Fall 2022 ME/CS/ECE759 Final Project Report
University of Wisconsin-Madison

AES Acceleration with GPU

Rakshith Macha Billava

December 14, 2022

Abstract

The purpose of this project is to parallelize the AES symmetric key encryption algorithm. AES is a block-wise algorithm that provides some scope to enable parallelism and improve performance. For the purpose of parallelizing the AES algorithm, CUDA is used. Each thread handles one element of the plain text and 512 threads are allocated in each block. The performance of the parallel and the naïve implementation was analyzed. Using this implementation, a performance gain of up to 40X was achieved.

Link to Final Project `git` repo: *https://git.doit.wisc.edu/MACHABILLAVA/finalproject759/-/tree/main*

Contents

1. General information

1. Your home department: Electrical and Computer Engineering Department
2. Current status: MS student
3. I release the ME759 Final Project code as open source and under a BSD3 license for unfettered use of it by any interested party.

2. Problem statement

AES is a symmetric cryptographic encryption algorithm in which the plain text – the data that needs to be encrypted, is encrypted and decrypted using a symmetric key – both receiver and the sender use the same key. AES is the most preferred encryption algorithm and several studies[1] show that AES outperforms in terms of computational time and memory requirement when compared to other encryption algorithms. In fact, AES is used in most online websites and transactions. Generally, in cryptography, 10% of the time is spent in the sender sharing the symmetric key with the receiver using asymmetric cryptographic algorithms, and 90% of the time is spent in encrypting the actual data using the symmetric key previously shared and the symmetric cryptographic algorithm. Reducing the time taken by AES encryption even by a small factor will greatly improve the performance of all the tools that use AES due to the sheer magnitude of the use of AES-encrypted data transfer.

Secondly, AES is a block-wise algorithm and has multiple rounds. Each round's output block depends on the previous round's output block. Therefore, parallelizing AES does pose an interesting challenge in how we can parallelize the algorithm to obtain better results.

The end result would be a performance analysis of naïve AES implementation vs parallel AES implementation.

3. Solution description

3.1 AES Algorithm

The AES algorithm can be broadly divided into two categories:

- Key Expansion
- AES Encryption

AES symmetric key is 128 bits (16 bytes) for AES-128 and 256 bits (32 bytes) for AES-256. In the key expansion step, this symmetric key is expanded to obtain round keys – 176 bytes for AES-128 and 240 bytes for AES-256. This round key will then be used in each round of the AES algorithm. For the key expansion step, the following procedure is followed –

- The first block (16 bytes) of the round key is the key itself.
- For the next block the last row of the key is shifted by 1
- Each element of the last row is substituted with the corresponding element from the S-Box matrix[2]
- To the substituted row, a round constant is added[3]
- The resultant row is then XORed with the first row of the previous round key block to obtain the first row of the current block.

- To obtain the subsequent rows, the previous row of the current block is XORed with the current row of the previous block.
- This entire procedure is repeated 10 times in AES-128 and 14 times in AES-256 to obtain the entire round key.

AES encryption is a block-wise algorithm wherein it handles one block at a time. The block that is currently being operated on is called the state matrix. AES encryption can be further divided into the following steps[4] –

- Add Round Keys – Each element of the round key is added to each element of the state matrix
- Substitute Bytes – Each element of the state matrix is substituted with the corresponding element from the S-Box matrix
- Shift Rows – The elements in the n^{th} row of the state matrix are shifted by n times
- Mix Columns – The resultant matrix of the previous step undergoes a Galois Multiplications[5][6][7]

Before the computation of the round key is started the Add Round Key step is introduced. The subsequent rounds except the last round follow this sequence – Substitute Bytes, Shift Rows, Mix Columns, and Add Round Keys. In the last round, the Mix Columns step is omitted.

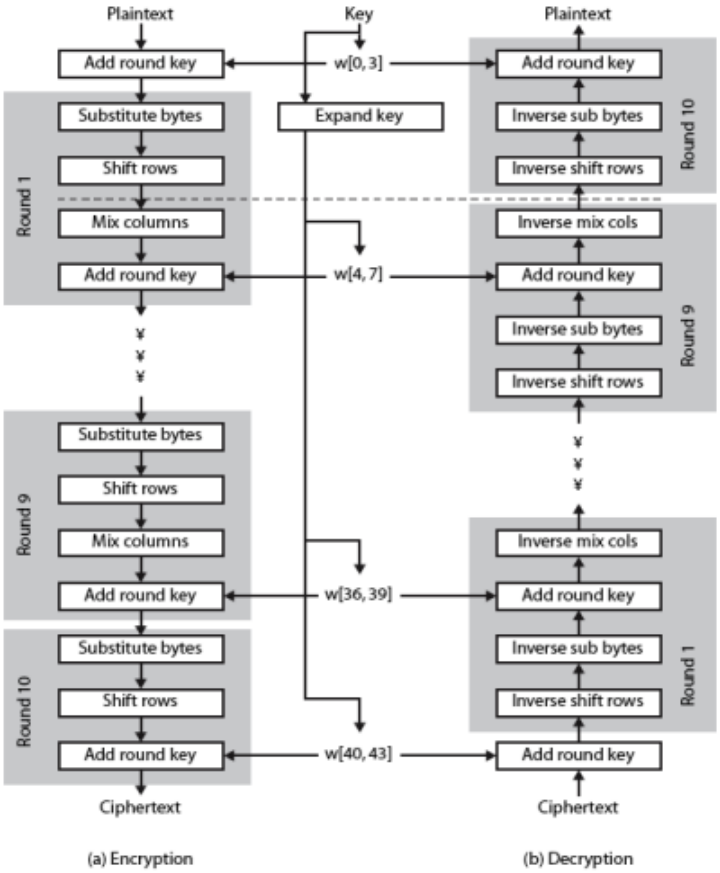


Figure 1 - AES algorithm[8]

For a detailed understanding of AES, CrypTool 2[9] which enables AES visualization can be used. For the convenience of the reader, the visualization of the AES key expansion and AES encryption is recorded

and can be found here. The AES algorithm was first implemented using just the CPU without any parallelization and then tested that the algorithm works. A context structure `aes_struct` was used to maintain pointers to all the buffers like plain text, key, and other configurations like AES mode. All matrices are stored using 1D arrays in column-major representation.

```
typedef struct aes_struct
{
    uint8_t aes_mode;                   // AES_ECB or AES_CTR
    uint8_t aes_key_length;             // In bits - 128 or 256
    const uint8_t* key;                 // Buffer to store AES key
    uint8_t* round_key;                 // Buffer to store round key
    uint8_t round_key_length;           // In bytes
    uint8_t* counter;                  // Buffer to store IV in CTR mode
    uint8_t* plain_text;                // Buffer to store plain text
    int plain_text_length;              // In bytes
    uint8_t* cipher_text;               // Buffer to store cipher text}
aes_struct;
```

A function call was implemented for each of the steps mentioned above and these were called for each round for each of the blocks in the plain text. The key expansion was implemented in a separate source file. The naïve implementation can be found here.

3.2 Key Expansion using OpenMP

In the key expansion, the output of each row depends on the result of the previous row. Similarly, the output of each round depends on the result of the previous round which makes things hard to parallelize. However, if we consider column-wise, the next element is dependent on the previous element of the same column but there is no dependency on any other elements from any other column. This gives way to parallelizing the column-wise generation of elements, but it involves some additional branching. As only 4 columns exist and it involves conditional statements OpenMP is used to parallelize key expansion. Each column is considered a section and is allocated to one core.

3.3 AES Encryption using CUDA

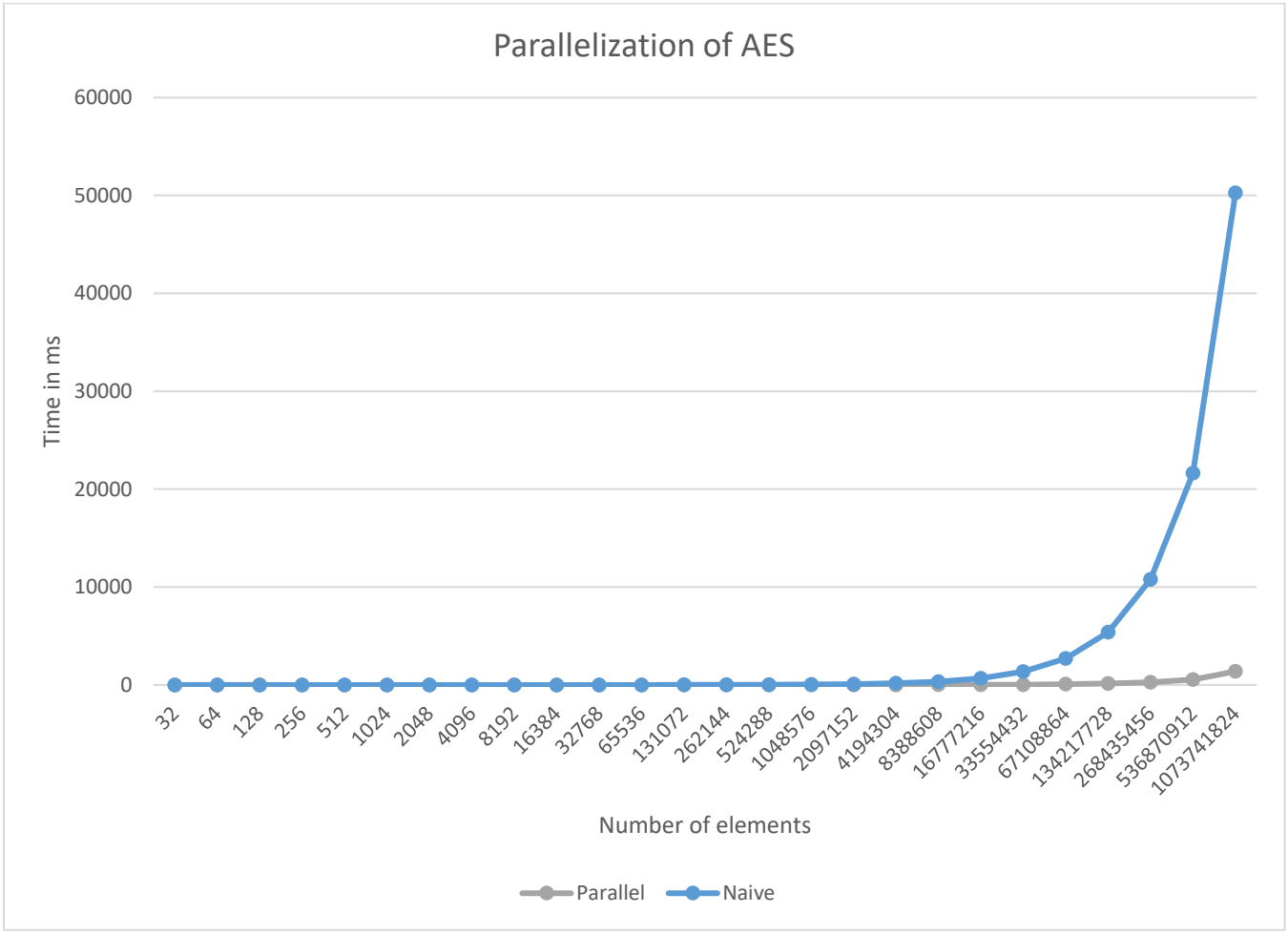
AES deals with 16-element blocks with comparatively less dependency on the outcome of the other elements and no dependency on the outcome of other blocks. So, the idea is to use 1 thread per element, in other words, all threads in a warp will handle 2 blocks (1 element per thread). The S-Box values, round keys, plain text, and the final cipher text are stored in the shared memory. The shift array constants (by how much should a particular element be shifted) and the Galois matrix elements are precalculated to avoid computations and are also stored in the shared memory. Each thread copies 2 to 4 elements from one of these buffers into the shared memory based on the `threadIdx.x` value. The function calls are removed or made inline, and the shifting of the elements is performed by modifying the index of the element. The Mix Columns step requires the elements of the entire row for the Galois multiplication which necessitates all the threads in a warp to be synchronized. The cipher text is then copied back to the device array. The number of threads per block is variable. However, all the constants like S-Box and round constants are needed by each block. Considering the amount of data that needs to be copied for each block it makes more sense to have bigger blocks. When tested I got the best performance with 512 threads per

block. The final implementation is tailored for 512 threads. Other optimizations like using a unified memory, and combined device array did not result in significant performance gains.

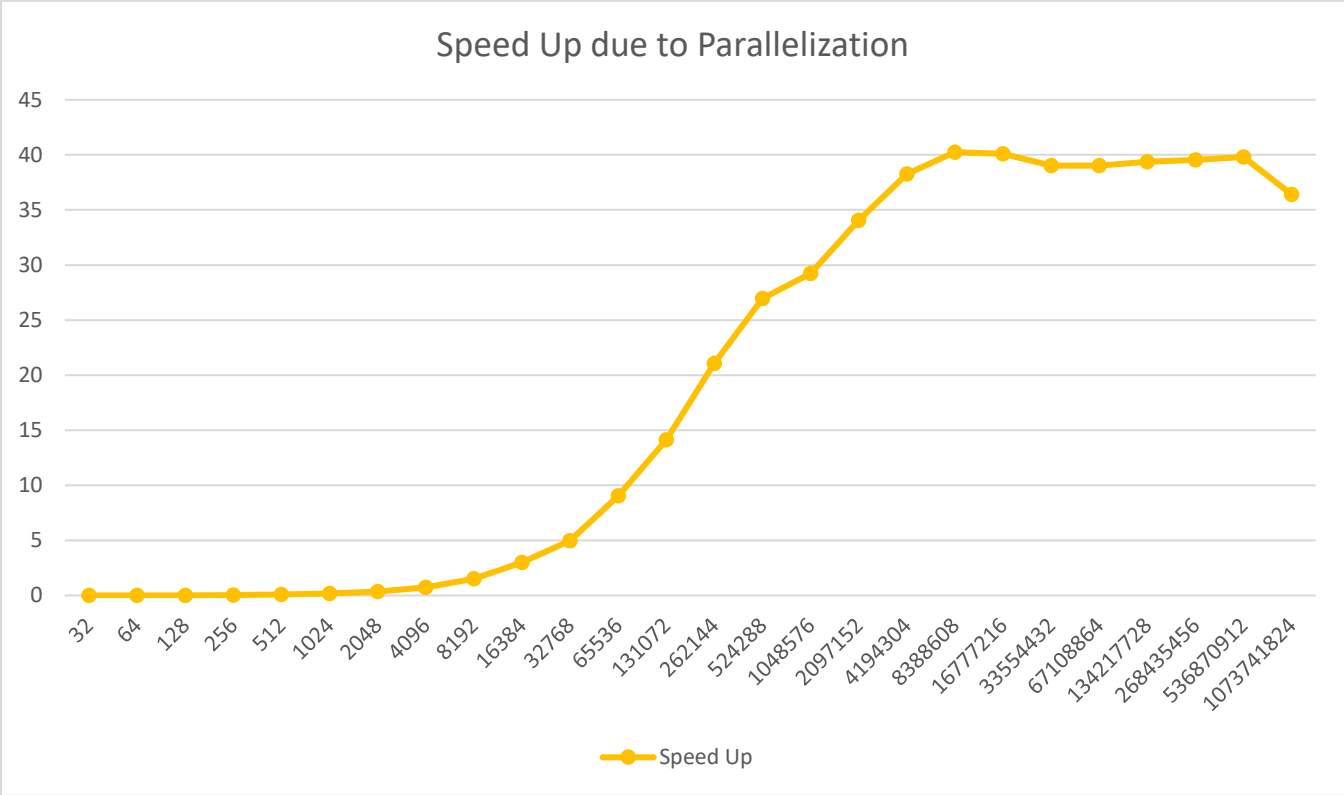
4. Overview of results. Demonstration of your project

This section includes the tests performed for the parallel AES implementation. However, similar tests have been performed for the naïve implementation too.

- Testing parallel AES with the default plain text and keys –
  Ensure `USE_DEFAULT_INPUTS`, `DISPLAY_INPUTS`, and `COPYABLE_FORMAT` (main.cuh:41) are set to 1. The default inputs are present in main.cu:30. The output is compared with any of the online AES calculators like [OnlineDomainTools] to verify that the AES algorithm works as expected.
- Testing parallel AES with random values of length 1024 –
  Ensure `USE_DEFAULT_INPUTS` is set to 0 and `COPYABLE_FORMAT` and `DISPLAY_INPUTS` (main.cuh:41) are set to 1. The output is compared with the result obtained from the online AES calculator. This is to test that multiple blocks are handling the data as expected.
- Timing analysis of the implementation is performed for the length of plain text ranging from 2^5 (32) to 2^{30} (1073741824) –



- Speedup (Time taken by naïve implementation / Time taken by the parallel implementation) is also plotted for a comparative study. The maximum speed up obtained is 40.2.



- Timing analysis of the Key Expansion step between the OpenMP and the naïve implementation was performed. Key length cannot be scaled. The time taken by the naïve implementation was 0.0006 ms whereas the OpenMP implementation takes about 0.257 ms. OpenMP performs worse compared to the naïve implementation. This is expected as the size of the key is 16 and only 176 elements of the round key need to be calculated. Parallelism overhead is too high when compared to the number of elements for computation. Also, a large amount of time is spent moving the data which results in diminishing results. This implementation was done just to demonstrate that even a seemingly non-parallelizable code can be parallelized with an unorthodox methodology. In case the number of elements is scaled up OpenMP implementation should provide faster results.
- The parallel implementation is also tested with the default and 1024 elements random inputs for the AES Counter mode[10].

5. Deliverables:

- The GitLab repository contains 2 folders for naïve and parallel implementations respectively. Both folders contain the Slurm script for executing the corresponding code. Running the Slurm command by using `sbatch taskrun.sh` should run the AES code with default inputs. The comments in the script provide directions to run timing analysis or change modes.

- The code structure for both implementations is almost the same. The main.cu/.cpp file is the entry point of the code. The default inputs also can be found in the same file. The `main()` function invokes the Key Expansion and AES encryption procedures. The main.cuh/.h file contains all the constants that need to be reviewed by the user. The README.md file contains further information about these configurations.
- The key_helper.cu/.cpp file contains all the helper functions and constants used by the Key Expansion step. Similarly, all the AES encryption-related functions and constants are in the aes_parallel.cu/.cpp and aes_parallel.cuh/.h files.

6. Conclusions and Future Work

The following enhancements can be considered as future work. Although some of these objectives are tangential to the main objective – parallelizing the AES algorithm, it is required for the project to be a complete solution.

- In this project, I have tried to obtain the maximum performance the best I can without taking any reference. The next step would be to look at research papers and try to understand if performance can be further increased.
- AES-256 mode is currently not supported. 32-byte blocks need to be supported and the same analysis needs to be performed on AES-256 mode.
- At the moment the number of threads is fixed to 512 as 512 threads gave the best performance. This needs to be made configurable so as to accept the number of threads from the user.
- The AES mode needs to be accepted as user input instead of changing the define.
- Counter mode of AES is supported only in the parallel implementation. The same needs to be added to naïve implementation.

The first 3 points do require considerable efforts to analyze and modify the current implementation. The last 2 enhancements are more of an aesthetic change.

The main learning is that even if a particular algorithm is seemingly non-parallelizable (in this case the current output depends on the previous output) the code can be parallelized by looking at it from a different perspective. The project also enabled me to think about the hardware first approach of writing code instead of thinking about convenience or making the code presentable. The project also provided a platform for me to practically reason which tool should be picked for parallelizing the algorithm and more importantly why. Although we already learned about these during our assignments, the project also taught me the importance of shared memory, cache locality, memory issues, etc.

References

[1] Babitha M.P. and K. R. R. Babu, "Secure cloud storage using AES encryption," 2016 International Conference on Automatic Control and Dynamic Optimization Techniques (ICACDOT), 2016, pp. 859-864, doi: 10.1109/ICACDOT.2016.7877709.
[2] S-Box Wiki
[3] AES key schedule - Wiki
[4] AES - Wiki
[5] Understanding AES Mix-Columns Transformation Calculation
[6] Rijndael MixColumns
[7] How to solve MixColumns - StackExchange
[8] Abdullah, Ako. (2017). Advanced Encryption Standard (AES) Algorithm to Encrypt and Decrypt Data.
[9] CrypTool 2
[10] Block cipher mode of operation - Wiki
[11] What is the canonical way to check for errors using the CUDA runtime API? – StackOverflow
[12] Stackoverflow and Nvidia Development forums
[13] AES – The Advanced Encryption Standard Explained (YouTube video)
[14] AES Explained (Advanced Encryption Standard) - Computerphile (YouTube video)
[15] Cypress Semiconductors README format