Advanced Encryption Standard

EE 5351 - Applied Parallel Programming

Final Project

Yebeltal Asseged – asseg003

Background

Advanced Encryption Standard (AES) , a variant of **Rijndaelck** block cipher**,** is a very common encryption algorithm developed by two Belgian Cryptographers Joan Daemen and Vincent Rijmen. It was originally designed to replace Data Encryption Standard (DES) that was designed in early 1970's by IBM.

Following DESCHALL Project that cracked a message encrypted with DES, in 1997 NIST announced their interest in finding a successor to DES. In addition to the perceived DES weakness and subsequent cracks, it has a relatively shorter 56-bit key that was becoming vulnerable to brute force attack. To mitigate the shorter key length, Triple DES was developed but found to be even slower. Triple DES didn't gain speed boost either in hardware and its large size made it unusable for limited resource systems.

The short comings of DES gave way to the adoption of a AES. In 1998, 15 candidates around the world submitted algorithms to NIST. Unlike DES, NIST was open for suggestions as to how the next algorithm should be chosen. This open concept was beneficial in the selection process since more eyes on the algorithms gave out unforeseen vulnerabilities; and subsequently only 5 were selected.

The 5 selected NIST standards were: **Twofish** and **Serpent** which were found to be highly secure but very complex and slower respectively; **RC6** was very simple but wasn't as secure as it was promised to be; IBM's **MARS** was fast but complex for implementation and **Rijndael** which had the best in class security, speed and simplicity even for smaller platform. In 2000, Rijndael was officially selected after subsequent rigorous tests, votes and debates by the open community. It was crowned AES and became effective since 2002 by NIST.

The need for a faster speed was among the many factors why **Rijndael** algorithm was eventually selected. Its parralizable algorithm makes it attractive for GPU implementation. This document details what makes AES algorithm and how it can be implemented on a GPU for even higher performance.

Introduction

AES is versatile in that it is able to accept 128, 192 and 256 bit keys.

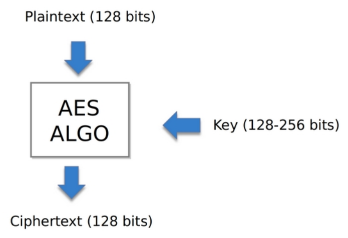


Figure 1. High level view of AES Encryption

As Figure 1 shows AES algorithm takes in a fixed 128 bit data and encrypts/decrypts it with the selected key size and outputs a 128 bit data. The key sizes translates to the number of rounds that needs to happen with in the AES block. A 128 bit key requires 10 rounds; 192 bit key requires 12 rounds and a 256 bit key requires 14 rounds as indicated in the figure 2.

One of the simplicity of AES is that one can easily swap out the number of rounds on Figure 2 if one wants to implement a 192 bit key instead. From a hardware perspective, this makes it easier to keep the same hardware and only increase the number of rounds with in the AES block.

Types of Block Cipher Modes

AES algorithm only handles 128 bit chunks. If one wants to encrypt more than 128 bit data, one will need to implement different cipher modes. These cipher modes vary in complexity and speed but they all do one job which is managing a large data set in to a smaller 128 bit chunks. Some of these modes include:

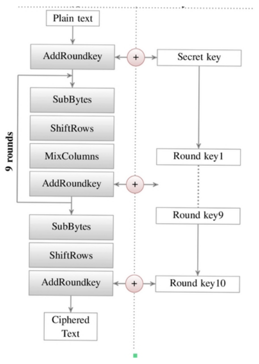


Figure 2. Ten rounds implemented on 128 bit key AES

Electronic Code Book (ECB), Cipher Block Chaining (CBC) and Counter Mode (CTR). ECB is the simplest to implement since it divides the large data set in to a padded 128 bit data chunks and encrypts/decrypts them accordingly.

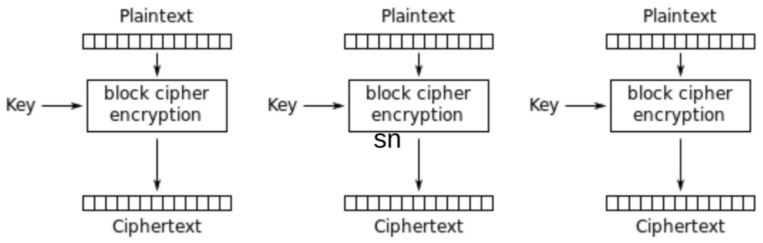


Figure 3. Electronic Code Book

Unlike ECB, CBC uses the previous encrypted/decrypted block to add confusion to the current block. It is a sequential operation which makes it slower but much more secure than ECB. CTR takes the the weakness of CBC which is speed and makes it faster by parallelizing each block just like ECB but with a benefit of higher security.

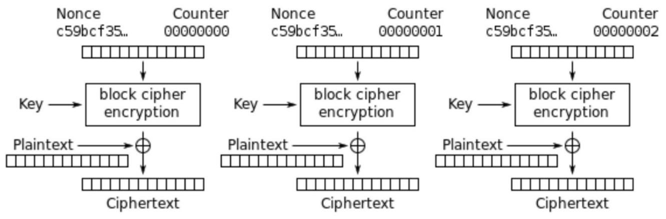


Figure 4. Counter Mode

Architecture and Implementation

This project focuses on accelerating AES algorithm on GPU and less on the method of block ciphers. As such, the ECB cipher was selected to show case advantages gained by running an AES algorithm on a GPU. Lets go over each parts that makes AES and how it was implemented on the GPU.

Key Expansion

Takes in a key and expands it for later use. For example for a 128 bit key, it takes in 16 byte and produces a 176 byte key that will be added to each round of the AES encryption/decryption as shown below. For reference, a 192 bit key requires 13 round keys, and a 256 requires 15 round keys.

Implementation of Key Expansion:

Its implementation on the GPU was to calculate the 176 bit expanded key on a single thread. Once this single thread calculates the 176 bits, it will share with threads within a the same block by storing it in a shared memory location. The calculation requires two 256 bit fixed constants SBOX and RCON. Each of these constants are loaded to shared memory when the kernel launches.

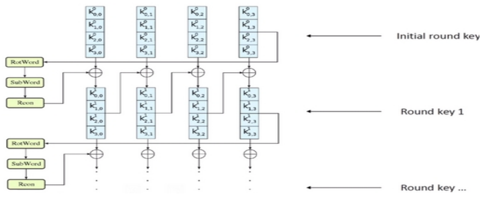


Figure 5. Key Expansion

Implementation of Key Expansion:

Its implementation on the GPU was to calculate the 176 bit expanded key on a single thread. Once this single thread calculates the 176 bits, it will share with threads within a the same block by storing it in a shared memory. The calculation requires two 256 bit fixed constants SBOX and RCON. Each of these constants are loaded to shared memory when the kernel launches. There are 256 threads in a block so each thread loads its respective data. Once all threads load there respective data, thread 0, will calculate the 176 bits and loads it back to the shared memory for all threads utilization in subsequent calculation.

Thread 0 sends the shared memory location of SBOX[256], RCON[256] and the 16 byte key to the private function **KeyExpansion**. The result will be saved to the shared memory address. This private function, performs a variety of calculation that involves rotation, substitution and rounding. A detailed comment is left inside the function.

Adding Round Key

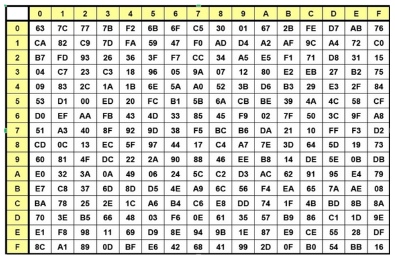
Adds each expanded key to the plain/cipher text weather one is doing encryption/decryption respectively.

Implementation of Adding round key:

Each thread with in a block is responsible for adding(XOR) a 16 byte plain/cipher data with its respective round key. This is implemented on **AddRoundKey** private kernel function.

Substitute Byte

Substitutes each byte with its respective non linear byte substitution table also known as S-BOX. This calculation is done Galois/finite field.



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Figure 6. S-BOX table

Implementation of substitute byte:

Each thread performs substitution for its 16 bytes it holds. Since all the SBOX is loaded in to the shared memory, it has a faster look up. This is implemented on **SubBytes** and **inv_SubBytes** private functions. Encryption and decryption use different tables hence why the need to create two private functions.

Shifting rows

With in a 16 byte, 4X4 matrix, shifting happens to add confusion in the algorithm. No shifting happens in the first row, a single rotation in the first row, 2 rotation in second row and 3 rotations in the last row. The same number of rotation but with opposite direction for decryption.

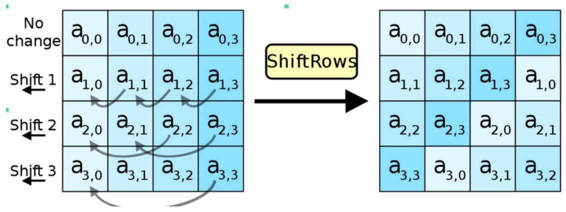


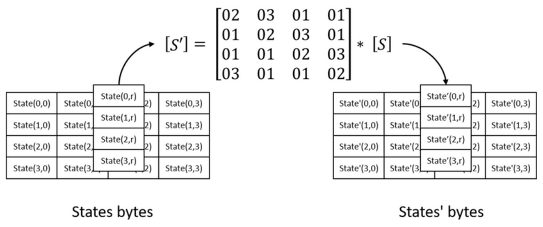
Figure 7. Shifting row for encryption

Implementation of Shifting bytes in a row:

Each thread performs shifting of 4X4 byte array. This is implemented on **ShiftRows** and **inv_ShiftRows** private functions. Encryption and decryption use shift operation in the opposite direction hence the need for two different functions.

Mixing Column

Mixing column is one of the primary ways dilution is achieved in AES. It is done by doing a dot matrix product of the 16 bytes with a set 4X4 matrix.



$[S'] = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} * [S]$
States bytes
States' bytes

Figure 8. Mixing of column

Implementation of Mixing Column:

Rather than doing a computation on a thread. A faster way was implemented. That would be precomputing all the possible dot product of the static vector with values ranging from 0 – 255 and creating a table for it. This table will then be saved in to shared memory location and all 16 bytes inputs to registers of a thread for a faster access. Then we do a simple XOR operation for each byte location. This way we utilize the precomputed values rather than computing the values for each byte on the go. This was implemented on MixColumns and inv_MixColumns private functions for encryption and decryption respectively. The down side of this method would be the initial loading of 256 precomputed bytes to shared memory. Since each thread loads its respective byte with in a block to the shared memory, and we also use registers to store the XOR result before we write it back to shared memory, the loss in transfer speed in the overall operation will be negligible.

Overall Design Architecture

In the included code, there are ten number of C files. It is attempted to create a hierarchical coding structure to easily follow the code. Files included are as follows:

main.cu : this file ties everything together. It initializes all the corresponding components and transfers information back and forth between object files.
aes_host_lib.cu/h : this file is a library for host operation. That is reading and writing to file. When reading the file it makes sure that the file is a multiple of 256 threads * 16 bytes. This will make sure that each of the threads have data to work with. It is padded with 0x00. During write operation, the padded data will be removed. It comes with its own status type definition **aes_host_lib_status_t.**
aes_device_lib.cu/h : this file is a library for device operation. That is all the operation that will be performed by the kernel with its kernel support functions. It also includes private device kernel functions. It comes with its own status type definition **aes_device_lib_status_t**.
aes_ui.cu/h : this file is intended to handle user communication. It accepts user input and structures required information. Once all the information is filled out, it passes the information for processing. It comes with its own status type definition **aes_ui_status_t**. For options available, a user can input **./aes_d -help** for more information. For a simple encryption test, **./aes_d -test -e** and decryption test, **./aes_d -test -d** can be entered**.**
aes_engine.cu/h: this file uses the device and host library to perform AES operation. Once the user inputs all the required information, **aes_engine** initializes the kernel and sets off operation. It also utilizes data streaming to help improve the performance by coping data while part of the data is being processed by the kernel. It comes with its own status type definition **aes_engine_status_t**.
aes_config.h: this is where further modification can be made based on the system its running on.

Performance Metrics

A significant amount of speed up was achieved especially for larger data sets. OpenSSL was used as a CPU based operation. OpenSSL is an open source cryptography library that offers multitude of applications in Transport Layer Security (TLS) protocols. Most linux machines have OpebSSL preinstalled in them but if you dont happen to have one the C source code is available on their github page – https://github.com/openssl/openssl . It can be cloned, and built if one doesn't have it preinstalled. Once you have it on your system. Its pretty much easy to do different crystallographic functions.

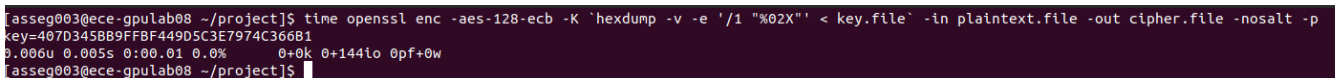
CPU

plaintext.file size = 69.6Kb

Command used:

```
openssl enc -aes-128-ecb -K `hexdump -v -e '/1 "%02X"' < key.file` -in plaintext.file -out cipher.file -nosalt -p
```

To measure speed **time** bash time command was used.

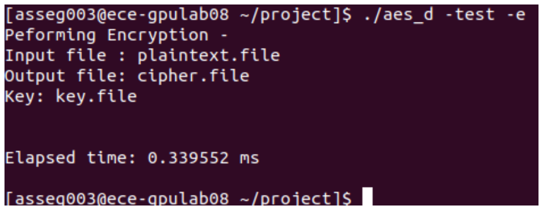


```
[asseg003@ece-gpulab08 ~/project]$ time openssl enc -aes-128-ecb -K `hexdump -v -e '/1 "%02X"' < key.file` -in plaintext.file -out cipher.file -nosalt -p
key=407D345BB9FFBF449D5C3E7974C366B1
0.006u 0.005s 0:00.01 0.0%      0+0k 0+144io 0pf+0w
[asseg003@ece-gpulab08 ~/project]$
```

Figure 9. CPU processing

This indicates that a total time of user+sys = 6ms + 5ms = 11 ms took to process. But this also includes writing and reading from the file system.

GPU



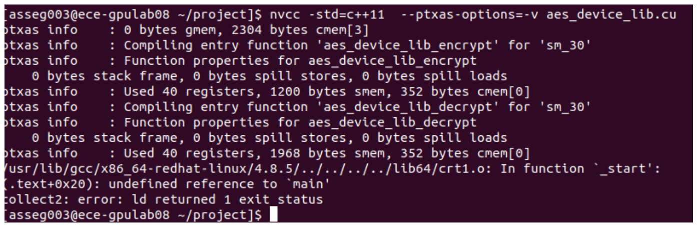
```
[asseg003@ece-gpulab08 ~/project]$ ./aes_d -test -e
Peforming Encryption -
Input file : plaintext.file
Output file: cipher.file
Key: key.file


Elapsed time: 0.339552 ms

[asseg003@ece-gpulab08 ~/project]$
```

Figure 10. GPU processing

it took a total of 0.371 ms to finish processing.
Running **nvcc -std=c++11 --ptxas-options=-v aes_device_lib.cu** gives us memory requirment for the kernel functions.



```
[asseg003@ece-gpulab08 ~/project]$ nvcc -std=c++11  --ptxas-options=-v aes_device_lib.cu
ptxas info    : 0 bytes gmem, 2304 bytes cmem[3]
ptxas info    : Compiling entry function 'aes_device_lib_encrypt' for 'sm_30'
ptxas info    : Function properties for aes_device_lib_encrypt
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 40 registers, 1200 bytes smem, 352 bytes cmem[0]
ptxas info    : Compiling entry function 'aes_device_lib_decrypt' for 'sm_30'
ptxas info    : Function properties for aes_device_lib_decrypt
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 40 registers, 1968 bytes smem, 352 bytes cmem[0]
/usr/lib/gcc/x86_64-redhat-linux/4.8.5/../../../../lib64/crt1.o: In function `_start':
(.text+0x20): undefined reference to `main'
collect2: error: ld returned 1 exit status
[asseg003@ece-gpulab08 ~/project]$
```

Figure 11. Encryption / decryption memory requirement

This means that
for GTX 1080

For encryption we are using 40 registers and 1200 bytes
For decryption we are using 40 registers and 1968 bytes

Maximum blocks only with thread:
2048 thread/SM * 1/(256) block/thread
2048/256 = 8 blocks / SM

Maximum blocks only with shared memory:
(total shared memory / block) / (kernel shared memory requirement per block)
49152 byte / block * 1/1200 block / byte
49152/1200 = 40 blocks / SM for encryption

49152 byte / block * 1/1200 block / byte
49152/1968 = 24 blocks / SM for encryption for decryption

Maximum blocks only with available registers:
65536 reg/block / (40 reg / thread * 256thread/block) =
65536/(40 * 256) = 6.4 ~= 6 blocks / SM for both encryption and decryption.
Which means we have 6*256*20 = 30720 threads x 16 byte/ thread = we can process 491.520 Kbyte simultaneously.

Citation

1. Explore scientific, technical, and medical research on ScienceDirect. ScienceDirect.com | Science, health and medical journals, full text articles and books. (n.d.). Retrieved May 9, 2022, from http://www.sciencedirect.com/

2. *AES (step-by-step)*. CrypTool Portal. (n.d.). Retrieved May 9, 2022, from https://www.cryptool.org/en/cto/aes-step-by-step

3. *OpenSSL*. GitHub. (n.d.). Retrieved May 9, 2022, from https://github.com/openssl/