High-throughput Implementation of The Advanced Encryption Standard Using a GPU

E4750 Heterogeneous Computing for Signal and Data Processing - Student Project - Prof. Z. Kostic

Alexander Ranschaert
Department of Electrical Engineering
Columbia University
New York, USA
anr2157

Ryan De Koninck
Department of Electrical Engineering
Columbia University
New York, USA
rd3033

***Abstract*—Recently, GPUs have become a go-to solution for accelerating demanding applications. In this report, the authors present a parallel implementation of the Advanced Encryption Standard (AES), using the PyCUDA framework. It was investigated how to map the resources required for this algorithm onto the CUDA memory hierarchy to achieve the best possible performance. The different versions were compared and profiled. For input data larger than 1MB, every version is faster than an existing CPU-based implementation. The best implementation was found to be 6x faster compared to the CPU for large files, and has a throughput of 20.54Gbps.**

***Index Terms*—GPU, Heterogeneous Computing, CUDA, Cryptography, Advanced Encryption Standard**

I. Introduction

The Advanced Encryption Standard (AES) is one of the most influential cryptographic algorithms used to securely transfer data. It is the result from a five-year long competition from NIST, in which the RijnDael block cipher was announced as the victor in 2001. AES is a symmetric cryptographic algorithm, meaning that the key for encryption and decryption are one and the same. Thus, in order for encryption to be secure, the key must be shared between the sender and receiver through a secure channel. As mentioned, the AES algorithm is a block cipher, meaning that the plaintext data to be encrypted will be divided into blocks. On each of these blocks, a sequence of transformations is performed, which eventually results in the encrypted data. Thanks to the symmetry that characterizes the algorithm, the decryption happens in a very similar way. The exact algorithm is described in detail in the revised version of the original proposal of RijnDael [1], and in the announcement note provided by NIST [2]. The importance of the algorithm, and its necessity to be sped up, are confirmed by Intel's introduction of AES-specific instructions [3].

The sequence of transformations that is applied to each block is, unfortunately, inherently not parallelizable. However, the encryption and decryption of each block is independent of every other block. This presents an opportunity for parallel processing, especially in this world where massive amounts of data are processed securely using AES. The exploitation of exactly this opportunity is described in this report. The potential of the parallelization of AES on GPUs has already been analyzed and realized in a plethora of academic work, resulting in one to two orders of magnitude performance increases over the sequential implementation on CPU [4] [5]. On Github, there are also multiple implementations of AES available, for example in Python [6] and in CUDA [7]. However, it is difficult to find sources that report exactly how the memory hierarchy influences the performance of the GPU-based implementation. Thus, this report describes various CUDA implementations of the AES algorithm using different memory organizations, and compares them.

The following section details the goals and objectives in Subsection II-A, and presents the system and software design in Subsection II-B by using pseudo code of the algorithm, a flowchart and a block diagram. Section II ends by providing some other, more practical, implementation details. The results are given in Section III, including an overview of the platform used for testing and timing in Subsection III-A and a comparison of the different implementations realized during this project in Subsection III-B. Finally, the work is discussed and further work is suggested in Section V.

II. Description

A. Goals and Objectives

During this project, both the encryption and decryption algorithms of AES are implemented to run on a GPU. To be exact, the AES algorithm with a key length of 128 bits and a block size of 128 bits is implemented. The goal of the project is to develop different implementations of the algorithm that use different memory organizations on the GPU, compare them, and report them in detail. Both AES encryption and decryption are implemented, but because of the symmetry of the algorithm it is only deemed necessary to investigate one of the two: in this project, the encryption implementations are compared. Additionally, the exploration of the different memory hierarchies is based on achieving

an optimal throughput speedup with respect to the sequential implementation on CPU.

B. System and Software Design

1) The AES Algorithm: As mentioned during the introduction, AES is a block cipher that sequentially performs transformations at a block granularity. The block size for AES is 128 bits, or 16 bytes. The transformations are organized in *rounds*. This behavior is described in pseudo code below for the encryption, which has been derived from the revised version of the original proposal of RijnDael [1]. Note that the division of the input in blocks is not explicitly given in the pseudo code.

```
AES(State, CipherKey) {
    KeyExpansion(CipherKey, ExpandedKey);
    AddRoundKey(State, ExpandedKey);
    for (i = 1; i < Nr; i++)
        Round(State, ExpandedKey + Nb*i);
    FinalRound(State, ExpandedKey + Nb*Nr);
}
```

The algorithm takes the `State` and the `CipherKey` as its inputs. The `State` is the plaintext data (in the case of encryption), which has been padded to a multiple of the block size (and divided into blocks that are individually processed). The first step of AES is to perform the `KeyExpansion`, which generates the round keys for the different `Round` operations based on the original `CipherKey`. The implementation of the `KeyExpansion` operation can be found in [1]. Then, a first `AddRoundKey` operation is performed. This operation simply performs a bitwise XOR between the `State` and the original `CipherKey`. The majority of the algorithms' execution time is spend in the for loop that performs the `Round` operation `Nr - 1` times. `Nr` defines the total number of `Round` operations to be performed, including the `FinalRound`. The `FinalRound` operation finalizes the algorithm, and outputs the encrypted data.

The pseudo code for the `Round` and `FinalRound` operations is given below. Again, these have been taken from [1].

```
Round(State,RoundKey) {
    ByteSub(State);
    ShiftRow(State);
    MixColumn(State);
    AddRoundKey(State,RoundKey);
}

FinalRound(State,RoundKey) {
    ByteSub(State) ;
    ShiftRow(State) ;
    AddRoundKey(State,RoundKey);
}
```

The `Round` operation performs four transformations on the `State` given the `RoundKey` for that round:

A) `ByteSub`: Uses each byte of the input `State` as the index for a look-up table called the S-box.

B) `ShiftRow`: The input `State` is a one-dimensional array that contains the original block data in bytes, organized in column-major order. The `ShiftRow` operation will cyclically shift every row of the column-major ordered `State`, by an amount determined by the row number. In the 16 byte block used in AES, each block can be represented as a two dimensional array with four rows and four columns. The first row does not get shifted, the second gets shifted by one place, the third by two and the fourth by three.

C) `MixColumn`: Performs a matrix-vector multiplication for every column of the input `State`:

$$\begin{pmatrix} b1' \\ b2' \\ b3' \\ b4' \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} b1 \\ b2 \\ b3 \\ b4 \end{pmatrix}$$

Important to note is that "the columns of the `State` are considered as polynomials over $GF(2^8)$ and multiplied modulo $x^4 + 1$ with a fixed polynomial $c(x)$, given by $c(x) = 03x^3 + 01x^2 + 01x + 02$," as stated in [1]. Consequently, performing these operations is not as simple as a normal matrix multiplication. For that reason, look-up tables are used to implement the multiplication by 2 and 3. Furthermore, The additions performed during the matrix multiplication are bitwise XORs.

D) `AddRoundKey`: As mentioned before, this operation performs a simple bitwise XOR between the input `State` and the `RoundKey`.

The `FinalRound` operation performs the same transformations as the `Round` operation, except for the `MixColumn` operation. For a more in-depth review of the algorithm and each of the transformations, see [1] or [2].

2) Flowchart: The algorithm presented above is implemented to run on a GPU. To understand how the algorithm is implemented on GPU, and how the data flows, consider the flowchart in Figure 1.

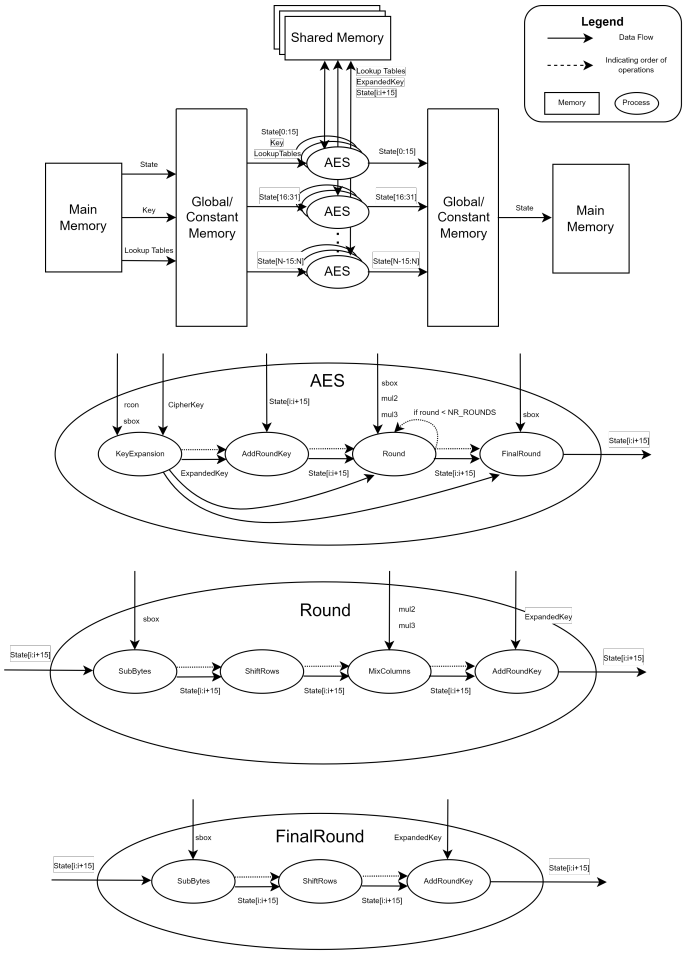

Fig. 1. Flowchart visualizing the data flow and the design on GPU.

The input `State`, the `CipherKey` and the look-up tables are sent from the main memory to the global and/or constant memory of the GPU. Each thread on the GPU executes the AES algorithm for one block (16 bytes) of the input `State`, using the shared memory in some implementations to store the look-up tables, the `ExpandedKey`, and/or the `State`. After completing the computation, each thread loads their results back into the global memory of the GPU. Finally, the result is written back to the main memory.

The data flow of the AES algorithm and the sequence of operations has also been visualized in Figure 1. This visualization emphasises the hierarchical approach that has also been used to implement the AES algorithm on the GPU to maximize reuse of code and thus minimize the amount of code written.

3) Block Diagram of the Memory Organization: To clarify the memory organization, Figure 2 visualizes where in the memory hierarchy the different data required for the execution of the AES algorithm are stored. Figure 2 displays the memory hierarchy one of the highest performing memory organizations discussed in this report, but it can be used for reference for the description of the other memory hierarchies as well.

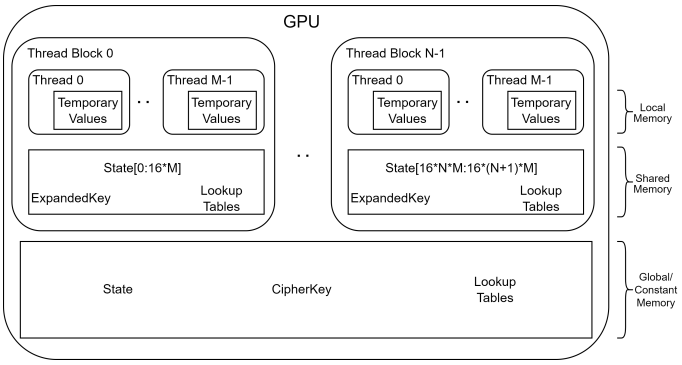

Fig. 2. Block diagram visualizing the organization of the memory hierarchy on the GPU and indicating the possibilities for storing the algorithm's data structures.

4) Other Implementation Details: The AES algorithm is implemented in PyCUDA. The CPU, executing the Python code, is used to control the GPU, execute test cases, and collect timing results. The CUDA kernel is called by the Python code, and includes the code to execute encryption or decryption from start to finish, as described in the flowchart above. The code for this project can be found on Github[1]. To create the reference sequential implementation, the `cryptography` Python library by the Python Cryptography Authority was used [8]. To streamline the verification process, the `pytest` framework is used [9]. Both standard tests (specified by NIST [2]) and tests created with the reference implementation were used to verify that the code works as intended. The dependencies are detailed in the `README` on the project Github page.

III. Results

A. Platform

Table I details the platform used to test and time the various implementations presented in the following section.

[1]https://github.com/eecse4750/e4750_2022fall_project-paes-rd3033-anr2157-

TABLE I
System Configuration

OS	Ubuntu 18.04 LTS
CPU	Intel(R) Xeon(R) CPU @2.00GHz (4vCPU)
Memory	15GB
nvcc Version	release 10.2, V10.2.89
GPU	Nvidia Tesla T4
GPU Memory	16GB

B. Performance

Before trying to speed up the algorithm as a whole by exploiting the different memories of the CUDA hierarchy, the individual operations that constitute the rounds are timed (without memory transfer included), in order to identify any potential bottlenecks. From Figure 3, it can be concluded that there are no large differences between the execution time of these elementary operations, and that optimizing the memory structure will result in the largest performance gains.

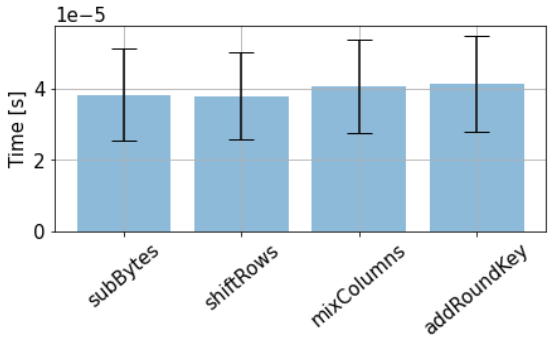

Fig. 3. Execution time (excluding memory transfer) of the most basic operations on the state.

Now, different memeory organizations will be considered. An overview of these different implementations is given in Table II. The CPU implementation in the following figures is indicated as (7), and is part of the Python cryptography library [8] as mentioned before. As mentioned in Section II-B, many bit-operations were implemented using constant look-up tables, so the location of these in memory will be crucial, as they are constantly in use. Since they do not change, they were either placed in constant memory (1, 2, 3, 5) or in shared memory (4, 6). The state that is being manipulated was put in the global memory for the naive version (1), and in shared or local memory for later versions (2-6). The expanded key was always placed in the shared memory. Since the AES standard refers to the state in column-major fashion, two designs (3,4) also used coalesced memory accesses to the state.

First, the ideal number of threads per thread block was determined. It could be disadvantageous to select a very large thread block size, as this could have resulted in having only a single thread block per streaming multiprocessor, which could negatively impact the performance. It turned out that this was not the case for the experiment in Figure 4, where 1GB of data was used. A small thread block size on the other hand will cause the GPU latencies to be no longer hidden. From Figure 4, it can also be concluded that the largest improvements in the overall performance come from avoiding the constant memory for the look-up tables, as seen for (4) and (6). Placing the state itself in local (6) or shared memory (2-4) does not have a huge impact, but it can be observed that putting the state in local memory is slightly faster. This is possible because each single thread operates on a different state, which was the proposition for parallelization. Placing the state in global memory (1) does have

TABLE II
DESCRIPTION OF THE IMPLEMENTATIONS

Where is it stored?	(1)	(2)	(3)	(4)	(5)	(6)
LUTs	Constant Memory	Constant Memory	Constant Memory	Shared Memory	Constant Memory	Shared Memory
State	Global Memory	Shared Memory	Shared Memory	Shared Memory	Local Memory	Local Memory
Coalesced access to state?	n.a.	No	Yes	Yes	No	No
Expanded Key	Shared Memory	Shared Memory	Shared Memory	Shared Memory	Shared Memory	Shared Memory

a significant negative impact.

If the thread block size is greater than or equal to 256 and a power of 2, the performance does not change with increasing block size. Consequently, 256, 512, and 1024 threads per thread block are all appropriate choices. For the remainder of the report, a thread block size of 512 is selected.

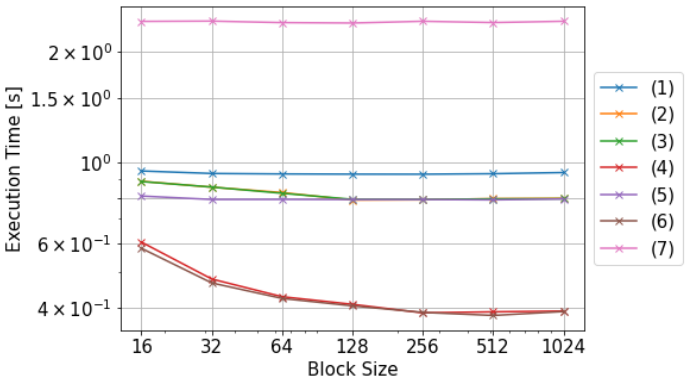

Fig. 4. Execution time (including memory transfer) vs thread block size (1GB of data, averaged over 25 iterations).

Figure 5 visualizes the execution times of the various implementations in function of the size of the input. It results in similar conclusions as above with respect to which implementations are the highest performing. If the size of the input data is larger than 1MB, every GPU implementation outperforms the sequential implementation on CPU. For larger input files, the best implementation (6) is around a factor 6 faster than the CPU version.

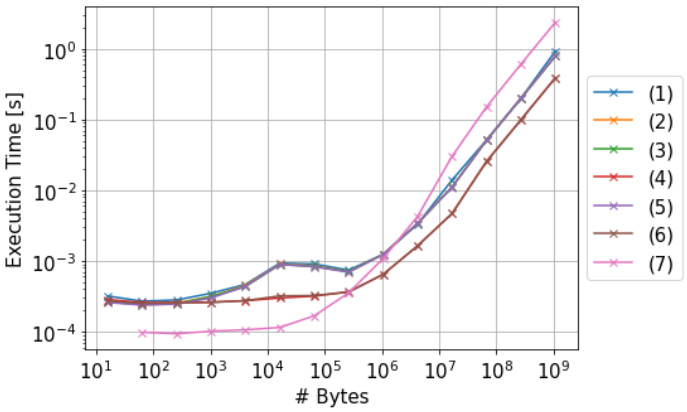

Fig. 5. Execution time (including memory transfer) vs input size (512 threads/block, averaged over 25 iterations).

To better understand why one version performs better than another, the different kernels were each profiled. The results for the most instructive cases are plotted in Figure 6. For the slower implementations, for which the LUTs are stored in constant memory ((1) and (3)), the GPU's available memory bandwidth is underutilized, while the streaming multiprocessors (SMs) are active for a long time to fetch the data. The memory is therefore clearly the bottleneck for the first 2 cases. When the LUTs are placed in the shared memory, the available bandwidth is used more efficiently, but the SMs are active for a smaller fraction of the total execution time. While moving the state from shared to local memory ((4) to (6)) did not have serious implications for the execution time, this does affect the execution under the hood. The L2 cache is now a lot more active, which most likely explains the tiny increase in performance for the last case (6). The throughput of this last case is 20.54Gbps.

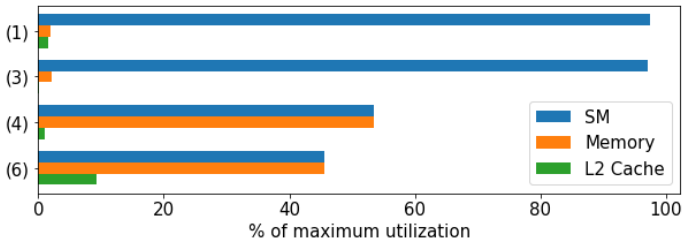

Fig. 6. Profiling

IV. Demonstration

Any `.txt` file can be encrypted or decrypted running the *AES.py* script as follows:

```
foo@bar:~$python AES.py input.txt
↪  output_encrypted.txt output_decrypted.txt
```

The input file should be present in the folder before running the script. The two output `.txt` files will then be automatically generated. Demo input and output files are present in the Github repository.

V. Discussion and Further Work

When the maximum throughput was reported, this included the memory transfer from host to GPU, which is responsible for a large portion of the total time. It is therefore crucial to consider this when trying to maximize the overall throughput. In this work, we do not compare the throughput over different system configurations, so the proposed implementation could very well perform a lot better on different systems. 20.54Gbps is lower than what was achieved in recent work, where the throughput was found to be 60Gbps [4]. This difference is still reasonable however. The reported speedup with a factor of 50, compared to the CPU should also be taken with a grain of salt, as the CPU version they used had a throughput of 1.2Gbps, which seems low compared to the throughput of 3.36Gbps that we have found, with a basic Google Cloud CPU.

The causes of the underutilization of GPU resources is something that should be investigated further if future goals are to get the maximum possible performance out of the GPU. Furthermore, there

are different implementations of AES that could result in a better speedup. In this project, the most basic version of AES using byte operations was implemented. However, it is possible to implement the algorithm using integer operations instead, and doing so allows an implementation with a set of four different look-up tables as mentioned in [1]. This would result in a more efficient implementation, as the new lookup tables 'already calculated' a lot of operations in advance.

VI. Conclusion

In this work, it was shown that by carefully managing the memory, the AES algorithm can be accelerated on a GPU by a factor of 6 compared to a CPU for the encryption of large files. This resulted in a throughput of 20.54Gbps. Several routes were investigated, and the best solution was obtained by placing the AES states in local memory and the look-up tables in shared memory, as this makes better use of the available memory bandwidth. Profiling revealed that the GPU's resources were still not fully utilized, so future implementations could still achieve an even better performance.

References

[1] J. Daemen and V. Rijmen, "Note on naming Rijndael Note on naming," Apr. 2003. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Standards-and-Guidelines/documents/aes-development/Rijndael-ammended.pdf

[2] NIST, "Federal Information Processing Standards Publication 197 Announcing the ADVANCED ENCRYPTION STANDARD (AES)," 2001. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf

[3] [Intel, "Intel® Advanced Encryption Standard (AES) New Instructions Set ," May 2010. [Online]. Available: https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf

[4] Q. Li, C. Zhong, K. Zhao, X. Mei and X. Chu, "Implementation and Analysis of AES Encryption on GPU," 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems, 2012, pp. 843-848, doi: 10.1109/HPCC.2012.119.

[5] A. A. Abdelrahman, M. M. Fouad, and H. Dahshan, 'Analysis on the AES implementation with various granularities on different GPU architectures', Advances in Electrical and Electronic Engineering, vol. 15, no. 3, pp. 526–535, 2017.

[6] BoppreH, "What's in the box," GitHub, Dec. 01, 2022. https://github.com/boppreh/aes (accessed Dec. 14, 2022).

[7] G. Narayanan, "Parallel AES Cryptography," GitHub, Nov. 12, 2022. https://github.com/gurupunskill/parallel-aes (accessed Dec. 14, 2022).

[8] "Welcome to pyca/cryptography — Cryptography 3.0.dev1 documentation," cryptography.io, 2022. https://cryptography.io/en/latest/

[9] H. Krekel and pytest-dev team, "pytest: helps you write better programs — pytest documentation," docs.pytest.org, 2022. https://docs.pytest.org/en/7.2.x/

Appendices

Individual Student Contributions (in %)

See Table III for the contributions to this project per student.

TABLE III
Individual Student Contributions

Task	Contribution [%]	
	Alexander Ranschaert	***Ryan De Koninck***
Setup (testing/timing)	20	80
Implementation of encryption	30	70
Implementation of decryption	100	0
Report	50	50
Presentation	50	50
Total	**50**	**50**