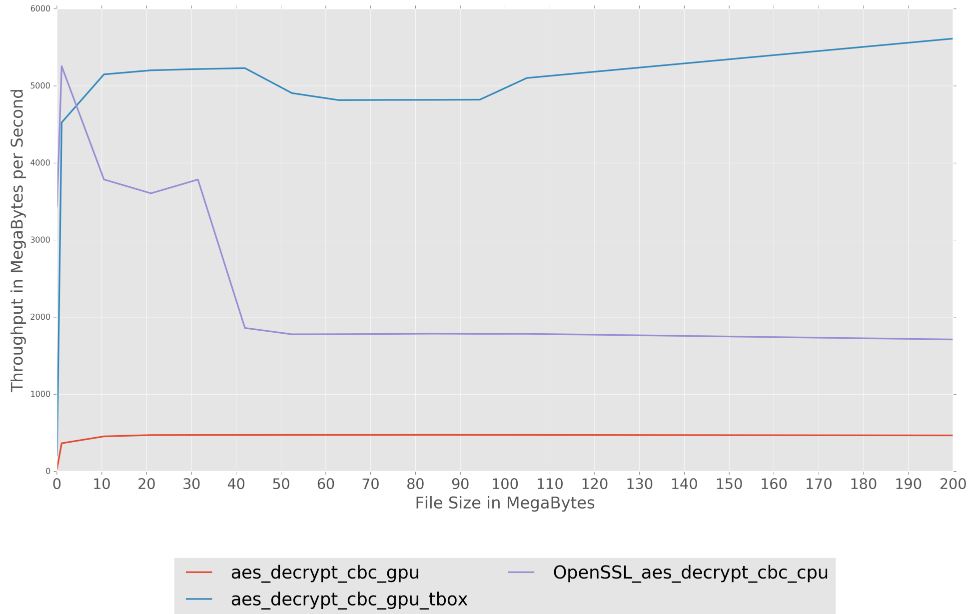
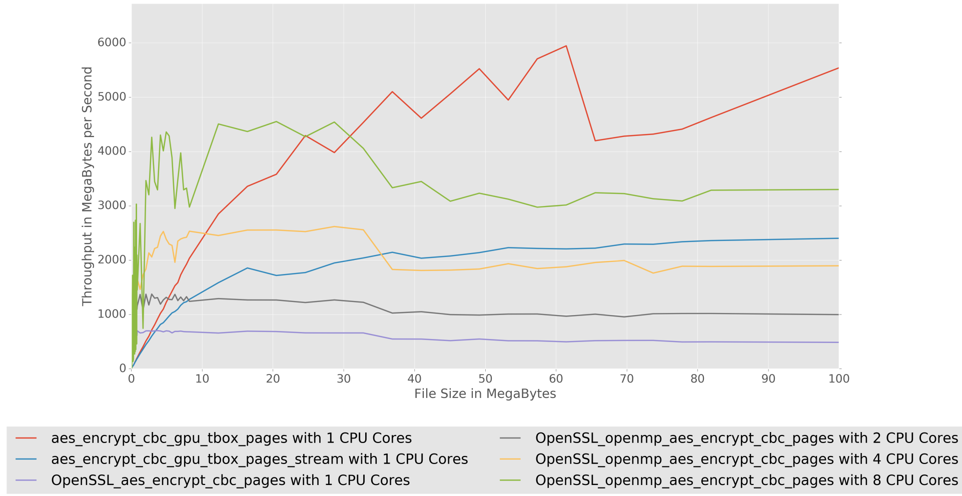


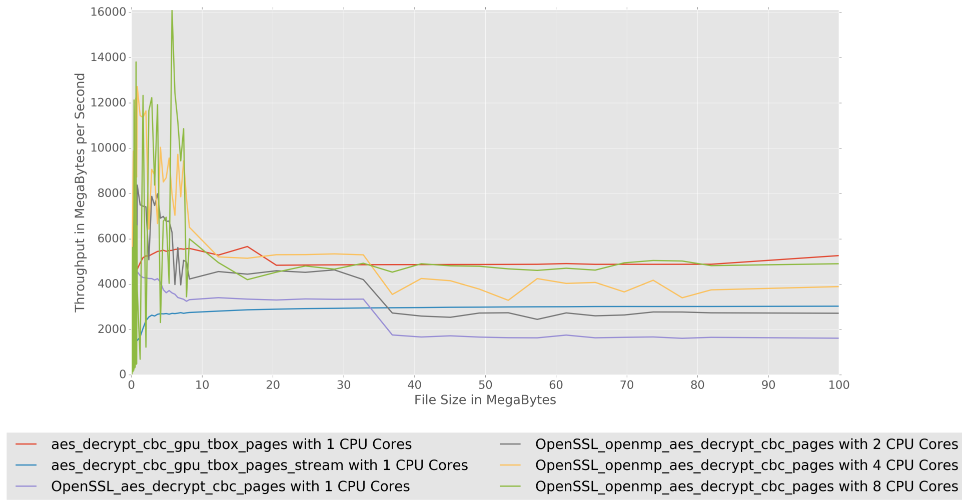
Figure 5.9: AES CBC Decryption with Copy Time Excluded

By using this assumption, it is logical to start thinking about what the processing throughput possibilities are if the cost of copying data to and from the GPU was completely eliminated. It is not necessarily a completely fair comparison to compare the CPU to the GPU when the GPU is, in the test case, inherently at a disadvantage because it must wait the time required for the data itself to arrive on the device, while the CPU can begin working immediately to process the data stored in host memory. In the world of high speed data accesses, the time to copy from a high speed device to GPU memory will be about the same as the time taken to copy to host memory, and the opposite is true as well. Therefore, the throughput that can be achieved with the CUDA GPU implementation when it does not have to make up for data copy times when compared to the CPU should be examined.

When actually comparing like systems, it shows that the GPU solution is a lot more attractive for high performance systems. Figure 5.9 shows clearly how large the difference in throughput is between the CPU and GPU even with smaller input data sizes, with the GPU overtaking the CPU at around 5MBs, and holding its own from quite early on. Additionally, the data shows that there is a linear increase in throughput as data sizes increase, so this

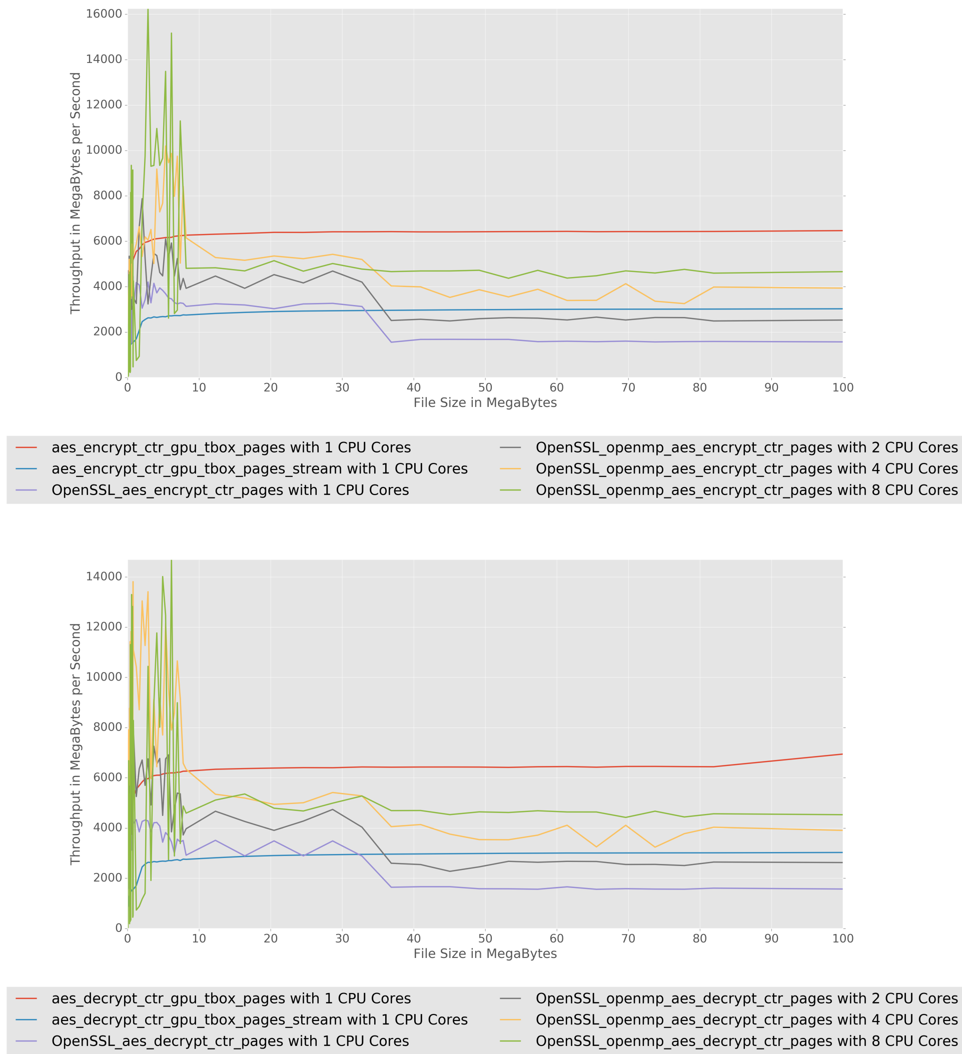


(a)



(b)

Figure 5.10: AES CBC Encryption(a) and Decryption(b) with Paged Data and Copy Time Excluded



(b)

Figure 5.11: AES CTR Encryption(a) and Decryption(b) with Paged Data and Copy Time Excluded

solution scales well. Figure 5.10 shows a picture similar to what has been seen before, except in this case the GPU surpasses the 8 core CPU running OpenSSL with AES-NI instructions in CBC mode, and even with smaller data sizes it performs remarkably. CTR mode is even better, as is expected from a highly parallelizable algorithm. Figure 5.11 shows the implementation far surpassing the 8 core CPU in both encryption and decryption.

Incredibly interesting to note, although not entirely unexpected, is that these latest charts shows that the implementation which uses streams is far slower than the implementation which does not. Streams are an effective way of hiding the run time expense of copying data to and from the GPU, but when that data already resides on the GPU, utilizing streams is actually quite slow and not particularly performant, as the application then has a large number of small kernels launched, running, and exiting instead of scheduling a single monolithic kernel. It should however be noted that the currently displayed charts are not a perfect representation of the performance of streams without a data copy expense, as Nvidia provides no way to measure the length of time it takes for operations in a stream to complete. This is none the less understandable as the entire point of streams is to be asynchronous. Therefore, in this case it is essentially comparing the previous data to only the non-stream AES CUDA implementation, but with the copy times removed for the non-stream version.

Filesize (Bytes)	GPU Encrypt (Mbps)	GPU Decrypt (Mbps)
8192	3	234
16384	7	458
24576	10	685
32768	13	920
122880	39	2499
163840	47	2837
1228800	313	4943
1638400	409	5177
12288000	2852	5290
16384000	3361	5668
122880000	6704	5751
163840000	5146	4892
204800000	6006	5756
245760000	6718	5757
286720000	6345	5757
327680000	6345	5759
368640000	6727	5756
409600000	6434	5757

Table 5.4: Table of raw processor throughput for the GPU implementation of AES CBC mode with paged data and data copy time excluded.

Another unexpected finding is found by examining the raw data which constructed Figure 5.9. Previous to this, the data was being analyzed with the Python aggregation tool Pandas, and a failure to manually inspect the data would have meant a loss of interesting data. Table 5.4 clearly shows that the GPU Encryption in CBC mode for paged data hits a wall around 6700MBps, whereas the Decryption maxes out at 5700MBps. Both algorithms are processing the same amount of data, and in the same paged format, but the encryption operates with one page per thread, and the decryption operates on one block per thread. Therefore, there needs to be additional investigation into why this is, and perhaps even implementing the decryption algorithm as one page per thread as well to conclude whether launching one thread per block of data at that scale is simply too many threads to effectively schedule, giving the one thread per page scheme a computational edge. Decryption launches one thread per block, meaning that for every page the implementation launches 512 threads (8192 bytes ÷ 16 byte blocks) rather than one block per page. The possibility may then exist that each thread processing multiple pages may be even faster than a

single thread per page given that some additional work is done to spread out the thread blocks across as many streaming multiprocessors as possible.

Chapter 6

Conclusion

In summary, what has been implemented is a very important step forward in the utilization of GPUs and CUDA in AES and perhaps possibly other cryptographic systems in the future as well, and provides excellent evidence to substantiate claims made in this paper. It is a very important step to realize that except for very large bulk data processing situations the GPU solution is not nearly as viable as simply utilizing the CPU for encryption or decryption. The run time penalty incurred by needing to move data to and from the GPU device is very costly, and while it is not so costly that some workloads would not still benefit from it, this fact also allows examination to take place from a step back, and to think about what sort of benefits could be realized if the architecture of the application were changed to remove some of these performance costs, by for example copying data straight to the GPU via DMA access. By changing the focus from traditional computing architecture to treating the GPU as a device which requires only minimal kernel scheduling from the CPU, it is possible to increase the performance gains, although perhaps at the expense of narrowing the possible fields which the implementation is applicable. This research has answered a number of important questions, as well as addresses a number of glaring issues in previous researchers efforts, which completely ignored the abundance and almost standard availability of hardware acceleration at the CPU level, even in most cheap consumer CPUs.

6.1 Future Work

There are a number of interesting concepts which were not explored or are only partially looked at for this research, but should perhaps be investigated in the future. At the forefront of these is utilizing the possibilities for DMA access to the GPUs global memory to build a service which does not utilize main memory at all, but instead copies data directly to and from the GPU memory from some external source. One such case would be having encrypted data stored on a high speed storage solution which directly copies memory from disk to the GPU, either utilizing local storage in the

form of PCIe connected NVMe solid state storage disks, or over high speed PCIe network interconnects such as Infiniband. NVMe storage is somewhat uncharted territory for GPU DMA uses, but Infiniband is fairly commonly adopted among libraries and applications which seek to utilize GPUs in high speed networked applications. The output data from the encryption or decryption routine could then be copied directly to the same source, or a different high speed data storage or transmission endpoint on the GPU host, effectively using the GPU as a middle point between two points which store and require two different states of the data, either encrypted or decrypted.

Similar to implementations found on the CPU, it may also be more performant to implement AES entirely in assembly language. While the CUDA libraries and parts of the runtime may be closed source, the assembly language utilized by CUDA enabled GPUs, known as the Parallel Thread Execution (PTX) instruction set, may be an even more optimal way to implement AES as described in this paper. This was not explored at all in this research, but may prove to be an interesting task for a future researcher who is inclined to utilize PTX. Analysis of the PTX assembly output, available from the Nvidia CUDA C Compiler, could be analyzed to determine if there could be a performance benefit to writing the PTX manually.

Also shown was the incredibly consistent throughput of GPU kernels, with low to non-existent variance in runtime given a particular input size and algorithm. Low variance means that the throughput for a given data size can be predicted to within a small margin of error, and then planned for accordingly. Financial systems, networks and communications, and real-time operating systems all rely to some degree on such properties, and it would be a very interesting endeavor to explore applications for GPUs in general, not just AES, in these fields, and others.

Lastly, while not strictly a research problem, a robust system for error checking the status of kernel launches and memory copies would be a welcome addition to this application code base. For the purposes of research, a system with no error checking has been sufficient due to the completely deterministic and preplanned input sizes. However, in the real world this cannot always be assumed, and as such methods to ensure that the library alerts the developer when an error has occurred is an important addition. One such example is that the current implementation, when the kernel is run, makes heavy usage of GPU registers on the streaming multiprocessor. As a result of this, kernels which are launched with too many threads in a

block, even if they are within the CUDA runtime and hardware limits, can fail to execute. Issues such as that need a more robust way to be recovered from on the fly, and then logged for the developer or system administrator to address.

Bibliography

[1] *Announcing the advanced encryption standard (aes).* → pages vi, 10, 16

[2] *Recommendations for Block Cipher Modes of Operation*, National Institute of Standards and Technology, 800-38A (2001). → pages 11

[3] *Concurrency is not Parallelism.* `https://talks.golang.org/2012/waza.slide`, 2012. → pages 4

[4] *CUDA C Best Practices - Coalesced Access to Global Memory.* `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#coalesced-access-to-global-memory`, 2015. → pages vi, 5, 7

[5] *CUDA C Best Practices - Thread Hierarchy.* `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#thread-hierarchy`, 2015. → pages vi, 6

[6] *CUDA C Best Practices - Thread Hierarchy.* `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#asynchronous-transfers-and-overlapping-transfers-with-computation`, 2015. → pages vii, 23

[7] *PG-Strom.* `https://github.com/pg-strom/devel`, 2015. → pages 1

[8] A. D. Biagio, A. Barenghi, G. Agosta, and G. Pelosi, *Design of a Parallel AES for Graphics Hardware using the CUDA framework*, 2009 IEEE International Symposium on Parallel & Distributed Processing, (2009). → pages 1, 23

[9] J. Daemen and V. Rijmen, *The design of Rijndael: AES — the Advanced Encryption Standard*, Springer-Verlag, 2002. → pages 13

[10] S. Fazackerley, S. M. Mcavoy, and R. Lawrence, *GPU Accelerated AES-CBC for Database Applications*, Proceedings of the 27th Annual ACM Symposium on Applied Computing - SAC '12, (2012). → pages 1

[11] K. Iwai, T. Kurokawa, and N. Nisikawa, *AES encryption implementation on CUDA GPU and its analysis*, 2010 First International Conference on Networking and Computing, (2010). → pages 1, 18

[12] A. H. Khan, M. A. Al-Mouhamed, A. Almousa, A. Fatayar, A. R. Ibrahim, and A. J. Siddiqui, *AES-128 ECB Encryption on GPUs and Effects of Input Plaintext Patterns on Performance*, 15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), (2014). → pages 1, 23

[13] D. Kirk and W. mei Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann, San Francisco, CA, 2nd ed., 2013. → pages 3, 5

[14] D. Le, J. Chang, X. Gou, A. Zhang, and C. Lu, *Parallel AES algorithm for fast Data Encryption on GPU*, 2010 2nd International Conference on Computer Engineering and Technology, (2010). → pages 1

[15] C. Mei, H. Jiang, and J. Jenness, *CUDA-based AES Parallelization with Fine-Tuned GPU Memory Utilization*, 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), (2010). → pages 1

[16] M. Patchappen, Y. M. Yassin, and E. K. Karuppiah, *Batch Processing of Multi-Variant AES Cipher with GPU*, 2015 Second International Conference on Computing Technology and Information Management (ICCTIM), (2015). → pages 1, 18, 23

[17] N. Wilt, *The CUDA Handbook - A Comprehensive Guide to GPU Programming*, Addison-Wesley, 1st ed., 2013. → pages 8, 20

[18] W. M. N. Zola and L. C. E. D. Bona, *Parallel speculative encryption of multiple AES contexts on GPUs*, 2012 Innovative Parallel Computing (InPar), (2012). → pages 1