AES Encryption/Decryption using CUDA

Geethakrishna Puligundla

Overview:

AES stands for Advanced Encryption Standard is a widely used symmetric encryption algorithm that provides secure data encryption. It is a variant of Rijndael block cipher algorithm.

It is developed as a successor to the Data Encryption Standard, AES was established as a federal standard by the National Institute of Standards and Technology (NIST) in 2001. Its design forces both efficiency and security, utilizing a block cipher approach to encrypt and decrypt data in blocks of 128 bits, with key lengths of 128, 192, or 256 bits. It is used to protect sensitive information and securing the internet communication. There are lot of modifications made to the original implementation which made it very strong and powerful.

However, traditional CPU-based implementations can be slow when processing large amounts of data. Even though there are very powerful CPU's available today they can't perform it parallel but if we leverage GPU's it is a massive improvement in performance and time. This project implements a simple CUDA-accelerated AES-128 encryption system that leverages GPU parallelism to achieve higher throughput.

Design and Implementation:

Let's see the AES algorithm, how actually it works and what are the internal steps needed to complete the encryption/decryption process. Later I'll explain how I used GPU to implement the parallelization model of it.

AES Algorithm mainly consist of 4 methods which are iteratively performed on each block to generate the cipher. Here number of iteration is depend on the symmetric key size.

Key size	Number of Rounds
128	10
192	12
256	14

In each round as I mentioned above it contains 4 methods. They are

1. SubBytes
2. ShiftRows
3. MixColumns (except final round)
4. AddRoundKey

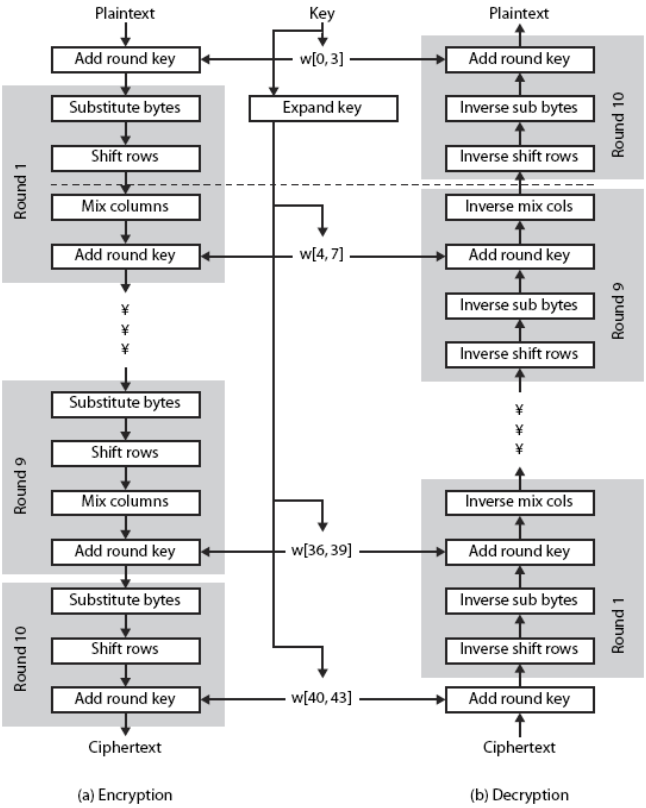

Fig: AES Architecture

The AES algorithm takes the input and converts into 128 bit block sizes and perform the above operations on each block. The each block is called state and it is transformed into a 4x4 matrix.

Key expansion process is where, we expand the key of size 128bit to be sufficient for all the rounds. The process uses the Rijndael key schedule, which includes the use of the RCON constant for key transformation. In my implementation original 128-bit key is expanded into 11 round keys (176 bytes).

Now let's see what each function does inside each round for encryption, however there is a change in last round (doesn't contains mixColumns).

1. **SubBytes:**

In this function, we substitute each byte in the state matrix with a precomputed S-Box key. It helps for adding confusion.

2. **ShiftRows:**

In this function, we transform the state matrix by different offsets for each row. It cyclically shifts rows left as below

- Row 0: no shift
- Row 1: shift 1 position
- Row 2: shift 2 positions
- Row 3: shift 3 positions

3. MixColumns:

In Mixcolums, we perform the matrix multiplication to add diffusion to the state matrix. We perform column-wise matrix multiplication using Galois Field arithmetic. However in the last round we wont perform this step.

4. AddRoundKey:

This is the final step in each round where the state matrix is transformed by performing the bitwise XOR operation with the round key. This step ensures that each round key is incorporated into the encryption process.

It follows the similar steps as show in the image for the decryption, where it does inverse operations.

Now let's see how I leverage the GPU's and implemented it. I primarily used the streams and different memory types to optimize the performance.

Instead of encrypting data one block at a time like a CPU would do, I split the work across thousands of tiny workers (GPU threads) that work simultaneously. Each worker takes a 16-byte chunk of data and encrypts it independently.

To further improve the performance, I leveraged CUDA's multi-stream capabilities by implementing four concurrent streams, which allowed me to overlap data transfers with computation, effectively hiding memory latency.

For optimal memory access patterns, I placed the frequently accessed S-box lookup tables and round keys in shared memory, while utilizing constant memory for static AES constants. This memory hierarchy optimization significantly reduced global memory access latency.

I chose to process data in 8MB chunks and implemented pinned host memory to enable efficient Direct Memory Access (DMA) transfers between the CPU and GPU.

Results:

I ran the encryption and decryption on a 1GB file and below are the results

```
  File: random_text_file.txt
  Size: 1073741824      Blocks: 2097152    IO Block: 1048576 regular file
Device: 47h/71d Inode: 12218139610859760426  Links: 1
Access: (0644/-rw-r--r--)  Uid: (488897/ gpuligu)   Gid: (10000/  cuuser)
Access: 2024-12-08 18:52:02.919293514 -0500
Modify: 2024-12-08 18:52:02.919293514 -0500
Change: 2024-12-08 18:52:02.919293514 -0500
 Birth: -
[gpuligu@node0035 AES_project]$
```

Encryption:

```
[gpuligu@node0035 AES_project]$ ./aes_cuda -e random_text_file.txt out.enc
Reading key from 'key.bin'...
Reading input file: random_text_file.txt
Read 1073741824 bytes
Padding the input file(PKCS7). Padded size: 1073741824 bytes
Encrypting the data using CUDA...
Kernel Time to complete the Encryption is: 334.833 ms
Writing encrypted data to: out.enc
[gpuligu@node0035 AES_project]$
```

```
[gpuligu@node0035 AES_project]$ head -n 5 out.enc
```

Decryption:

```
[gpuligu@node0035 AES_project]$ ./aes_cuda -d out.enc plain.txt
Reading key from 'key.bin'...
Reading input file: out.enc
Read 1073741840 bytes
Decrypting the data using CUDA...
Kernel Time to complete the Decryption is: 676.435 ms
Writing decrypted data to: plain.txt
[gpuligu@node0035 AES_project]$
```

```
[gpuligu@node0035 AES_project]$ head -n 5 plain.txt
This is a plain text file used for AES
This is a plain text file used for AES
This is a plain text file used for AES
This is a plain text file used for AES
This is a plain text file used for AES
[gpuligu@node0035 AES_project]$
```

Conclusion:

In conclusion, it shows the significant potential of GPU parallelization for cryptographic operations. By using multi-stream processing, efficient memory management with pinned memory, and dynamic thread configuration based on GPU capabilities. While this implementation serves as an educational prototype, there are several areas for potential improvement, including further optimization techniques such as memory transfer overhead, reducing the number of kernel calls and support for additional AES variants.

References:

- https://en.wikipedia.org/wiki/Advanced_Encryption_Standard
- https://www.youtube.com/watch?v=O4xNJsjtN6E
- https://www.youtube.com/playlist?list=PLBlnK6fEyqRgj06MEnp2VebJ8DgMpF0Zj
- https://github.com/kokke/tiny-AES-c
- https://www.youtube.com/playlist?list=PLKK11LigqitiRH57AbtyJyzsfbNfA8nb-