Fast AES Implementation: A High-Throughput Bitsliced Approach

Omid Hajihassani, Saleh Khalaj Monfared, Seyed Hossein Khasteh, and Saeid Gorgin

Abstract—In this work, a high-throughput bitsliced AES implementation is proposed, which builds upon a new data representation scheme that exploits the parallelization capability of modern multi/many-core platforms. This representation scheme is employed as a building block to redesign all of the AES stages to tailor them for multi/many-core AES implementation. With the proposed bitsliced approach, each parallelization unit processes an unprecedented number of thirty-two 128-bit input data. Hence, a high order of prallelization is achieved by the proposed implementation technique. Based on the characteristics of this new implementation model, the ShiftRows stage can be implicitly handled through input rearrangement and is simplified to the point where its computing process can be neglected. In this implementation, costly Byte-wise operations are performed through register shift and swapping. In addition, the need for look-up table based I/O operations, which are used by the Substitute Bytes stage is eliminated through using S-box logic circuit. The S-box logic circuit is optimized to simultaneously process 32 chunks of 128-bit input data. We develop high-throughput CTR and ECB AES encryption/decryption on 6 CUDA-enabled GPUs, which achieve 1.47 and 1.38 Tbps of encryption throughput on Tesla V100 GPU, respectively.

Index Terms—AES, CTR, ECB, GPU, data representation, CUDA, high-performance

1 Introduction

The emergence of high-throughput parallel processing units offers incentives such as gaining better execution performance to different domains of compute-intensive applications. The trend of applications migrating from crude sequential execution on conventional hardware to high-performance implementation on massively parallel platforms has hiked in popularity among various research domains. Among these, medical imaging [1], machine learning [2], [3], [4], and data science [5] are a few number of domains that attract the attention of different scholars and researchers to parallel programming.

Data encryption is an indispensable part of today's digital world guaranteeing protection and confidentiality of data transfer and storage amongst a diverse user group ranging from conventional users to renowned enterprises and companies. Throughout the years, different encryption techniques and methods have been introduced to meet the aforementioned need for security among which are several symmetric and asymmetric data encryption algorithms [6], [7], [8], [9]. The Advanced Encryption Standard (AES) [7] is a complex symmetric block cipher designed to succeed the Data Encryption Standard (DES & 3DES) [6]. AES is designed specifically to be not prone to the attacks that DES has shown vulnerability to such as the brute-force attack [10].

Inseparable features of these days' digital world are bulky and large data sets having four traits of volume, variety, velocity, and veracity referred to as the big data [11], [12]. Unfortunately, most of the encryption standards and algorithms at hand were not in any way designed or implemented to meet the demands of the aforementioned features of big data [13], [14]. So, hereby, we strive to propose a data representation scheme that is integrated into the bitsliced implementation of Electronic Codebook (ECB) mode and Counter (CTR) mode AES [15] that enables them to score the highest published and reported encryption throughput, which is achievable on conventional GPUs and is also scalable to high-performing GPUs. Through this, we meet the needs of a wide range of users by accommodating tools of high-throughput AES cryptography.

In order to achieve this goal, we set off to design a bitsliced AES implementation that eliminates the computationally intensive operations in all the stages involved in the AES. Moreover, it empowers each processing unit to process more input data at the same time. First, all of the AES stages embed in themselves numerous instructions that require frequent shift and mask operations at the Byte granularity. In

- *O. Hajihassani is with the Institute for University of Alberta, and also Research in Fundamental Sciences (IPM), No. 1, Shahid Farbin Al-ley, Shahid Lavasani st., Tehran, Iran, P.O. Box 19395-5531. E-mail: hajihass@ualberta.ca.*
- *S. Khalaj Monfared is with the Institute for Research in Fundamental Sciences (IPM), Tehran, Iran and also with the K. N. Toosi University of Technology No. 1, Shahid Farbin Alley, Shahid Lavasani st., P.O. Box 19395-5531, Tehran, Iran. E-mail: monfared@email.kntu.ac.ir.*
- *S. H. Khasteh is with the K. N. Toosi University of Technology, Faculty of Computer Engineering, Seyed Khandan, Shariati Ave, P.O. Box 16315-1355, Tehran, Iran. E-mail: Khasteh@kntu.ac.ir.*
- *S. Gorgin is with the Electrical Engineering and Information Technology Department of Iranian Research Organization for Science and Technology (IROST), Tehran, Iran and also with the Institute for Research in Fundamental Sciences (IPM) P.O. Box 19395-5531, Tehran, Iran. E-mail: gorgin@ipm.ir.*

Manuscript received 29 Mar. 2018; revised 2 Apr. 2019; accepted 4 Apr. 2019. Date of publication 15 Apr. 2019; date of current version 11 Sept. 2019. (Corresponding author: Saeid Gorgin.) Recommended for acceptance by D. Talia. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TPDS.2019.2911278

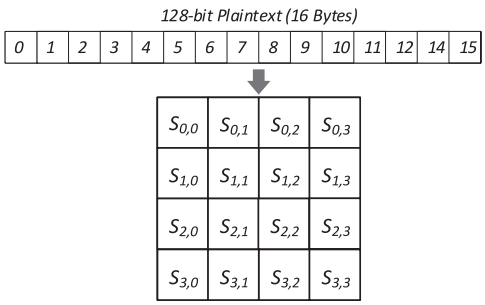

Fig. 1. Organization of 128-bit input into the state array.

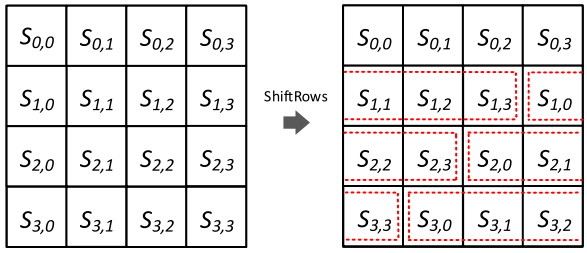

Fig. 2. ShiftRows stage of AES.

platforms that do not support fine-grain Byte-wise operations, these instructions greatly reduce the encryption and decryption throughput. In our proposed implementation model, we introduce a data representation scheme and use it in the alteration of the AES stages so that they have far less costly Byte-wise operations. This data representation model exploits register shuffle and swapping instead of the costly Byte-wise shift and mask operations. This is done by changing the representation of the data from the commonly used row-major to the column-major representation to achieve our predefined goal of high-throughput encryption. Moreover, in our bitsliced implementation, AES ShiftRows stage is implicitly handled through argument rearrangement in the MixColumn stage inputs. Second, with the previously mentioned data representation model, the proposed bitsliced implementation enables each executing thread to operate on 32 numbers of 128-bit input data at the same time. By combining this feature with the diminished need for excessive Byte-wise operations, the proposed bitsliced AES implementation achieves unprecedented orders of speedup.

Here, the ECB and CTR modes of operation for AES are implemented with our proposed bitsliced technique. The proposed representation model is similar to what we have used to implement the bitsliced DES cryptanalysis [10]. Our GPU implementation that is discussed in details, fruitfully employs the proposed data representation model and all of the GPU resources such as the shared memory resulting in peak encryption throughput. In our work, each GPU thread simultaneously operates on a batch of 32×128-bit data. It is worth noting that the proposed data representation model is not limited to a predefined number of GPUs or the Compute Unified Device Architecture (CUDA) [16] and can be fully incorporated in implementations aimed at other available parallel hardware platforms.

The remaining of this paper is structured as follows: We give a background on AES in Section 2. Section 3 discusses the related literature on the proposed methods to increase the AES encryption throughput and their published and reported cryptography evaluation results. In Section 4, our proposed bitsliced technique is discussed in depth and its incorporation into all of the AES stages is elaborated on. In Section 5, different AES modes of operation are reviewed and the underlying reason for implementing only the ECB and the CTR modes is outlined. Section 6 gives the ECB and CTR modes implementation results on different CUDA-capable GPUs and compares these results to previous works, furthermore evaluation and comments are given for the AES decryption procedure. Finally, Section 7 concludes the paper and outlines the intended future work.

2 Background

The basics of the Advanced Encryption Standard was introduced by Rijmen and Daemen in their proposal to the National Institute of Standards and Technology (NIST) in a call for a superseder of the Data Encryption Standard (DES & 3DES) [6], which is proved to be prone and vulnerable to many attacks such as the brute-force attack [10]. NIST published AES in 2001 and since then AES has been implemented and used by many different cryptography libraries and packages [7].

AES is a symmetric block cipher algorithm, which takes in 128-bit plain/ciphertext input and encrypts/decrypts it with differently sized encryption keys of 16, 24, or 32 Bytes (i.e., 128, 192, 256 bits). AES operates based on the size of the encryption key and is referred to as the AES-128, AES-192, and AES-256 for the 128-bit, 192-bit, and 256-bit encryption key sizes, respectively. In the original AES documentation and implementation, the 128-bit input is divided into a 4×4 array of 8-bit elements. This original two-dimensional representation scheme is employed in references to illustrate the AES internal stages and is referred to as the state array. The organization of the input data in the state array is illustrated in Fig. 1, where $S_{m,n}$ equals the $(4 \times m + n)^{th}$ data Byte segment. AES, unlike DES, is not a Feistel structure in the sense that all of the input is processed at each round of AES instead of the common one-half at a time processing in the Feistel-like block ciphers.

The encryption in AES includes the expansion of key and a series of permutations and substitutions at the Byte-level. In the Substitute Bytes stage, an S-box Look Up Table (LUT) is used for Byte-wise substitution of the blocks. Many implementations normally use S-box LUT. However, due to the original substitution stage being based on a firm mathematical foundation, one can calculate the substitute Byte for any given input Byte by using the mathematical formulae and operations in $GF(2^8)$. In this paper, instead of employing, the I/O bound S-box LUT; we set off to employ the mathematical approach proposed in [17] by incorporating our proposed data representation into it. By doing so, our parallelization units will no longer be imposed to the burden of costly I/O instructions. This approach to the S-box calculation immunes our AES implementation to timing attacks. Another stage of AES is the ShiftRows that includes simple Byte-level shifts and follows simple permutation steps. The permutations employed by ShiftRows is illustrated in Fig. 2.

The MixColumn stage operates on each column of the state array. In each column, every single element (i.e., Byte) is substituted by a value that is a function of all the elements in that column. The MixColumn function is handled by

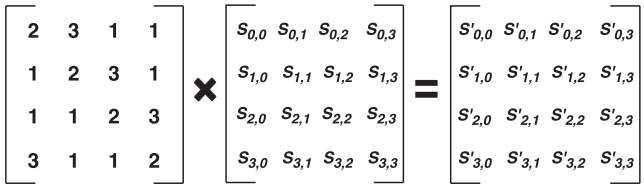

Fig. 3. MixColumn Stage performed by a matrix multiplication in $GF(2^8)$.

multiplication of the specific MixColumn matrix with the state array, which is illustrated in Fig. 3. The multiplication operations are in Galois Field $(GF(2^8))$.

The AddRoundKey stage simply XORs each round's key with its input block. The round key size is the same in all the types of AES and is built from the expanded encryption key. AES encryption begins with a single AddRoundKey stage followed by 9, 11, 13 (AES-128, AES-192, and AES-256) rounds including all the AES internal stages (Substitute bytes, ShiftRows, MixColumn, and AddRoundKey). This is continued by a single round including only three stages, excluding the MixColumn stage.

3 Related Works

Since its introduction, AES was the subject of different improvement and enhancement techniques targeted at its design and implementation. In the second version of the Rijndael document [18], Rijmen and Daemen proposed the use of T-box look-up table instead of the initially improvised S-box look-up table employed in the Substitute Bytes stage of AES. The proposed T-box approach fuses together the Substitute Bytes and the MixColumn stages from the original AES and undertakes them by the use of preprocessed look-up tables. Moreover, it is possible that one can implement and employ mathematical formulae in Galois finite field to directly calculate the tables instead of LUTs.

In the direct mathematical implementation of the Substitute Bytes stage, many strived to reduce the size of the 8-bit Galois field S-box to smaller 4 and 2-bit calculations [19], [20]. The approach that we built upon and modified to fit our proposed data representation scheme is discussed in [17]. This work gives a compact implementation of S-box by employing the "tower field" approach from [21]. This approach to S-box by using subfield arithmetic and logic gate optimization can be implemented requiring fewer logic gates and hence requiring fewer instructions in software design.

AES is implementable on different platforms such as microcontrollers, CPU, GPU, and FPGAs. For instance, OpenSSL is a fully commercial cryptography toolkit for TLS and SSL protocols [22] which accommodates different AES modes of operation such as electronic codebook (ECB), cipher block chaining (CBC), and output feedback (OFB) supporting a wide range of platforms including 32 and 64-bit ARM and Intel CPUs. Due to the popularity of AES, Intel designed a specific instruction set dedicated to the AES cryptography [23]. This set of instructions is referred to as AES-NI. On a 2.7 GHz, Intel Core i5-5257U CPU, OpenSSL using AES-NI outperforms the OpenSSL naïve implementation by roughly six-fold. AMD offers the Secure Memory Encryption (SME) [24], which performs encryption on the data stored in DRAM memories. SME secures data from different attacks offering fast in-memory AES encryption to a vast range of applications.

TABLE 1
Proposed Works on AES

Ref.	Mode	Throughput (Gbps)	Platform	Year
[26]	CBC	34	GTX 580	2011
[35]	ECB	60	C2050	2012
[32]	CTR	72	C2050	2012
[34]	ECB	205	HD 7970	2014
[33]	CTR	149.5	GTX 980	2016
[31]	ECB	78.6	GTX 480	2016
[30]	ECB	605.9	P100	2017
[27]	ECB	79	K40m	2017
[28]	ECB	280	GTX 1080	2017

The authors of [25] proposed a technique for an efficient design of the MixColumn stage on FPGA. This technique reduces the multiplication by 3 of each element to a single multiplication by 2 and an addition in GF(2^8), instead of calculating a separate multiplication by 3 for every element in the state array. In our implementation, we utilize and modify this MixColumn technique.

For the GPU, there have been many works and implementations to utilize the high-throughput parallel computation power for the AES algorithm. Among all of these works, for instance, [26] gives an implementation of AES in CBC mode, relying on the fact that the decryption procedure in this mode could be implemented in parallel which results in much faster run (roughly 35 Gbps) on the GPU compared to the AES-NI [23]. Furthermore, [27] gives an implementation based on T-box look-up table, with round keys and T-boxes stored in the shared memory. This work investigates the impact of different input sizes on the overall performance of the GPU. [28] presents an optimization on the AES by accommodating 32 Bytes per thread in GTX 1080, achieving high-throughput speed of 280 Gbps in the ECB encryption.

The bitsliced implementation was employed by Eli Biham in DES cryptography [29]. In this approach, a 64-bit CPU was viewed as 64 $\times$ 1-bit CPUs working in SIMD manner on the input data. Therefore, the costly shift and mask instructions in permutation and expansion operations were replaced by simply referencing different registers. Many adapted this representation view and proposed the bitsliced AES implementation [30], [31]. Due to the evolution of GPU hardware, it is now possible to implement the bitsliced AES on different GPUs. Reference [30] proposed a bitsliced ECB-AES implementation on GPU with different and flexible number of plaintext blocks operated on by a single thread. The aforementioned work roughly achieves 606 Gbps in the configuration where each thread packs four plaintext chunks into eight 64-bit variables referred to as the Bs64. Works from [32] and [33] have implemented the CTR operation mode for AES on modern GPUs. In Table 1, compilations of all these works alongside other look-up table-based implementations [34], [35] are given. Furthermore, Appendix G gives more detailed information for each of these works.

The problem with the already proposed bitsliced approaches is their view on representing the plaintext input. The utilized representation in these works demands costly Byte-level shift and mask operations at different stages including the ShiftRows. Our proposed representation, on the other hand, eliminates the need for the aforesaid costly operations

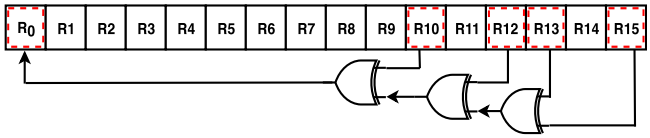

Fig. 4. The 16 bits LFSR for pseudo random number generator.

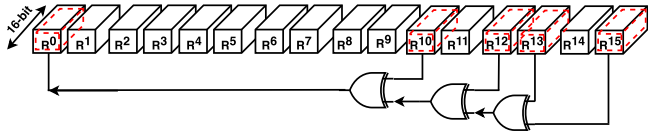

Fig. 6. The bitsliced SIMD approach with column-major representation for LFSR circuit.

throughout all of the AES stages. In addition, stages like Substitute Bytes and MixColumn are fully tailored to fit the maximum throughput criteria of our work. Each of the GPU thread, in our work, encrypts on an order of 32 × 128-bit chunks of input data in SIMD manner. This combined with the fact that, in our suggested method, the number of the instructions does not linearly increase with the number of the data elements, will greatly increase the implementation throughput.

4 Proposed Bitsliced Approach

In this section, we introduce our novel bitsliced AES implementation by first, proposing our bitsliced column major representation and explaining how it could be applied to potential applications by showing a simple example. Then, new data representation model for AES and its advantages compared to the previous techniques is presented. Afterwards, we incorporate the introduced representation scheme in the implementation of all the AES stages to show its efficiency.

4.1 Bitsliced SIMD Vectorization

Bitslicing technique initially was introduced for cryptographic implementations to accelerate the procedure of the encryption/decryption for different cryptographic algorithms [29]. Recently bitslicing is utilized to simplify the circuit architecture and reduce hardware cost for quantum microprocessors [36].

Armed with the fact that the costly shift and rotation operations can be reduced to simple swapping in the bitsliced representation, column major representation could increase the utilization of computation units in specific algorithms.

Before getting into the detailed proposed implementation of the AES, in this section, we show that our proposed bitslicing approach could be successfully applied to applications beyond cryptography. To emphasize this, here we investigate a simple example, by applying our bitsliced approach to a Linear Feedback Shift Register (LFSR).

A simple 16-bit LFSR is illustrated in Fig. 4. Every new pseudo-random bit is generated at every clock (MSB of register R). Assuming a 16-bit width architecture (16-bit registers) for a hypothetical circuit, at each clock to produce a new bit, the register needs to be shifted to the right and the Least Significant Bit (LSB i.e., R_0) is to be generated in bit level according to the feedback polynomial function $g\,(x) = x^{15} + x^{13} + x^{12} + x^{10}$. The LSB would be updated at every clock by accessing the R in bit-level granularity as is shown in Equation (1).

$$R_0 = R_{15} \oplus R_{13} \oplus R_{12} \oplus R_{10}. \tag{1}$$

Note that $\oplus$ represents the bit-wise XOR. In this architecture to generate $2^{10} = 1024$ pseudo random bits, the R should be shifted to the right 1024 times and at each clock the LSB should be updated.

This, of course, is the naïve implementation of the LFSR. In the more sophisticated parallel SIMD implementation which is illustrated in Fig. 5, by employing 16 threads and one register each (which are carefully initialized), the number of clock-cycles and shift operations could be reduced to 64 per each register which adds up to 16×64 (i.e., 1024). In this parallel implementation approach, the need for shift and costly bit-level access is still mandatory.

Fig. 6 represents the bitsliced implementation approach to highly vectorized LFSR. In this approach, R^i represents the i^{th} register containing 16 bits from 16 different data chunks in the identical bit position. Changing the representation in this manner (column-major) results in full utilization of the XOR operator as all of the 16 bits of registers are XOR-ed in a single instruction (in the 16-bit architecture). Now, each execution thread operates on 16 bits at each clock. In this way instead of XORing and operating on 16×1-bit registers at each time by 16 threads, just one thread operates on 1×16-bit register, which improves the processor data path utilization.

In our bitsliced approach, the previously stated parallel SIMD scenario could be reduced to 16 registers with no need for costly bit-level operations and bit-level shifting in each thread. Also, by employing our proposed approach the calculation of the LSB at each cycle is reduced to a single register-level operation for all 16 LSB bits (R^0). This results a huge speed-up over the conventional SIMD parallel implementation. By Employing 16 threads, to have a comparison to the previous SIMD scenario, the 1024-bit pseudo-random bits can be easily generated in only 4 clock cycles. Note that in this approach 16 registers are used in a single thread, which accommodates total number of 16×16 registers. As a side note, to generate pseudo-random bits, the registers should be initialized carefully with non-identical and preferably random values (initial states) to prevent the possibility of collision.

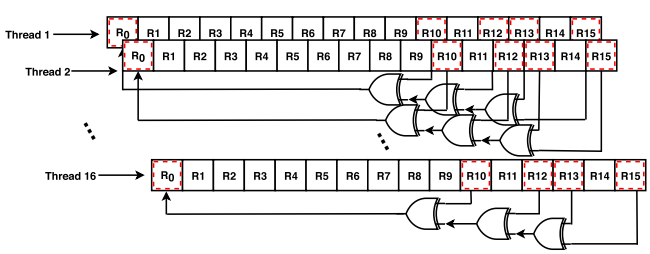

Fig. 5. Parallel LFSR Implementation.

4.2 Data Representation Scheme

Based on the related works on the implementation of AES, emerging GPU technology makes it possible for the bitsliced AES to be accelerated on the massively parallel platform of GPUs. The previous bitsliced methods of AES utilize a data

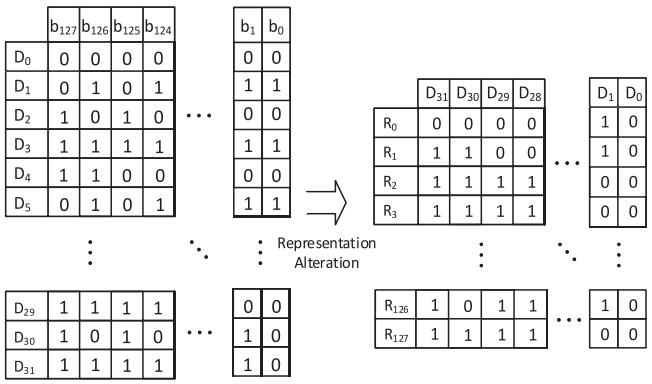

Fig. 7. The change in representation from the conventional row-major (Left) to the column-major representation (Right).

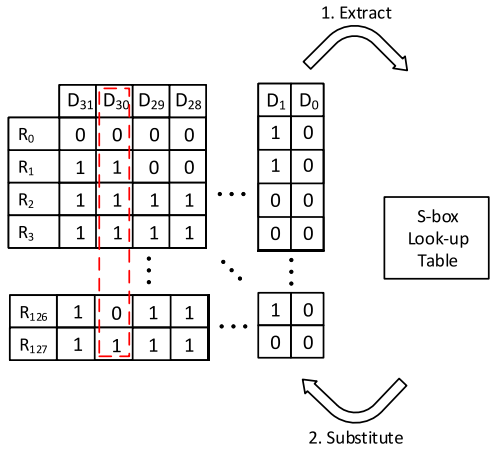

Fig. 8. Using Look-up table-based S-box with the proposed column-major representation.

representation model, which is in need of numerous shift and mask operations in different AES stages. In our suggested technique, by introducing a new representation scheme to store the plaintext or ciphertext data chunks, we strive to eliminate the entire shift and mask instructions in all of the AES stages. By doing so, we reach an unprecedented order of AES encryption and decryption throughput. By altering the representation of data from the conventional row-major to the column-major representation, a number of 32 blocks of 128-bit data are encrypted/decrypted at any instance of time by each parallelization unit. By the row-major representation, we refer to the data representation in which each 128-bit chunk of data is stored in four 32-bit registers. By conventional representation, we are not referring to the representation used in [30], [31]. The conventional representation indicates the way that the data is stored in common programming practices.

We illustrate the effect of the change in the representation on 32 $\times$ 128-bit input data. In the conventional representation, 32 $\times$ 4 $\times$ 32-bit registers are used, this means that each of the thirty-two 128-bit data chunks, is stored in 4 $\times$ 32-bit registers. In the new column-major representation, to store the same input data, 128 $\times$ 32-bit registers are needed. In this implementation, the proposed column-major representation suggests a new way to store the data chunks in a set of registers in which 128 number of 32-bit registers are employed. This way, R_0 instead of holding $b_{0,0}$, $b_{0,1}$, $b_{0,2}$, $b_{0,3}$, $\ldots$, $b_{0,127}$ from the first data ($b_{n,m}$ indicates the m^{th} bit from the n^{th} data), holds $b_{0,0}$, $b_{1,0}$, $\ldots$, $b_{31,0}$, which are the least significant bits from all of the 32 data chunks. Now, the first register having an index of zero stores and saves the least significant bits from all of the 32 chunks of 128-bit data.

In this representation, each register stores bits that have an identical position from all the 32 data chunks. In other words, R_0 stores the least significant bits, R_1 stores the second bits of all the 32 data chunks, this goes on and stays the same as R_{127} stores the 127^{th} bits from all the 32 data chunks. The alteration from the row-major to the column-major representation is illustrated in Fig. 7. The left part of Fig. 7 gives the representation of the data in the row-major format and the set of registers on the right side of the figure indicates the column-major representation.

The benefit of the proposed data representation scheme is that a single GPU thread processes 32 chunks of 128-bit data at each instance of time in SIMD manner. In addition, the storage of all the bits of data in the column-major representation eliminates the need for iterative shift and mask operations involved in different stages of AES. For example, if we want to replace and shift b_0 with b_1, we simply change the reference of R_0 and R_1 registers. By doing so and by knowing that R_0 stores the LSB of all the 32 data chunks, now, the shift operation is performed on all the 32 data chunks in a single instance of time. Moreover, by only changing the reference of the registers through the input arguments of function calls, the need for extra temporary registers is diminished.

4.3 Substitute Bytes

In this section, based on the proposed data representation model, we implement the Substitute Bytes stage of AES to simultaneously operate on 32 chunks of 128-bit data. The original Substitute Bytes stage of the AES takes in an 8-bit segment from the 128-bit data and substitutes that 8-bit segment with a different Byte. First, we illustrate the use of LUTs to undertake the Substitute Bytes stage. Then we will thoroughly show that by using the proposed representation, if we have utilized LUTs to handle the Substitute Bytes stage of the AES, we had to use numerous shift and mask operations to both extract and substitute the data.

In Fig. 8, Substitute Bytes is handled by the utilization of a preprocessed look-up table. For the LUT based implementation, the input data size is equal to 8 bits.

Note that the LUT in this figure takes in and returns 8-bit data. Moreover, the transformation of representation is handled in the same way as is handled in Fig. 7. In this approach to Substitute Bytes:

1. The data needs to be extracted from all of the corresponding registers.
2. The old 8-bit value is replaced with the new value.

These two functionalities require costly shift and mask operations. For example, in Fig. 8, as the S-box look-up table takes in 8-bit inputs each time, the data of D_{30} is read from eight registers from the total of 128 registers for 16 times by the use of several Byte-level shift and mask operations and the new value is written back to the corresponding eight registers each time.

Generally, for all the 32 data chunks, the 8-bit value needs to be extracted and then replaced with its new value. Therefore,

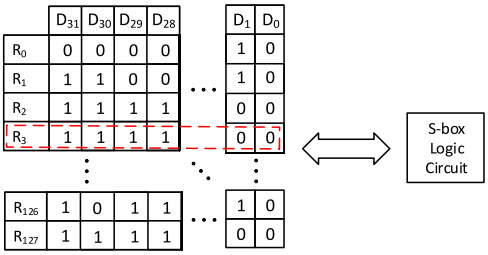

Fig. 9. Using logic circuit-based S-box with the column-major representation.

by using LUTs our proposed representation imposes a heavy cost of the required shift and mask operations on the execution time. Moreover, the use of LUTs is costly due to the use of extensive I/O bound instructions needed for accessing LUTs per each 8-bit segment of the total 128-bit input data. This means that to perform the Substitute Bytes stage on 32 data chunks, the I/O operations increases 32 times as well and also many more shift and mask operations need to be used. Hence, by employing this approach, the SIMD execution feature of our implementation will be undermined, which is because the number of instructions in the code is linearly scaled with the number of presented data elements. As the number of operations increases linearly, there is no gain from the increased number of processed data elements.

To eliminate the cost of the I/O bound operations and the cost of extensive shift and mask operations in the aforementioned extraction and replacement tasks, we seek an alternative approach to implement the Substitute Bytes stage. So, we build upon and modify the compact approach to Substitute Bytes introduced in [17]. This work gives a very compact algorithm for S-box calculation having 253 logic gates. The algorithm given in [17] uses subfields of 4 and 2-bit for the 8-bit S-box calculations.

The implementation of the S-box stage by the use of [17] and our proposed representation is given in Fig. 9. Here, the S-box takes in values of 8-bit data. Unlike the LUT approach, instead of inputting single 8-bit inputs for 16 times per each of the 128-bit data chunks, $R_0, R_1, R_2, \ldots$, and R_{127} are passed into the S-box logic circuit eight registers at a time and the logic gates affect all the 32 data chunks at once. This way, without a linear increase in the number of the instructions corresponding to the number of the data elements, 32 chunks of data are operated on in parallel.

In sum, instead of inputting 8 bits at a time to the S-box function, eight 32-bit registers are passed in and being operated on. This way 32 data chunks are operated on in parallel. It should be mentioned that, now, instead of passing 8 bits of data from the total of a single 128-bit data chunk, eight registers from the total of 128 registers representing 32×128-bit data are passed into the S-box logic circuit. The CUDA implemented of the S-Box with our bitsliced approach is given in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/2911278.

4.4 MixColumn

The MixColumn stage of AES is handled by a matrix multiplication of the state array with the MixColumn matrix. This matrix multiplication is in $GF(2^8)$ and is shown in Fig. 3. In the conventional representation of the state array, each element represents 8 bits from a single 128-bit data. In

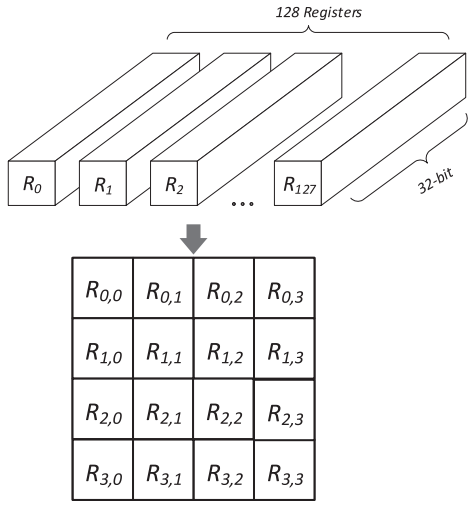

Fig. 10. The organization of all the 128 registers into the register-based state array.

Fig. 3, $S_{0,0}$ represents $b_0, b_1, b_2, \ldots, b_7$. This can be expressed by the following formula. In Equation (2), $S_{n,m}$ indicates the element in m^{th} column and n^{th} row of the state matrix and $index$ equals $32 \times m + 8 \times n$

$$S_{m,n} = b_{index+0}\ ,\ b_{index+1},\ \ldots,\ b_{index+7}. \tag{2}$$

In our implementation, we change the state array with our proposed representation in a way that each element of the state array instead of representing 8 bits from a single data chunk, represents 8 registers accommodating identical bits from 32×128-bit data chunks. By identical bits, we are implying that each register stores and represents bits with identical indices from all the 32 data chunks. Fig. 10 demonstrates the organization of all the 128 registers into the new state array.

Equation (3) gives the relationship between m, n and the batch of corresponding registers comprising each $R_{m,n}$. $index$ is parameterized in the same way as is initialized in Equation (2).

$$R_{m,n} = R_{index+0}\ ,\ R_{index+1},\ \ldots,\ R_{index+7}. \tag{3}$$

What this new state array means is that instead of representing only 8 bits from a single data, each state array element represents eight registers embedding 32 chunks of data in themselves. Now, we will consider the multiplication of the MixColumn matrix with the proposed state array. The new value for $R_{i,j}$ (i.e., $R'_{i,j}$) is calculated as indicated by Equation (4).

$$R'_{i,j} = 2 \times R_{i,j} \oplus 3 \times R_{|i+1|_4,j} \oplus R_{|i+2|_4,j} \oplus R_{|i+3|_4,j} \tag{4}$$

In Equation (4), a number of register shift operations handle the multiplication by 2. Moreover, in our representation, there is no need for Byte-level shift operations as one can substitute the registers with one another to perform the needed shift operations as mentioned in Section 4.2. It is important to notice that the multiplication is in $GF(2^8)$ and in the case of overflowing, the result of the multiplication is modulo by $g\,(x) = x^4 + x^3 + x + 1$ polynomial. When the registers are shifted to the next register, for the purpose of multiplication, the overflowing of the multiplication can be

detected when the MSB of the original 8-bit value is equal to 1. For example, the result of the multiplication of X by 2 is given in Algorithm 1.

Algorithm 1. Multiplication of X by 2 in $GF(2^8)$

Input: $X = x_0\ ,\ x_1,\ \ldots,\ and\ x_7$
Output: $2{\times}X$
{ $Carry\ = x_7$
$x_7 = (x_6)$
$x_6 = (x_5)$
$x_5 = (x_4)$
$x_4 = (x_3) \oplus \text{Carry}$
$x_3 = (x_2) \oplus \text{Carry}$
$x_2 = (x_1)$
$x_1 = (x_0) \oplus \text{Carry}$
$x_0 = \text{Carry}$
}

The multiplication by 3 can be reduced to a multiplication by 2 and an addition (XOR) in $GF(2^8)$. Hence, in our implementation code, Equation (4), which is representing the calculation of one element from the state array in the MixColumn stage, is expanded by Equation (5).

$$\begin{aligned} R'_{i,j} &= 2 \times R_{i,j} \oplus 3 \times R_{|i+1|_4,j} \oplus R_{|i+2|_4,j} \oplus R_{|i+3|_4,j} \\ &= 2 \times (R_{i,j} \oplus R_{|i+1|_4,j}) \oplus R_{|i+1|_4,j} \oplus R_{|i+2|_4,j} \oplus R_{|i+3|_4,j}. \end{aligned} \tag{5}$$

For example, based on Equation (5), $R_{0,0}$ is calculated as follows.

$$\begin{aligned} R'_{0,0} &= 2 \times R_{0,0} \oplus 3 \times R_{1,0} \oplus R_{2,0} \oplus R_{3,0} \\ &= 2 \times (R_{0,0} \oplus R_{1,0}) \oplus R_{1,0} \oplus R_{2,0} \oplus R_{3,0}. \end{aligned}$$

The CUDA implementation of the MixColumn stage is available in Appendix B.

4.5 ShiftRows

The ShiftRows stage of the AES embeds shift and permutation of state array elements. As already indicated, in the conventional representation, each element of the state array accommodates 8 bits from a 128-bit data.

However, in our proposed representation, as mentioned in the MixColumn stage, each element of the state array represents eight registers from the total of 128 registers.

The ShiftRows function performed on the register-based state array is the same as the approach in Fig. 2 (instead of $S_{m,n}$, $R_{m,n}$ should be considered). In this representation, ShiftRows is implemented by register shift and swapping instead of the costly Byte shift and swapping. In the actual implementation of the ShiftRows stage, the register shift and swapping is implicitly expressed in the code by changing the input references of the MixColumn stage. Hence, the ShiftRows stage is embedded into the arrangement of the MixColumn arguments and there is no need to perform any explicit shift operations at any granularity.

4.6 AddRoundKey

In the AES implementation on GPU, all of the round keys for all of the AES rounds are processed once in the host CPU. By processing the value of the round keys once, there is no computational difficulty imposed on the GPU execution. However, storing the preprocessed round keys imposes I/O operations and memory complexity to the execution.

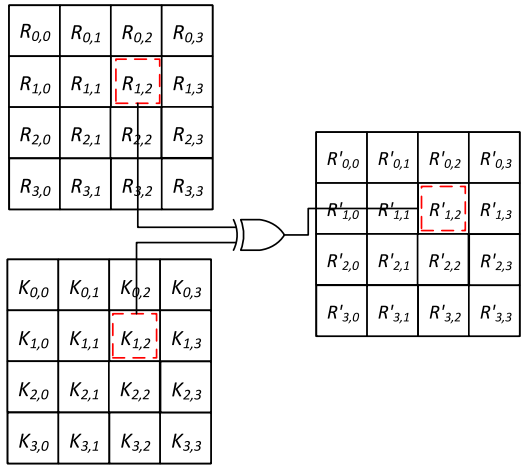

Fig. 11. Register-wise XORing of the register-based state array and the round key state array.

As mentioned before, by using the proposed column-major representation, each GPU thread simultaneously operates on 32 data chunks. Hence, in order to apply the round key to all of the 32 data chunks in parallel, each round key is expanded into 128 registers and then is XORed with the 128 registers from the MixColumn stage. Therefore, in the processing of the round keys, the column-major representation should be adopted in order to represent the value of all the round keys. The value of each of the 128 bits of a single round key should be copied 32 times to fill each register from the total of the 128 registers. As the value of the round key bits are duplicated, each of the 128 round key registers are either 0xFFFFFFFF or 0x00000000.

The state array representation of the 128 registers accommodating a single round key is similar to Fig. 10. In this case, each element instead of being $R_{m,n}$ is $K_{m,n}$, which represents 8 registers from the total of the 128 registers, K_0, K_1, ..., and K_{127}.

Fig. 11 illustrates the result of the XORing operation on the elements of the key state array and the register-based state array from the MixColumn stage output. Here, the elements having identical indices from both of the state arrays are XORed together.

We have employed two different techniques in our AES implementation to store and handle the keys. One way is to store the round keys and to keep the values for all the round keys in the GPU's global memory. Then, the 128 registers representing each round key are moved to the GPU's shared memory to be used. This should be done one round key at a time, which is due to the fact that the size of the generated round keys exhausts the available shared memory space. For example, in the case of AES with n rounds, $n \times 128\times$ 32-bit registers are needed. This number of needed registers easily exceeds the available shared memory space for each thread. By storing the round keys in the global memory and fetching them into the shared memory, at the end of each round, several I/O accesses should be made which hinders the achievable throughput.

TABLE 2
Specification of the Employed GPUs

Name	Single Precision (Gflops)	Double Precision (Gflops)	Memory Bandwidth (GB/s)
GTX 980	4612	144	224
GTX 1050 Ti	1981	62	112
GTX 1080	8228	257	320
GTX 1080 Ti	10609	332	484
Tesla P100	8071	4036	732
Tesla V100	14028	7014	900

In order to diminish the damaging effect of the required I/O accesses in the previous approach and to fine tune our execution performance, we propose a technique in which the round key values are etched into the code with a scripting manner before the compilation of the code. To do so, we propose the use of PyCUDA to generate the unique execution code accommodating round key values from a generic AES code. In this approach, to eliminate the use of "if" statements to select the round key values each time, we have employed logical multiplexer like code with a one-hot encoded selector input. This way there is no divergence present in the code.

5 AES Modes of Operation

AES is a block cipher cryptography algorithm, which takes in a single 128-bit plaintext/ciphertext alongside a key value and returns a 128-bit output. By applying AES encryption to inputs greater than 128 bits in size, a couple of security issues arises. If the same key is used to encrypt each 128-bit segment, in the case in which two or more number of the 128-bit input segments are alike, their encryption process renders same results. This gives an attacker the opportunity to exploit this occurring feature to analyze the plaintext data.

Different operation modes were improvised to apply the block cipher algorithms to the data streams. Four out of the five modes of operation used by AES are immune to the mentioned security risk posed by the aforesaid issue. Here, we do not have the intention to skim over these modes of operation. Other resources have done a great job on that [9], [15]. Actually, what we are concerned with is the parallelization level of these operation modes.

The simplest mode of operation, which is void of any intuition and is actually prone to the previously mentioned security risk, is the electronic codebook (ECB) mode. Other modes such as the cipher block chaining (CBC), cipher feedback (CFB), output feedback (OFB), and counter (CTR) are designed to be perfectly immune to the discussed operation risk on data streams. Among all these five modes of operation, only two, which are the ECB and the CTR modes are capable of being implemented in a parallel manner [9]. The lack of parallelism in other modes, namely CBC, CFB, and OFB are due to the chained encryption scheme improvised in their core design. In all of these modes, each block's encryption relies on the encryption results from its previous blocks. This renders the encryption of 128-bit segments in a data stream sequentially in nature.

In this paper, we have applied the proposed bitsliced representation scheme to all the stages of the AES-128 algorithm. We have used the proposed representation in the implementation of ECB and CTR operation modes of AES algorithm. It is worth noting that in the CTR mode the encryption results from the incremental counter inputs can be processed in advance and kept for the XORing phase with the original plaintext input. However, it is possible, we intend to recompute these intermediate results for different data stream inputs. In the following section, the achievable throughput results from the implementation of the ECB and the CTR modes are given.

6 Evaluation Results

In this section, we will present the evaluation results of the throughput achieved from the execution of the ECB and the CTR modes of operation for AES on a number of CUDA-enabled GPUs. In this study, six Nvidia GPUs are deliberately selected for evaluation purposes. The employed GPUs have different build characteristics such as different single and double precision throughput and memory bandwidth. These features are carefully selected to represent a wide range of execution platforms. First, the range of the selected GPUs completely represents the platforms available to a wide range of users spanning home and enterprise users. Second, these GPUs give us a fair comparison with the previously proposed methods.

6.1 Setup

The systems that we have employed in order to evaluate the proposed methods on many-core platforms are discussed here. The GPUs employed for the evaluation of the proposed solutions include GTX 980, GTX 1050 Ti, GTX 1080, and GTX 1080 Ti, which all are on machines with two Intel XEON E5 2697 V3 CPUs clocked at 2.6 GHz. Each machine has 128 GB of DDR3 RAM. Also, we evaluate our algorithm on Tesla P100 and V100 accelerators on cloud platforms to prove the scalability of our work and also to present a fair compassion with best known work [30]. We used CUDA version 9 in the programming of the GPUs. The specification of all these GPUs is given in Table 2. The employed GPUs are based on earliest state of the art microarchitectures such as Maxwell and Pascal.

6.2 Performance Profiling

By using Nvidia Visual Profiler and Nvidia CUDA Profiling Tool (CUPT), the performance of the proposed implemented algorithm is analyzed in detail. Moreover, hardware cost, and the overall throughput of the implemented sub-functions are determined. Appendix C gives the detailed report for our implementation. This data is extracted by Nvidia Visual Profiler and Nvidia Nsight Compute. Note that the results are for the implementation of AES-ECB encryption on GTX 1050 Ti. It has not escaped our notice that due to the huge data intensive operation in AES shared memory is used to minimize the I/O delay. Our experiments show that the overall resource utilization is quite acceptable for our implementation which results in a huge performance gain.

6.3 ECB Implementation

Here, the performance results from the execution of the ECB mode of operation implemented with our proposed bitslicing

TABLE 3
ECB Encryption Throughput

Name	Throughput (Gbits per sec.)	Performance /GPU cost (Gbits per sec./$)
GTX 980	489.44	1.529
GTX 1050 Ti	296.74	1.484
GTX 1080	805.16	1.008
GTX 1080 Ti	1156.07	0.896
Tesla P100	832.58	0.171
Tesla V100	**1384.51**	0.209

TABLE 4
CTR Encryption Throughput

Name	Throughput (Gbits per sec.)	Performance /GPU cost (Gbits per sec./$)
GTX 980	541.81	1.693
GTX 1050 Ti	324.04	1.660
GTX 1080	843.17	1.053
GTX 1080 Ti	1213.60	0.941
Tesla P100	854.9	0.176
Tesla V100	**1478.42**	0.223

approach are given. In this implementation, we use shared memory to store and fetch the input data for each thread from the global memory. Moreover, each GPU thread processes and operates on 32 numbers of 128-bit data chunks. Table 3 gives the performance achieved by the implementation of the proposed ECB mode on all of the GPUs given in Table 2. The performance per cost metric of each GPU platform is also provided to give a fair evaluation on different GPUs since achievable performance is highly related to the employed platform. This metric is obtained by normalizing the performance value of each GPU platform (by dividing the performance value by its approximate GPU cost). The detailed procedure of the performance per cost calculation is given in Appendix G. Note that the GPU prices might change in the course of time; hence, the values should be updated in case prices are altered.

The peak performance is achieved by the Nvidia Tesla V100 that equals 1.38 Terabits per second of encryption throughput. This is achieved by a GPU grid equal to 524,288 threads. These threads are organized into 8192 blocks each having 64 threads. Here, 2 Gigabits of data that was already stored in the GPU's global memory is encrypted in 0.00144 seconds.

The top achieving bitslicing approach proposed in [30] achieves roughly 606 Gbits of encryption throughput on a Tesla P100 Nvidia GPU. In comparison with this solution, our implementation on the same platform (Tesla P100), improves the AES-ECB by 1.37x. Our proposed bitsliced technique on GTX 1080 Ti is observed to outperform the work from [30] by 7.1x in terms of performance per cost metric, which is a considerable achievement. This achievement in the performance per cost metric, is due to the fact that our implementation could be applied to single precision granularity (32-bit) computations which is the main processing power available in a wide range of cheap and general-purpose GPUs like GTX 1080 and GTX 1080 Ti, but Tesla P100 is known for its powerful double precision computation (64 bit) and is suitable for high-tech scientific applications and R&D purposes.

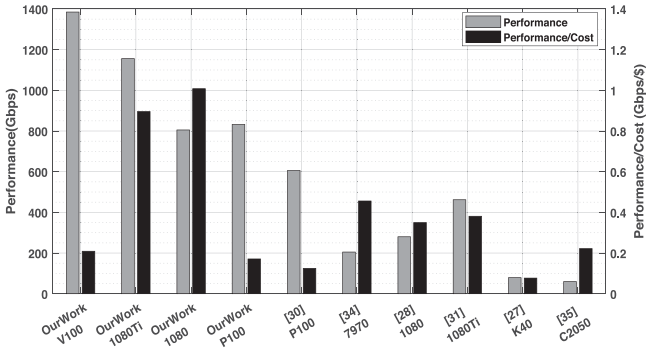

Fig. 12. The AES-ECB Encryption performance and performance/cost achieved by our work and other related efforts.

A compilation of leading previous works proposed on the ECB implementation is given by Fig. 12. The GPUs used by each work are outlined in Table 1. In Fig. 12, We compare our work with the most recently proposed high-performance works given by [30] and [28] on the same reported GPU platforms. Upon our request, authors of [31] provided us with the bitsliced ECB implementation code from [31], which we test on GTX 1080 Ti for a fair comparison. Moreover, due to the lack of access to outdated GPUs employed by others and implementation code from other works, the performance per cost metric is proposed for the sake of evaluation, in Fig. 12, to give a fair comparison with other referenced works. The detailed evaluation of the reported performance per cost metric for these works are given in Appendix G. In Fig. 12, the left vertical axis shows the performance Gbps range and the right vertical axis shows the performance per cost range in terms of Gbps/$. Moreover, our work is also presented by the performance achieved on the GTX 1080Ti and Tesla V100 GPUs which prove to be our best evaluated implementations in terms of performance.

The work from the [34] is benchmarked on a Radeon HD 7970, which is close to the GTX 1050 Ti in the single precision processing power. GTX 1050 Ti outperforms the implementation from [34] by 1.44x. However, the GPU employed in [34] is not based on the CUDA architecture.

The stage involved with the change of the representation is two orders of magnitude faster than the AES encryption phase. Furthermore, the PCI-e connections at hand cannot handle the performance of our AES implementations. However, the available Nvlink 2.0 [37] interconnection speed can handle data communication rate of up to 300 GBytes/s.

6.4 CTR Implementation

In this section, the achievable AES-CTR encryption throughput of all the GPUs given in Table 2 are thoroughly discussed. Here, the counter mode of operation for AES is implemented with the proposed bitslicing approach in our work. In this implementation, each thread operates on and encrypts 32×128-bit data chunks. Here, each GPU grid is comprised of 524,288 threads organized into 8192 blocks each having 64 threads. 32,768 Bytes of shared memory is used by each GPU block accommodating the input data for 64 threads. Table 4 illustrates the throughput achieved by the implementation of the proposed CTR mode.

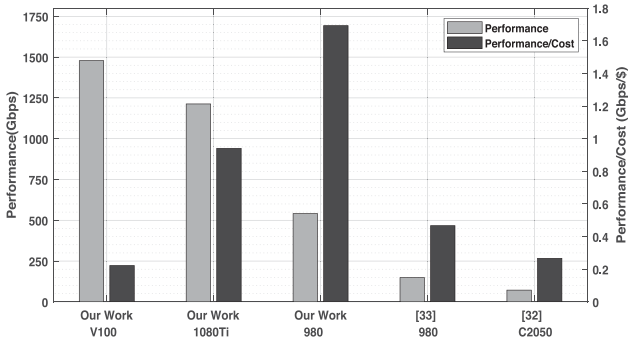

Fig. 13. The AES-CTR Encryption performance and performance/cost achieved by our work and other related works.

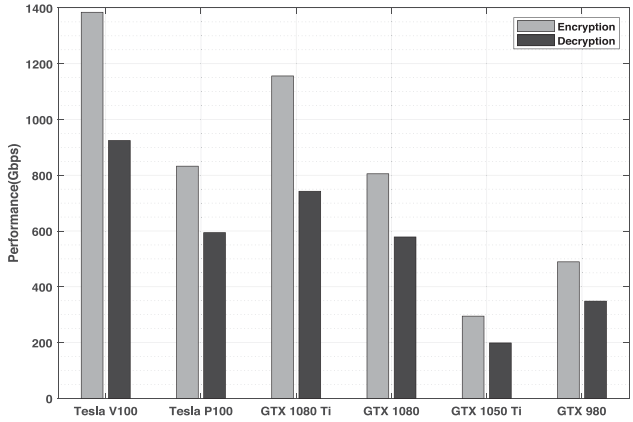

Fig. 14. The AES-ECB Encryption and Decryption comparison.

In the CTR implementation, the peak performance is achieved by the Nvidia Tesla V100 that equals 1.47 Terabits per second of encryption throughput. Here, 2 Gigabits of data is encrypted within 0.001352 seconds.

In our work, unlike other AES implementations on GPU, there is no outstanding difference between CTR and ECB modes of operation throughput.

An illustration for the difference between all of the previous works proposed on the CTR implementation is given in Fig. 13. Again, our works are presented by the performance achieved on GTX 1080 Ti and Tesla V100 GPUs. Also, we implement our algorithm on GTX980 to compare our work to the best known CTR implementation on GPU [33]. Appendix G gives the performance cost normalized metric evaluation details.

In this evaluation, each GPU thread encrypts 32 counter values. Instead of reading from a file or the memory, the counters are generated in the CUDA code by each of the GPU threads. 32 counters are generated from an initial seed equal to the global and the unique position of each thread in the GPU grid. The counters are initially generated in the proposed representation and do not need to be altered from the row-major to the column-major representation. After the encryption, each output should be changed in representation from the column-major back to the row-major representation. Hence, there is no need for the actual input data to be represented in the column-major representation. This way, there is need for only one phase of representation change in the code. Therefore, the AES-CTR code execution achieves higher throughput compared to our AES-ECB implementation.

6.5 Decryption Implementation

The decryption of the AES algorithm in the proposed bitsliced approach, on the both modes of ECB and CTR is implemented based on the same proposed data-representation. The inverse-Sbox and inverse-MixColumn are implemented in the proposed bitsliced manner as the building blocks of the decryption algorithm. However, as could be predicted, due to the fact that the inverse-MixColumn requires much heavier matrix multiplication (in $GF(2^8)$) compared to the MixColumn in encryption, the circuit in the decryption algorithm suffers more complexity compared to the encryption procedure. The detailed proposed implementation of inverse-Sbox and inverse-MixColumn are available in Appendix D and E, respectively. A comparison between decryption and encryption throughput in ECB mode, of the proposed bitslicing approach is given in Fig. 14.

It is worth mentioning that in some applications, the efficiency and throughput of encryption procedure is much more important compared to the decryption phase. For instance, in the satellite communication [38], where high-speed data should be encrypted, then sent or carried by the satellite communication system.

6.6 Multi-GPU

As it is feasible to utilize more than one GPU in a single machine, we here propose a solution that can further scale up the throughput of the AES CTR and ECB execution throughput on a greater number of GPUs within a single standalone machine. Another important factor that empowers us to use multiple numbers of GPUs is the fact that the AES encryption in both of the ECB and CTR modes of operation shows a strong trait of data parallelism.

First, the input data is apportioned amongst all of the available GPUs. In the partitioning phase, if the GPUs are alike in the processing power metric, the input data is equally broken down into same sized partitions. Moreover, if the GPUs are not identical, the input data is apportioned amongst them with regard to their processing power. Then, each GPU goes on by encrypting its appointed share of the input data.

In this approach, one CPU thread is needed to invoke the kernel code on each of the available GPUs. The GPUs can be managed by OpenMP threads in parallel. The rest of the encryption process is handled in the same way as discussed in the single-GPU implementation of the CTR and ECB modes of operation. In our implementation, our evaluation setup is comprised of two GTX 980 GPUs. In the setup in which two identical GPUs are employed, the achieved performance shows a near linear trend, however, it is known that by increasing the number of GPUs to 4 and 8 in this technique, the overall performance doesn't follow a linear increase due to the cost of data scheduling latency, data concatenation, and other parameters [39]. Our experiment with two GTX 980 GPUs resulted in 92 percent performance of theoretical separated 2 GPU (100 percent). The results for two GPU implementation are available in Appendix F.

In the case where the GPUs are not identical, the input data needs to be dynamically or statically apportioned amongst the GPUs. One can dynamically partition the input data by executing sample micro encryption benchmarks on each GPU. Then, by comparing the execution throughput of all of the available GPUs, a precise insight into each GPU's

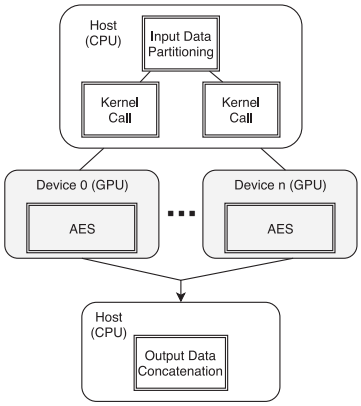

Fig. 15. Multi-GPU execution scheme using multi-GPUs within a single machine.

processing power is gained. Furthermore, one can statically partition the data set by taking into account the processing power reported by the GPU vendors. The execution scheme in this approach is given in Fig. 15. In this figure, first, the input data is divided equally in size by the quantity of available GPU resources. Then, each portion is fed to one of the identical GPUs for encryption.

7 Conclusion

With the emergence of the notion of big data, a strong demand for high-throughput security and encryption mechanisms has risen. In this paper, we propose a novel bit slicing approach for AES cryptography, which achieves high-throughput execution through the introduction of a new data representation scheme. This data representation model alters the representation of the input data from the conventional row-major representation to the proposed column-major representation, illustrated in Fig. 7, Section 4.

Moreover, through the incorporation of the proposed representation scheme into all of the stages of the AES algorithm, we have eliminated the need for the costly Byte-level shift and mask operations extensively used throughout all of the AES functions. We have employed the proposed bit slicing approach in GPU to implement the CTR and ECB modes of operation for AES. Unlike all the previously proposed bit slicing techniques, in our work, each GPU thread simultaneously operates on 32 numbers of 128-bit data chunks.

Our work, on a Tesla V100 achieves 1.47 and 1.38 Terabits per second of encryption throughput for CTR and ECB modes of operation for AES, respectively. The evaluation results on five other GPUs, which are deliberately employed to span a wide range of computing platforms, is given in Section 6. Also, we have considered the cost of platforms to give a fair comparison with other works. Hence, we have incorporated the performance/cost metric into the evaluation section.

In addition, the scalability of our implementation on two GPUs shows close to linear scalability. Our suggested work outperforms the top achieving throughput from [30] by 2x and 11.3x in terms of performance and performance/ cost metrics, respectively. Based on the best of our knowledge, this is the highest achieved encryption throughput. We intend to evaluate our proposed bit slicing technique's side-channel resistance feature in our future work.

Acknowledgments

We are grateful to Prof. Hamid Sarbazi Azad, head of the school of computer science, for his support and useful guidance. We also would like to thank all the other members of HPC lab at IPM Institute. Furthermore, we would like to acknowledge authors of [31] who generously provided their implementation code. Finally, we also wish to sincerely appreciate the efforts put by the anonymous reviewers to pinpoint all the shortcoming of the original manuscript.

References

[1] M. A. Alsmirat, Y. Jararweh, M. Al-Ayyoub, M. A. Shehab, and B. B. Gupta, "Accelerating compute intensive medical imaging segmentation algorithms using hybrid CPU-GPU implementations," *Multimedia Tools Appl.*, vol. 76, no. 3, pp 3537–3555, Feb. 2017.

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, and M. Isard et al., "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pp. 265–283, 2016.

[3] S. Rahmani, A. Ahmadzadeh, O. Hajihassani, S. Mirhosseini, and S. Gorgin, "An efficient multi-core and many-core implementation of k-means clustering," in *Proc. ACM-IEEE Int. Conf. Formal Methods Models Syst. Des.*, 2016, pp. 128–131.

[4] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[5] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," *ACM SIGPLAN Notices*, vol. 51, no. 8, 2016, Art. no. 11.

[6] National Institute of Standards and Technology, FIPS PUB 46-3: Data Encryption Standard (DES), Oct. 1999, supersedes FIPS 46-2. [Online]. Available: http://www.itl.nist.gov/fipspubs/fip186-2.pdf

[7] N.F. Pub, "197: Advanced encryption standard (AES)," *Federal Information Processing Standards Publication*, vol. 197, no. 441, p. 0311, 2001.

[8] B. Schneier, "Description of a new variable-length key, 64-bit block cipher (blowfish)," in *Proc. Int. Workshop Fast Softw. Encryption*, 1993, pp. 191–204.

[9] W. Stallings, *Cryptography and Network Security: Principles and Practice*. London, United Kingdom: Pearson, 2016.

[10] A. Ahmadzadeh, O. Hajihassani, and S. Gorgin, "A high-performance and energy-efficient exhaustive key search approach via GPU on DES-like cryptosystems," *J. Supercomputing*, vol. 74, no. 1, pp. 160–182, 2018.

[11] S. John Walker, "Big data: A revolution that will transform how we live, work, and think," *Int. J. Adv.*, vol. 33, no. 1, pp. 181–183, 2014.

[12] N. Marz and J. Warren, *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Shelter Island, NY, USA: Manning Publications Co., 2015.

[13] M. Chen, S. Mao, and Y. Liu, "Big data: A survey," in *Mobile Netw. Appl.*, vol. 19, no. 2, pp. 171–209, 2014.

[14] N. Khan, I. Yaqoob, I.A.T. Hashem, Z. Inayat, M. Ali, W. Kamaleldin, M. Alam, M. Shiraz, and A. Gani, "Big data: Survey, technologies, opportunities, and challenges," in *Sci. World J.*, vol. 2014, 2014, Art. no. 712826.

[15] M. Dworkin, "Recommendation for block cipher modes of operation methods and techniques," National Inst of Standards and Technology Gaithersburg MD Computer security Div, Tech. Rep., 2001.

[16] D. Kirk, "NVIDIA CUDA Software and GPU parallel computing architecture," in *Proc. 6th Int. Symp. Memory Manage.*, vol. 7, pp. 103–104. 2007.

[17] D. Canright, "A very compact S-box for AES," in *Proc. Int. Workshop Cryptographic Hardware Embedded Syst.*, 2005, pp. 441–455.

[18] J. Daemen and V. Rijmen, "AES proposal: Rijndael," 1999.

[19] A. Satoh, S. Morioka, K. Takano, and S. Munetoh, "A compact rijndael hardware architecture with s-box optimization," in *Int. Conf. Theory Appl. Cryptol. Inf. Security*. 2001, pp. 239–254.

[20] N. Mentens, L. Batina, B. Preneel, and I. Verbauwhede, "A systematic evaluation of compact hardware implementations for the rijndael S-box," in *Proc. Cryptographers' Track RSA Conf.*, 2005, pp. 323–333.

[21] C. Paar, "Efficient VLSI Architectures for Bit-parallel Computation in Galois fields," PhD Thesis, Inst*itute* for Experimental Math., Univ*ersity* of Essen, 1994.
[22] J. Viega, M. Messier, and P. Chandra, *Network Security with OpenSSL: Cryptography for Secure Communications*. Newton, MA, USA: O'Reilly Media, Inc., 2002.
[23] J. Rott, "Intel advanced encryption standard instructions (AES-NI)," Technical Report, Technical Report, Intel, 2010.
[24] D. Kaplan, J. Powell, and T. Woller, "AMD memory encryption," *White paper*, 2016.
[25] S. Ghaznavi, C. Gebotys, and R. Elbaz, "Efficient technique for the FPGA implementation of the AES mixcolumns transformation," in *Proc. Int. Conf. Reconfigurable Comput. FPGAs*, 2009, pp. 219–224.
[26] K. Jang, S. Han, S. Han and S. B. Moon, "SSLShader: Cheap SSL acceleration with commodity processors," in *Proc. 8th USENIX Conf. Netw. Syst. Des. Implementation*, 2011, pp 1–14.
[27] J. Ma, X. Chen, R. Xu, and J. Shi, "Implementation and evaluation of different parallel designs of AES using CUDA," in *Proc. IEEE 2nd Int. Conf. Data Sci. Cyberspace*, 2017, pp. 606–614.
[28] A. A. Abdelrahman, M. M. Fouad, H. Dahshan and A. M. Mousa, "High performance CUDA AES implementation: A quantitative performance analysis approach," in *Proc. Comput. Conf.*, 2017, pp. 1077–1085.
[29] E. Biham, "A fast new DES implementation in software," in *Proc. Int. Workshop Fast Softw. Encryption*, 1997, pp. 260–272.
[30] N. Nishikawa, H. Amano, and K. Iwai, "Implementation of bitsliced AES encryption on CUDA-enabled GPU," in *Proc. Int. Conf. Netw. Syst. Security*, pp. 273–287, 2017.
[31] R. K. Lim, L. Ruth Petzold, and Ç. Kaya Koç, "Bitsliced high-performance AES-ECB on GPUs," in *The New Codebreakers*, Berlin, Germany: Springer, 2016, pp. 125–133.
[32] W. M. N. Zola, and L. C. E. De Bona, "Parallel Speculative Encryption of Multiple AES Contexts on GPUs," in *Proc. IEEE Innovative Parallel Comput.*, 2012, pp. 1–9.
[33] W. K. Lee, H. S. Cheong, R. C. W. Phan, and B. M. Goi, "Fast implementation of block ciphers and PRNGs in maxwell GPU architecture," *Cluster Comput.*, vol. 19, no. 1, pp. 335–347, 2016.
[34] N. Nishikawa, K. Iwai, H. Tanaka, and T. Kurokawa, "Throughput and power efficiency evaluation of block ciphers on kepler and gcn Gpus using micro-benchmark analysis," *IEICE Trans. Inf. Syst.*, vol. 97, no. 6, pp. 1506–1515, 2014.
[35] Q. Li, C. Zhong, K. Zhao, X. Mei, and X. Chu, "Implementation and analysis of AES encryption on GPU," in *Proc. IEEE 14th Int. Conf. High Perform. Comput. Commun., IEEE 9th Int. Conf. Embedded Softw. Syst.*, 2012, pp. 843–848.
[36] G. Tang, K. Takata, M. Tanaka, A. Fujimaki, K. Takagi and N. Takagi, "4-bit bit-slice arithmetic logic unit for 32-bit RSFQ microprocessors," *IEEE Trans. Appl. Superconductivity*, vol. 26, no. 1, pp. 1–6, Jan. 2016.
[37] D. Foley, "Nvlink, pascal and stacked memory: Feeding the appetite for big data," Nvidia.com, 2014.
[38] J. R. Vacca, *Satellite Encryption*. Cambridge, MA, USA: Academic Press Inc, 1998.
[39] L. Chen, O. Villa, S. Krishnamoorthy and G. R. Gao, "Dynamic load balancing on single- and multi-GPU systems," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2010, pp. 1–12.

Omid Hajihassani is working toward the master's degree in the University of Alberta. He is member in the HPC Laboratory at Institute for Research in Fundamental Sciences (IPM), Tehran, Iran. He has conducted high-quality research on porting problems to high-performance solutions, big data, and cloud computing.

Saleh Khalaj Monfared is working toward the bachelor's degree in the K. N. Toosi University of Technology, Tehran, Iran. He is an active member in the HPC Laboratory at Institute for Research in Fundamental Sciences (IPM), Tehran, Iran. His research interests focus on wireless networks and cryptography and high-performance computing.

Seyed Hossien Khasteh received the BSc degree in electrical engineering, the MSc degree in AI, and the PhD degree in artificial intelligence all from the Sharif University of Technology. He is currently an assistant professor with the Computer Engineering Department, K. N. Toosi, University of Technology, Tehran, Iran. His current research interests include social network analysis, machine learning and big data analysis.

Saeid Gorgin received BS and MS degrees in computer engineering from the South branch, and the Science and Research branch, of the Azad University of Tehran. He received the PhD degree in computer system architecture from Shahid Beheshti University, in 2010. He is currently an assistant professor of computer engineering in Department of Information Technology and Intelligent Systems of Iranian Research Organization for Science and Technology (IROST), Tehran, Iran. His research interests include computer arithmetic and VLSI design.