AES Implementation on GPU

Himanshu Jain
Department of Electrical and Computer Engineering
University of Arizona

Ran Zhao
Department of Electrical and Computer Engineering
University of Arizona

{himanshujain, rzhao}@email.arizona.edu

***Abstract* - With the advent of Cryptoanalysis, more and more applications are starting using Advance Encryption standard (AES) to protect the information security. However, the encryption activity suffers from its computational intensiveness that makes it to use large computational resources with high performance issues. In this paper, we propose an effective implementation of AES-ECB symmetric cryptographic primitive using the CUDA framework. We obtained a speedup of up to 4x against an advanced CPU (Intel i7x84) based implementation. Moreover, our experiment also provides insights on the optimization techniques to efficiently exploit the hardware resources by exposing the different kinds of parallelism. Additionally, implementation with overlapping between processing and data transfer yielded 2.75 Gbps throughput include the data transfer time.**

Index Terms* - *AES, ECB, CUDA, GPU, Optimization, speedup, streaming.

I. INTRODUCTION

Today's web communications and cloud computing requires more secure data communications than ever before and as a result, block ciphers cryptography becomes more important day by day. The encryption and decryption of a data is a computationally intensive task and takes a heavy toll on the computational resources. Therefore, it is sensible to build dedicated Hardware and software accelerators.[1]
Meanwhile, the aggressive competition for the Graphic Processing Unit (GPU) market is driving these architectures towards increasing level of hardware parallelism, while containing the cost. To enable GPU as an efficient cryptographic acceleration unit, CUDA statements need to be inserted into AES programs. Although AES contains rich data parallelism, optimized code exhibits very light calculation workload with high throughput.
In the remainder of this paper, we address the issue of deploying encryption primitives on the GPU hardware by providing a parallel AES design targeting the NVIDIA GeForce 6.1 GPU family. We explore several parameters choices realizing different implementations and offering different insights on the memory optimization technique that needs to be taken into account when implementing cryptographic algorithms on graphic oriented hardware. An extensive experimental campaign supports our implementation choices.

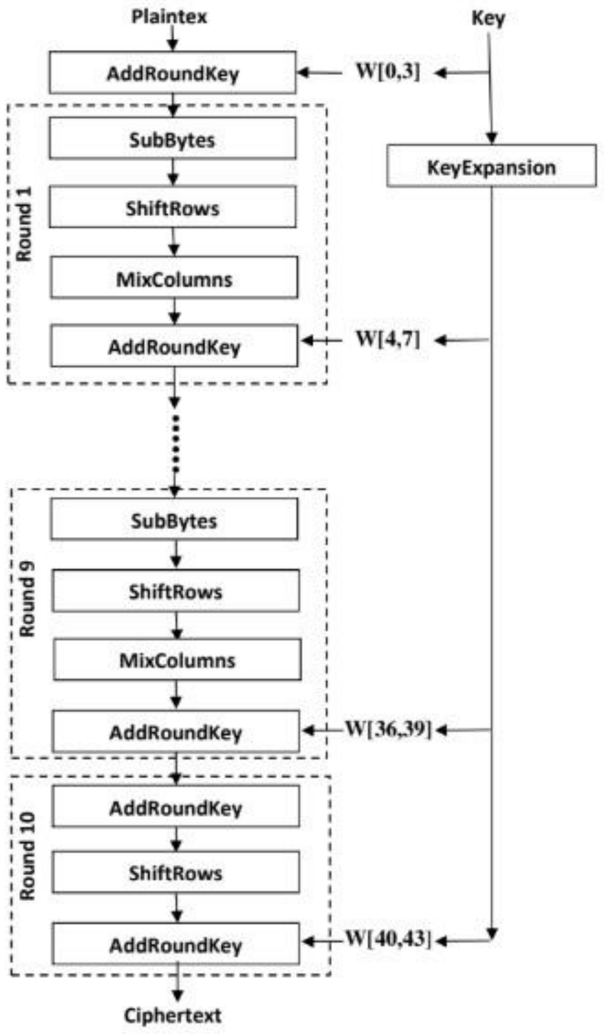

Fig 1. AES encryption

The rest of the paper is organized as follows. Section II presents a detailed description of the implemented AES algorithm. Problem Background and Related work is introduced in section III. Section IV explains our parallel design strategy for AES. Section V explains optimization strategy. The experiment result on different platform are presented and analyzed in section VI. Finally, Section VII concludes this work and discusses future research directions and possible future work followed afterwards.

II. Algorithm Overview

Advanced Encryption Standard (AES) is a symmetric cryptographic algorithm announced by National Institute of Standards and Technology (NIST) to replace the Data Encryption Standard (DES). The standard adopted three blocks ciphers, AES-128, AES-192 and AES-256, from a large collection originally published as Rijndael due to its ease of implementation on a wide range of 8-bit to 32-bit processing platforms as well as its high performance. On 32-bit processors, AES can be implemented by a sequence of bit-wise XOR operations, table lookups and 8-bit shifts.

This paper is focused on AES-128 whose overall structure is shown in Fig. 1. The input to the encryption algorithm is a 16-byte data block, which is depicted as a square matrix. This block is copied into a state matrix, which is modified at each stage of encryption. The first 9 rounds are composed by the same steps, SubBytes, ShiftRows, MixColumns and AddRoundkey. There is no MixColumns at the last round.

SubBytes	$b_{i,j} = S[a_{i,j}]$
ShiftRows	$\begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix} = \begin{bmatrix} b_{0,j} \\ b_{1,j-1} \\ b_{2,j-2} \\ b_{3,j-3} \end{bmatrix}$
MixColumns	$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix}$
AddRondKeys	$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = \begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}$

Table I Algebric form of four steps in each round

KeyExpansion in Fig. 2 takes an input of 16-byte key and generates a linear array of 176-byte RoundKeys. They provide distinct 16-byte RoundKey to each of AddRoundKey process, which is denoted by w in Fig. 1 & 2. The different steps of the round transformation can be combined into a single set of tables lookups, allowing for very fast implementation on processors having word length of 32 bits.

$$\begin{bmatrix} k_0 & k_4 & k_8 & k_{12} \\ k_1 & k_5 & k_9 & k_{13} \\ k_2 & k_6 & k_{10} & k_{14} \\ k_3 & k_7 & k_{11} & k_{15} \end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix} w_0 & w_1 & w_2 & w_3 \end{bmatrix}$$

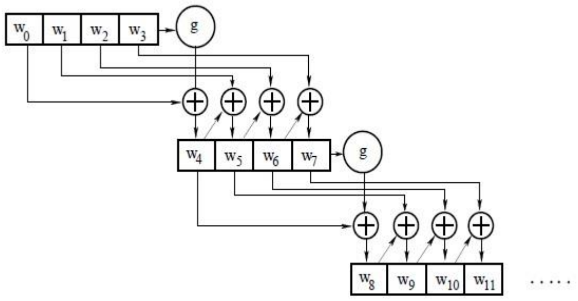

Fig 2. Key expansion form 16-byte key matrix

The four transformations of a round can be expressed in algebraic form as shown in Table I, where aij the generic element of the state matrix and kij is the element of RoundKey matrix. In the ShiftRows equation, the column indices are taken mod 4. There four expressions can be combined into one 2 single equation:

$$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} S[a_{0,j}] \\ S[a_{1,j-1}] \\ S[a_{2,j-2}] \\ S[a_{3,j-3}] \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}$$
$$= \left(\begin{bmatrix} 2 \\ 1 \\ 1 \\ 3 \end{bmatrix} \bullet S[a_{0,j}] \right) \oplus \left(\begin{bmatrix} 3 \\ 2 \\ 1 \\ 1 \end{bmatrix} \bullet S[a_{1,j-1}] \right)$$
$$\oplus \left(\begin{bmatrix} 1 \\ 3 \\ 2 \\ 1 \end{bmatrix} \bullet S[a_{2,j-2}] \right) \oplus \left(\begin{bmatrix} 1 \\ 1 \\ 3 \\ 2 \end{bmatrix} \bullet S[a_{3,j-3}] \right) \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}$$

Equation 1 Combined equation into single expression

where S is the S-box table.

III. Problem Background

Unfortunately, the encryption and decryption processes of AES takes a considerable amount of time for large data size. The loop level dependency and choice of memory storage deeply effected the performance of algorithm at architecture level. Some Researchers have worked on different implementation techniques of AES on GPU are as follows:

Andrea [1] et.al. effectively implement the AES block cipher, exploited parallelism as fine grained and coarse-grained approaches. In fine grained approach, 32 -bit words of the state computed by each AES round is manipulated by operating system independently one from each other. In coarse grained approach, different blocks are encrypted by different instances of the algorithm. They report to achieve throughput improvements of up to 14 times over the CPU (intel core duo) as a baseline.

Chonglei & Hai [2] proposed the approach to arrange data in different GPU memory spaces properly overcoming the extra communication delay. Their fine-tuned memory utilization approach achieves the upper bandwidth for GeForce 9200 which is 6.4GB/s.

Nishikawa [3] et. al. experiments are based on the thread level granularity and the memory allocation styles of variables of AES to achieve 35GBps throughput on GeForce GTX285. They consider them as important factor for effective processing and exhibited 2-30% effectivity on the performance.

Ahmed et.al.[4] presented eight parallel AES schemes, mainly divided on the shared memory approach and wrap shuffle approach, focusing on three aspects of optimization: key memory location, computational granularity and thread block size seeking the best throughput. The work has achieved the highest performance up to 279.86 Gbps on the GTX-1080-pascal architecture.

IV. DESIGN OF A PARALLEL AES

The design of parallel implementation of the AES algorithm is chiefly dependent on the choice of grain of parallelism to expose. Taking reference from [1], we divide the parallelism task into two broad section fine grained and coarse grained.

In the initial design stage, we have two ways to parallelize the program. One is parallelism of the algorithm during encryption process, and the other one is reading the input data and divide them into 256 blocks of data. Since the initial version parallelism on algorithm has no improvement. We define coarse grained a solution that ignores internal level of parallelism of the algorithm and focus instead on the Data-level parallelism exposed by operation mode such as ECB. In these schemes different blocks can be encrypted by different instances of the algorithm running in parallel. On the other hand, the fine-grained parallelism focuses on sharing the state matrix between threads to optimize the implementation.

A further improvement design choice is the storage location of S-box and Key. Since they contain exclusively read only data, they may be loaded into the constant memory or shared memory. However, issues like bank conflicts and choice of thread/block came into the picture. A third design choice is overlapping of data transfer and data under processing.

V. OPTIMIZATION STRATEGY

We have divided work of AES on GPGPU into three major implementation techniques:

- Memory Usage Optimization
- Parallel granularity
- CPU-GPU data transfer optimization

A. Memory Optimization

In this subsection, this work focuses on three main parameters: (1) Input plaintext, (2) S-box, (3) Key expansion.

1) Input plaintext: In order to achieve maximum performance, we read our input for 256*16 byte every time for encrypted plain text, which are stored into global memory. Based on the GPU configuration, there is either 256 thread or block being processing parallelized depends on the design choice. The plaintext data is stored in column major wise matrix.
2) S-Box table: Many approaches for accessing AES S-box table using GPU have been done to obtain maximum performance. The best choice is to store in the shared memory as the access speed is faster and the requirement for optimized usage is less stringent.
3) Key expansion: Since the data is not too large, the best approach is to precompute the encryption keys in CPU, and they are only generated once for entire encryption process. The expanded keys are then copied to GPU global memory and then performances are evaluated for different strategy like in constant memory and shared memory configuration.

B. Parallel Granularity

According to the previous works, [1][2][3][4], the approach where each thread can encrypt one whole block. We are using 128-bit AES decoder, each block of data has to go through a 128bit key which is 16 bytes. Thus, 16 byte per thread is the best one, no synchronization and shared data between threads are needed. In our optimization technique we didn't change the work load distribution on the thread.

C. CPU-GPU data transfer optimization

Generally, in GPGPU, the data transfer overhead is substantial. However, to exploit the effective performance, it is necessary to consider the overhead caused by data transfer between CPU and GPU. Generally, to hide this overhead, NVIDIA GPUs provide the function of overlapping data transfer and processing using stream programming model.

VI. EXPERIMENT ANALYSIS

We present different configuration of our implementation described earlier on the GeForce GTX 1080Ti GPU architecture. Following are the main characteristics of this:

Cuda Capability	6.1
Total MP	28
CUDA cores/MP	128
Total amount of Global Memory (MB)	11172
Total amount of constant memory (B)	65536
Total amount of shared memory per block (B)	49152
Maximum number of threads per MP	2048
Maximum number of threads per block	1024

Table II. GTX 1080Ti GPU specification

For the global version, all internal functions are device functions except key_expansion, which is performed on CPU, i.e. host function. We took parallel thread granularity as 16Bytes /thread. We achieved a limited speedup of 2 times with respect to baseline serial version as there is no memory coalescing occurring on the GPU.

```
__global__ void Cipher (unsigned char *in, unsigned
char *out, unsigned char* RoundKey, int Nr)  {
     int i, j, k, round = 0;
     k = threadIdx.x + blockIdx.x*blockDim.x;
     if (k < 1024) {
          unsigned char state [4][4];
          for (i = 0; i < 4; i++)
          {
               for (j = 0; j < 4; j++)
               {
                    state[j][i] = in [16*k + i * 4 + j];
               }
          }
          AddRoundKey (0, state, RoundKey);
          for (round = 1; round < Nr; round++) {
               SubBytes(state);
               ShiftRows(state);
               MixColumns(state);
               AddRoundKey (round, state, RoundKey);
           }
          SubBytes(state);
          ShiftRows(state);
          AddRoundKey (Nr, state, RoundKey);

          for (i = 0; i < 4; i++)
          {
               for (j = 0; j < 4; j++)
               {
                    out [16 * k + i * 4 + j] = state[j][i];
               }
          }
     }
}
```

Algorithm I Global memory version

In the given algorithm I, state [4][4] is local to each thread and major column wise selected. Over this parallel version, the experiment is further optimized in terms of memory and streaming process.

```
__global__ void Cipher (unsigned char *in, unsigned
char *out, unsigned char *RoundKey, int Nr)
{
...
    blk = blockIdx.x;
    if (blk <= 256) {
        __shared__ unsigned char state [4][4];
        i = threadIdx.x;
        j = threadIdx.y;
        state[j][i] = in [16 * blk + i * 4 + j];
        AddRoundKey (0, state, RoundKey);
            (10 Rounds of sub bytes shit rows and mix
the columns)
        out [16 * blk + i * 4 + j] = state[j][i];
}
}
```

Algorithm II shared memory version

For the shared memory (algorithm II), state [4][4] is shared between the rounds for each block. There are different versions have been performed with respect to storage location of key for each round. In first version, Key is stored in the global memory, in second version key is stored in the shared memory, while in the last version key is stored in the constant memory.

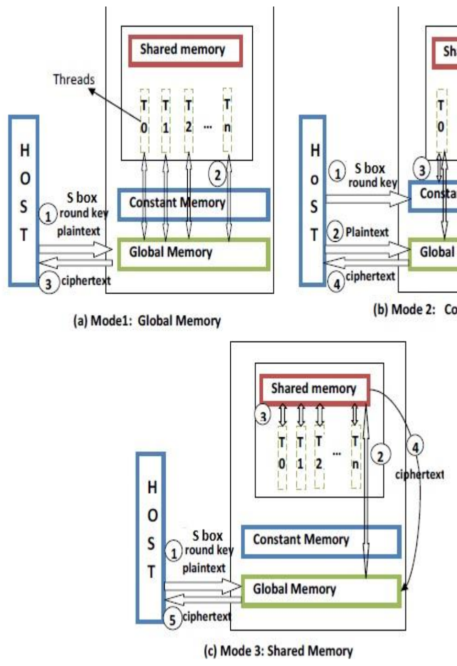

Fig 3. Different modes of optimization strategy

Optimization Strategy	Total time(s)	Kernel time (%)	Kernel time(s)	Average kernel time(us)	Speed up
Shared memory (key in global memory)	12.407	98.81	4.04	200.16	3.95x
Shared memory (key in shared memory)	13.502	98.82	4.61	228.77	3.64x
Shared memory (Key in constant memory)	13.126	98.99	4.09	202.81	3.74x

Table III Execution time analysis in different optimization modes

The batch size for the data parallelization is taken as 256. So, the input data will be sent as a chunk of 256*16 bytes for data parallelization. From the GPU perspective, the kernel occupancy level is only 46.2% as it is limited by the block size. Each kernel invocation consumes 8KiB shared memory and shared memory per block is 16B with registers per thread equal to 32.

From the general configuration of memory, it is pre-assumed that the configuration with key storage in shared memory will be fast but the experiment result shows that key storage in global memory is fast (the default case),then the key storage location in constant memory is faster than the shared memory. It is due to the fact of bank confliction. Shared memory inherently causes bank conflict since the key is accessed 10 time when it is stored in the shared memory.

While in global memory and constant memory, it prevents bank conflicts as it can't be modified. Moreover, the impact is less in small file as every thread has to reach constant memory for each round.

The experiment also analyses the effect of increasing number of threads on the execution, but it decreases with increase in thread numbers as it will break the shared memory coherence and create more bank conflicts and thus more complicate kernel.

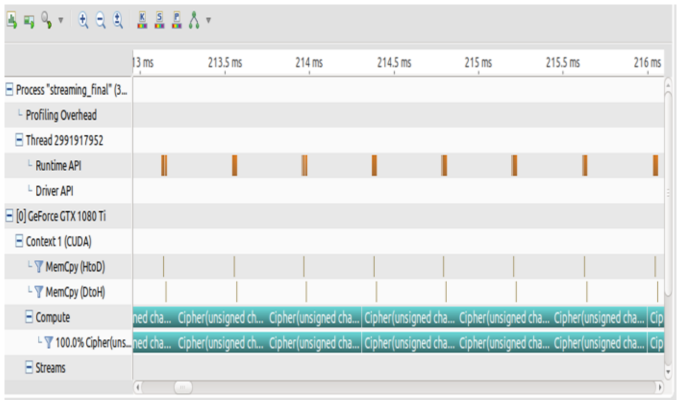

Fig 4. kernel streaming for global memory version

Parallelism can also be achieved in between CUDA - memory copy and the kernel computation using streaming. Fig. 4 depicts the global streaming version. The total overlapping is around 4-5% depending upon the time execution of memory copy to kernel. The streaming is somewhat helpful in global version while due to large overhead and cuda Device-synchronization operation, it seems to poor overlapping in the shared memory version.

Fig 5 shows the experiment where different sizes of files and their execution time is measured with our optimized code and the CPU as baseline code. The experiment measures the speedup of the code when tested on different input file size.

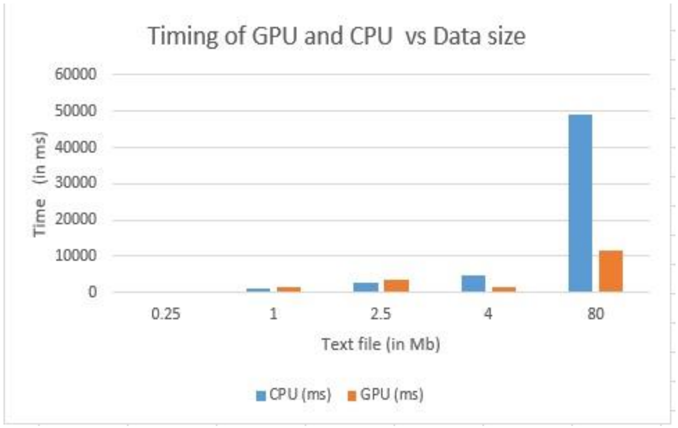

Fig 5. Timing of GPU& CPU vs data size

For the small input file size, CPU-GPU memory transfer takes more time than the computation, so serial version seems to be fast, while for the large input file size, the computation time is greater than the communication time so overall time for GPU is less than the CPU version.

VII. CONCLUSION

In summary, we have performed an experimental study of the CUDA based AES implementation. A detailed investigation with different optimization technique has also been performed with respect to baseline serial implementation. We also embraced the memory configuration issues related to hardware such as bank conflicts and number of blocks limiting case. The study also explores the deign issue of data streaming. Overall, we successfully achieved the 4 times speedup with a 2.75 Gbps overall throughput.

However, the results of the streaming as well as throughput is not satisfactory as compared to other authors. So, in addition to improvement in implementation, one can leverage the issue of the above and also can implement decryption and encryption with other modes of block like CBC, CTR to extract parallelism as future work.

REFERENCES

[1] Biagio D.A., Pelosi G. et.al., Design of a Parallel AES for Graphics Hardware using the CUDA framework. IEEE, International Parallel and Distributed Processing Symposium (IPDPS), 2009.

[2] Mei C., Jiang H., et.al., CUDA -based AES parallelization with fine-tuned GPU memory Utilization. IEEE, International Parallel and Distributed Processing Symposium (IPDPS), 2010.

[3] Iwai K., Nishikawa N., et.al., Acceleration of AES encryption on CUDA GPU. IEEE, International Journal of Networking and Computing (IJNC), 2012.

[4] Abdelrahman A. A., Fouad M.M., et. al., High Performance CUDA AES Implementation: A quantitative performance approach, IEEE, Computing Conference, 2017.

[5] Guao liang G., Quian Q., et.al., Different implementations of AES Cryptographic Algorithm. IEEE, International conference on High performance Computing and Communications (HPCC), 2015.

[6] Li Q., Zhong C., et.al., Implementation and Analysis of AES encryption on GPU. IEEE, International Conference on High Performance computing and communications. (HPCC), 2012.

[7] Nvidia, “CUDA C Programming Guide,” [Online].

[8] Nvidia, “GPU GEM3”. [online].