Velocity-Aware Parallel Encryption Algorithm with Low Energy Consumption for Streams

Xiongwei Fei, Kenli Li, *Senior Member, IEEE*, Wangdong Yang, and Keqin Li, *Fellow, IEEE*

Abstract—In the environment of cloud computing, the data produced by massive users form a data stream and need to be protected by encryption for maintaining confidentiality. Traditional serial encryption algorithms are poor in performance and consume more energy without considering the property of streams. Therefore, we propose a velocity-aware parallel encryption algorithm with low energy consumption (LECPAES) for streams in cloud computing. The algorithm parallelizes Advanced Encryption Standard (AES) based on heterogeneous many-core architecture, adopts a sliding window to stabilize burst flows, senses the velocity of streams using the thresholds of the window computed by frequency ratios, and dynamically scales the frequency of Graphics Processing Units (GPUs) to lower down energy consumption. The experiments for streams at different velocities and the comparisons with other related algorithms show that the algorithm can reduce energy consumption, but only slightly increases retransmission rate and slightly decreases throughput. Therefore, LECPAES is an excellent algorithm for fast and energy-saving stream encryption.

Index Terms—Advanced encryption standard, flow control, low energy consumption, parallel encryption, scaling frequency, velocity-aware

1 Introduction

1.1 Motivation

Heterogeneous many-core architecture has the advantages of better performance and energy efficiency compared to homogeneous many-core architecture [1]. This is because the coprocessors in the heterogeneous architecture, such as GPUs, have simple control and high throughput. Therefore, super computers, such as the TOP 500 [2] supercomputers, widely adopt heterogenous architecture currently. Therein the architecture of CPU + GPU becomes the mainstreams thanks to powerful computing capacity, low price, and supporting general computing of GPUs. Despite all of these, for example, Titan, the second place of the TOP 500 computers listed in Nov. 2015, employs Nvidia K20x and still consumes a large amount of 8.208 MW on average. Thus, the problem of lowering energy consumption of GPUs is significant, and then is hotly studied currently.

Cloud computing provides resources by the demands of users with perfect elasticity and scalability, so it is used widely. However, in this open environment, user data are off the control of the owners and more likely to be stolen by adversaries. Encryption is a basic technology to protect user data from stealing. There are massive users in cloud computing. These users produce a large amount of data in the manner of streams. If adopting encryption to protect these data, cloud servers will undertake heavy computation burden.

Cloud servers must efficiently deal with the encryption of data streams; otherwise users experience will be impacted and services will even be disrupted. In order to improve efficiency of encryption, parallel technologies that use many heterogenous cores provide a feasible solution, but with the increase of frequencies and the number of cores, energy consumption increases correspondingly. This causes some problems of heat dissipation, system stability, even environment, etc. Hence, it is emergent to parallelize encryption of data streams in a low energy consumption fashion.

However, in practice, user data streams have the properties of burstiness, variance, realtime, etc. Without considering these properties, parallelism technologies will consume more energy because slow streams will waste energy. For reducing the energy consumption, a feasible solution is first to sense velocity and then to scale the frequencies of GPUs correspondingly. The key point is how to sense the velocity of streams. In this work, we use a sliding window to buffer stream data and sense its used volume in real time.

With the rapid development of electronic commerce and network finance, encryption has become an important measure in protecting secret data of these applications. AES is the symmetric encryption/decryption standard due to its higher security and performance compared to its competitors [3]. AES has several modes, of which the Counter (CTR) mode can be parallelized fully and have the property of provable security [4]. Thus, we choose the CTR mode of AES to encrypt stream data in this work.

In summary, the motivation of this paper is to encrypt stream data of cloud users, especially the data of electronic

- *K. Li and W. Yang are with the College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China. E-mail: {lkl, yangwangdong}@hnu.edu.cn.*
- *X. Fei is with the College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China and the School of Information Science and Engineering, Hunan City University, Yiyang 413000, China. E-mail: feixiongwei@hnu.edu.cn.*
- *K. Li with the College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China and the Department of Computer Science, State University of New York, New Paltz, NY 12561. E-mail: lik@newpaltz.edu.*

Manuscript received 29 Nov. 2016; revised 2 Mar. 2017; accepted 20 Apr. 2017. Date of publication 24 Apr. 2017; date of current version 30 Aug. 2021. Recommended for acceptance by S. Yu, P. Mueller, J. Pei, and K. Liu. Digital Object Identifier no. 10.1109/TBDATA.2017.2697446

commerce or network finance, in low energy consumption and high performance fashion using parallelism and scaling frequencies methodologies. The key points include sensing the velocity of the data streams and scaling the frequencies of GPUs.

1.2 Our Contributions

In this paper, our contributions include four aspects as follows:

1) We adopt dynamic scaling frequency technology to save energy consumption in stream data encryption.
2) We adopt a sliding window to control streams and sense current velocity based on thresholds.
3) We design and implement a velocity-aware parallel encryption algorithm with low energy consumption for streams, called LECPAES.
4) We evaluate LECPAES from energy consumption, throughput, and retransmission rate through comparisons of CPU Serial (CS), CPU Parallel (CP), and GPU Parallel (GP) AES algorithms for streams.

1.3 Organization

The remainder of this paper is organized as follows. Section 2 reviews some related works. Section 3 introduces some models which will be used in this work. Section 4 gives the algorithms. Section 5 introduces the experiments. Section 6 discusses the experimental results. Section 7 provides conclusions and a look to the future.

2 Related Work

Currently, there are two research directions on GPU energy consumption. One is saving the energy consumption, and the other is evaluation and prediction of energy consumption. These two directions will be described in Sections 2.1 and 2.2, respectively. In addition, some works on parallelizing AES are described in Section 2.3.

2.1 Saving Energy Consumption

Some recent research on the first direction are described as follows. Abe et al. [5] analyzed GPU-accelerated systems on power and performance and found that the total energy reduction is trivial using voltage and frequency scaling of CPUs, but can be achieved by scaling voltage and frequency of GPUs.

Ge et al. [6] found that GPU frequencies and energy consumption have a linear relationship in compute-intensive applications. Therefore, scaling down GPU frequencies can reduce energy consumption effectivley.

Ma et al. [7] dynamically distributed workload to CPUs and GPUs based on the previous execution time and dynamically scaled the frequencies to reduce the energy consumption. Arora et al. [8] accurately predicted idle durations and then adopted power gating to save energy consumption of GPUs.

Tang et al. [9] proposed a secret sharing protocol using Elliptic Curve Cryptosystems (ECC) and a proactive share refreshing protocol, which are both efficient and can save energy consumption in communications and processing. Further, they derived a multi-party signature scheme suitable for low-power devices in wireless networks. Liu et al. [10] introduced MoTE-ECC, a highly optimized yet scalable ECC library, which saves energy consumption of nodes through reducing the execution time of two scalar multiplications.

In this work, we adopt the idea of scaling voltage and frequency of GPUs in [5]. Because AES is also a compute-intensive application, which will be analyzed in Section 3.2, the energy consumption of AES can be saved by scaling down GPU frequencies based on [6]. Our work differs from [7] in that our work scales GPU frequencies based on the current velocity of stream.

2.2 Predicting Power and Energy

The second direction is also a research key point because it can provide analyses and insight on how to save energy. Some researchers have studied it based on specific models.

Ma et al. [11] adopted statistical analysis to model the power consumption of Nvidia GPUs but first required to analyze GPU workloads quantitatively. Nagasaka et al. [12] proposed a statistical model which used the GPU performance counters to estimate power consumption of GPUs. The model has high accuracy but is invalid for kernels with texture accesses.

Hong and Kim [13] proposed an analytical model to estimate the execution time of a GPU parallel program. It is useful to understand the bottlenecks of performance in a GPU parallel program and can be used to estimate the energy consumption of the program.

Wu et al. [14] proposed a machine learning model capable of predicting the performance and power of GPUs across a range of hardware configurations. It first gathers a collection of kernels on a real GPU with various configurations, and then estimates the performance and power of new kernels using machine learning with an average error of 15 percent.

Kasichayanula et al. [15] analyzed per-component of power consumption of GPUs such as floating point units, shared memory, and global memory using Nvidia Management Library (NVML) to measure real-time power and energy consumption, and analyzed the power and energy consumption of three kernels on Nvidia Tesla C2075.

In summary, we can get inspiration on energy-saving by means of energy prediction and analyses. For example, we can consider the properties of an application and then correspondingly use computing resources energy efficiently.

2.3 Parallelizing AES

CPU + GPU heterogeneous computers are suitable for the parallel computing on compute-intensive applications, such as the SParse Matrix and Vector multiplication (SpMV) problem [16], [17]. Because AES is a compute-intensive application too (see analysis in Section 3.2) and GPUs support general computing with the merits of low cost and powerful computing, naturally much attention is paid to improve the performance of AES by parallelizing it on the CPU + GPU architecture.

Manavski [18] first parallelized AES on GPUs using Compute Unified Device Architecture (CUDA). His work is significant in two aspects. On one side, the difficulty of programming is reduced; on the other side, the performance of parallel AES is improved about 20 times compared to the serial AES.

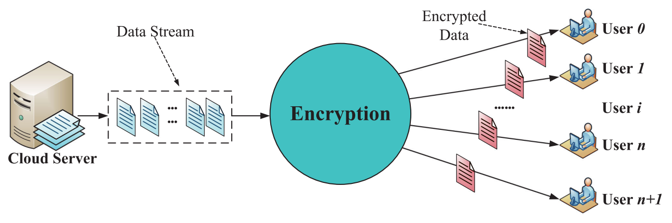


Fig. 1. Scenario of encryption of user data stream in cloud computing.

Later, Maistri et al. [19] parallelized AES adopting CUDA also, and got good ratio of performance to price. Then, Iwai et al. [20] optimized AES from the aspects of parallel granularity and storage distribution, etc., and found that the execution efficiency of parallel AES can be improved by using the granularity of 16 byte/thread and storing T tables in shared memory.

Some other works on optimizing and analyzing parallel AES on GPUs can be found in Refs. [21], [22], [23], [24], [25], [26], [27], [28], [29].

The above works [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29] only deal with single data file, and do not consider the encryption of data stream of users. However, encryption of data stream becomes more and more usual in practice, especially in cloud computing. The data stream produced by massive users have the properties of burstiness, variance, realtime, etc. Therefore, if the data stream is still encrypted in the pattern of single file, it is likely to result in the lack of efficiency and even to cause service failure.

3 Models

3.1 Stream Encryption Model

In cloud computing environment, users' data are outsourced to cloud servers. These data can be processed by or stored in a cloud server. If users want some of their data, these data should be transferred back to users. Because there are massive users in the environment, these data transfers form data streams. Due to the openness of the environment, these data need to be protected by encryption to keep confidentiality. Fig. 1 illustrates this scenario, where massive users request some data from a cloud server. The cloud server responses these requests and will form a user data stream. This stream should be encrypted to guarantee data confidentiality.

In practice, encryption of data stream could face the following three problems:

1) If the average inflow velocity of data stream is faster than the average outflow velocity of encryption, cloud servers will not achieve all the encryption of the data stream. This will cause part failure of services.
2) If the instant inflow velocity of data stream is faster than the outflow velocity of encryption, but the average inflow velocity of the data stream is slower than the outflow velocity of encryption. This can be solved by buffering the burst throughput to stabilize.
3) If the average inflow velocity of data stream is slower than the average outflow velocity of encryption, sometimes cloud servers will be idle but still consume energy. This will cause high energy consumption.

For the first problem, because the velocity of traditional serial encryption is lower than the mainstream velocity of users' data streams, it can be considered to improve the velocity of encryption by parallel encryption using the existing cores of computers. Therefore, in this paper, we propose a solution which uses CPU parallelism or GPU parallelism to improve the efficiency of encryption in heterogenous environment. The details of the solution are described in Section 3.2.

For the second problem, we propose a solution which uses a sliding window to tame the burst data flow of users and achieve robust services. The details of the solution are described in Section 3.3.

For the third problem, we propose a solution, which first senses the velocity of data stream and then dynamically scales the frequencies of GPUs to reduce energy consumption. The details of sensing velocity and saving energy by scaling frequencies dynamically are described in Sections 3.4 and 3.5, respectively.

3.2 AES Parallelizing Model

AES is a block ciper. Each block is long as 16 bytes. The encryption process of a block includes multiple iterations, each of which is called a round. The number of rounds is determined by the version of AES. AES has three versions of 128, 192, and 256. For simplicity, we use AES-N to express the version, where N is one of 128, 192, and 256. A different version of AES needs a key with different length and will perform different number of rounds Nr. Specifically, AES-N needs a key of length N bits and will perform $Nr = N/32 + 6$ rounds.

Fig. 2 describes the encryption process of a block of plain text. Except round $N/32 + 6$ misses a $MixColumn$, each round has four procedures, i.e., $SubBytes$, $ShiftRows$, $MixColumns$, and $AddRoundKey$. $AddRoundKey$ in a different round needs a different round key. Note that before the first round, there is an $AddRoundKey$ also. Therefore, $AddRoundKey$ will be executed $N/32 + 7$ times totally. Correspondingly, $N/32 + 7$ round keys are required.

These round keys are extended from the key of N bits by the procedure $KeyExtension$ as shown in the left part of Fig. 2. The procedure needs to be executed in serial due to data dependence. Nevertheless, once generated, the round keys can be used in other different blocks. This means that $KeyExtension$ needs to be executed only one time in serial in the entire process of encrypting the same plain text.

In a round, $SubBytes$, $ShiftRows$, $MixColumns$, and $AddRoundKey$ involve some operations as listed below respectively (Note: We use the following abbreviations. O: XOR; L: Lookup; R: Rotation; M: Multiplication.).

1) $AddRoundKey$ executes 16 XOR operations ($16O$) on a plain text block, called $state$, and a round key.
2) $SubBytes$ substitutes data using S-$Boxes$ and can be accelerated by 16 T table lookups ($16L$).
3) $ShiftRows$ rotates rows by 0, 1, 2, or 3 byte(s) respectively to confuse data, thus it has 6 byte rotations ($6R$).
4) $MixColumns$ transforms and mixes data in columns which has $4 \times 4 = 16$ multiplications ($16M$) and $3 \times 4 = 12$ XORs ($12O$).

When encrypting a state, AES-N has the number of operations

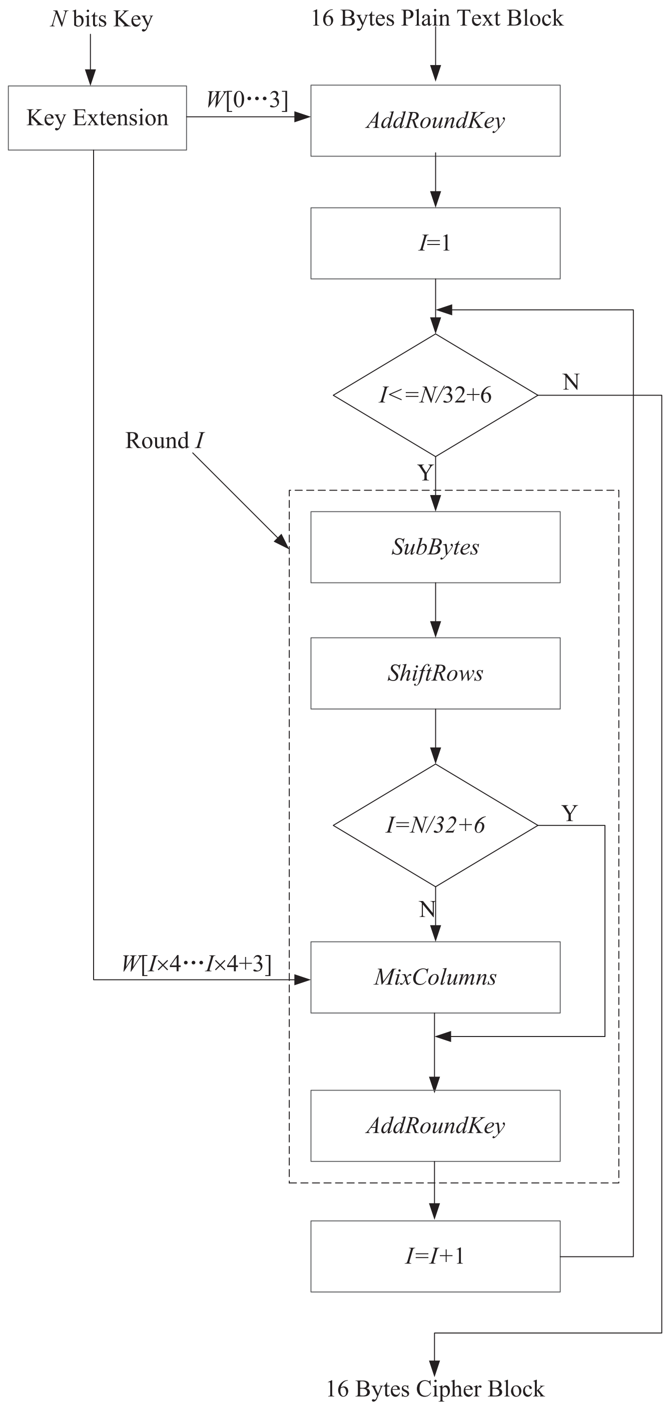


Fig. 2. Encryption process of a block.

$$OPs = Nr \times (OPsb + OPsr) + (Nr + 1) \times OPak + (Nr - 1) \times OPmc, \tag{1}$$

where $OPak$, $OPsb$, $OPsr$, and $OPmc$ represent the number of operations of $AddRoundKey$, $SubBytes$, $ShiftRows$, and $MixColumns$, respectively. For example, AES-256 has operations:

$$\begin{aligned} OPs &= 14 \times (28O + 16L + 6R + 16M) + 16O - 16M - 12O \\ &= 396O + 224L + 84R + 208M, \end{aligned}$$

in encrypting a $state$ long as 16 bytes. If not considering the differences of operations, AES-256 has 912 operations.

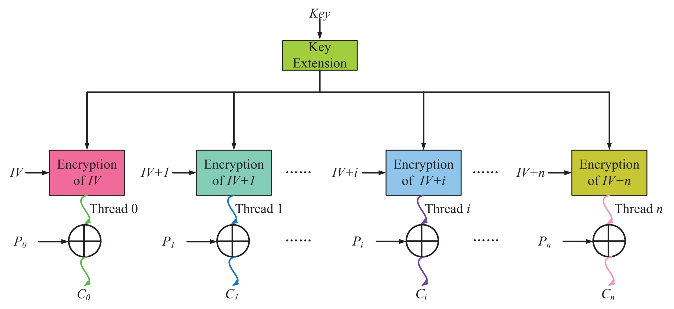


Fig. 3. Encryption parallelizing model of n blocks in CTR mode.

Therefore, AES-256 has the computing complexity of $O(N) = N^{2.46}$, where N represents the number of bytes which want to be encrypted. Similarly, AES-192 and AES-128 have 780 and 648 operations by using and their computing complexities are $N^{2.40}$ and $N^{2.33}$, respectively.

As a conclusion, AES is a computing-intensive application. It is suitable to be executed on GPUs, because GPUs can support these integer operations through Computing Unified Device Architecture (CUDA) and efficiently hide the overhead of transferring data by large-scale parallelism.

AES has five modes, i.e., Electronic Code Book (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB), and Counter (CTR). ECB mode can be fully parallelized, but cannot hide the mode of plaintext and probably suffers active attacks from adversaries. CBC and CFB are hard to suffer active attacks with higher security than ECB but are hard to parallelize. OFB possibly suffers active attacks and hard to parallelize. CTR can be fully parallelized and has proved security, which is strongly recommended by Lipmaa et al. [4]. Naturally in this work, we adopt CTR mode and the encryption process of AES can be parallelized fully among blocks.

Fig. 3 shows the details of parallel encryption in CTR mode, where IV is the initial value of the counter, P_i represents the ith block of plain text, and C_i represents the ith block of cipher text. From the figure, we can see that, after the key extension executed in serial, different values $IV + i$ are encrypted and then added P_i to get C_i in parallel by different threads. It is apparent that both $IV + i$ and P_i are independent in different encryptions.

If the number of blocks of the plain text N_b is more than the number of threads N_t, these blocks will be distributed evenly, i.e., the thread T_i, $0 \le T_i < N_t$, will undertake $N_b/N_t + i < N_b\%N_t$, where / and % represent the operation of integer division and remainder, respectively. For example, plain text with 8 blocks is encrypted by four threads. Each thread will encrypt 2 blocks. For another example, plain text with 10 blocks is encrypted by four threads. Each of the first two threads will encrypt 3 blocks and each of the other threads will encrypt 2 blocks.

3.3 Stream Control Model

With the varying of behaviors of users, user data stream will have different velocity. Sometimes data stream has high velocity; sometimes data stream has low velocity. For burst high-velocity stream, a control mechanism should be adopted to avoid the abruption of services. Therefore, in this paper, we propose a methodology of employing a sliding window to control the burstiness of stream.

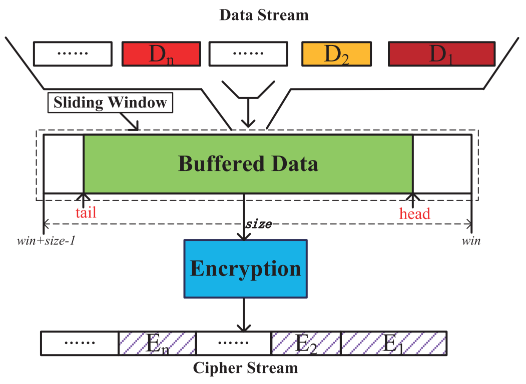


Fig. 4. Stream control model.

Fig. 4 depicts the model of a sliding window, where a user data stream $D_0, D_1, \ldots, D_n, \ldots$ flows into the window and then flows out under the control of the sliding mechanism. The data stream will be encrypted to the corresponding cipher stream $E_0, E_1, \ldots, E_n, \ldots$.

The window corresponds to a chunk of storage space with size of $size$, which can be set by the profile of the cloud server. The green part of the window represents the buffered data, i.e., the data stored in the window currently.

$head$ points to the start byte of the buffered data and $tail$ points to the next byte to the end of the buffered data. As data flow in or out, $head$ and $tail$ will change cyclically.

Apparently, whether user data can flow in the window is determined by the current available size as of the window. How to calculate as? First, the size of the buffered data bs can be calculated using

$$bs = \begin{cases} tail - head & if\ tail \geq head \\ size - head + tail & otherwise. \end{cases} \quad (2)$$

Second, $as = size - bs$. Therefore, as can be calculated using

$$as = \begin{cases} size - tail + head & if\ tail \geq head \\ head - tail & otherwise. \end{cases} \quad (3)$$

Based on the value of as, whether the data of a user D_i with length of $len(D_i)$ can flow in the window $isFi$ is decided by

$$\begin{aligned} isFi = (tail < head \wedge head - tail \geq len(D_i))|| \\ (tail \geq head \wedge size - tail + head \geq len(D_i)), \end{aligned} \quad (4)$$

where "||" is the operation of "or".

For example, assume that a window is with size of 10, i.e., $size = 10$ and it locates at from addresses 20 to 29. Here, we denote the starting address of the window as win, i.e, $win = 20$. Sometime $head$ and $tail$ are 23 and 26, respectively. Now data of a user with length of 6 want to flow in the window. Because $isFi$ is $true$ after the calculation of Eq. (4), these data can flow in the window. Then, the data will be put into addresses 26, 27, 28, 29, 20, and 21. For conciseness, we use $[x, y]$ to denote the space between addresses x and y. In other words, the data will be put into $[26, 29] \cup [20, 21]$, where $\cup$ is the operation of union. And then, $tail$ will be changed to 22.

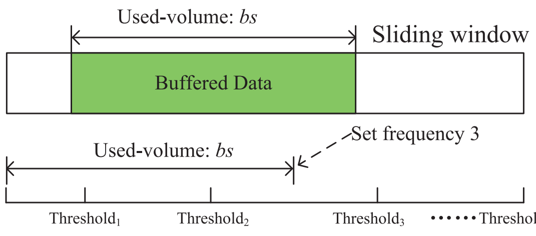


Fig. 5. Sensing velocity model.

Note that once $tail$ goes beyond the next byte to the end of the window, which will restart from the beginning of the window. Formally, if $isFi$ is $ture$ and $tail + len(D_i) > win + size$, the data will be put into two separate parts. One part is from $tail$ to $win + size - 1$ and the other is from win to $tail + len(D_i) - size - 1$. Then, $tail$ will rollback to $tail + len(D_i) - size$. For the above example, because $tail + len(D_i) = 26 + 6 = 32 > win + size = 20 + 10 = 30$, the data will be put into from $tail = 26$ to $win + size - 1 = 29$ and from $win = 20$ to $tail + len(D_i) - size - 1 = 26 + 6 - 10 - 1 = 21$. And then, $tail$ changes to $tail + len(D_i) - size = 26 + 6 - 10 = 22$.

In another aspect, i.e., $isFi == true$ and $tail + len(D_i) < win + size$, the data will be put into from $tail$ to $tail + len(D_i) - 1$ and then $tail$ will change to $tail + len(D_i)$. For the above same window but different $len(D_i) = 3$, the data will be put into $[26, 28]$ and $tail$ will change to 29.

In the last aspect, i.e., $isFi == true$ and $tail + len(D_i) == win + size$, the data will be put into from $tail$ to $tail + len(D_i) - 1$ and then $tail$ will change to win. For the above same window but different $len(D_i) = 4$, the data will be put into $[26, 29]$ and $tail$ will change to 20.

In summary, if data of a user can flow in the window, the storage space s and $tail$ can be calculated by

$$s = \begin{cases} [tail, tail + len(D_i) - 1] \cup [win, tail + len(D_i) - size - 1] \\ \qquad if\ isFi = true \wedge tail + len(D_i) > win + size; \\ [tail, tail + len(D_i) - 1] \\ \qquad if\ isFi = true \wedge tail + len(D_i) \leq win + size, \end{cases} \quad (5)$$

and

$$tail = \begin{cases} tail + len(D_i) - size & if\ isFi = true \wedge tail + len(D_i) > win + size; \\ tail + len(D_i) & if\ isFi = true \wedge tail + len(D_i) < win + size; \\ win & if\ isFi = true \wedge tail + len(D_i) = win + size, \end{cases} \quad (6)$$

respectively.

3.4 Sensing Velocity Model

The sliding window can be used not only to tame a burst data stream, but also to sense the current velocity of the stream. We propose a methodology of first sensing the current bs of the window and then determining the level of the velocity by comparing to some pre-set thresholds. Because the realtime velocity changes constantly, it is hard to sense the velocity precisely. But we can use bs of the window to approximate the velocity. Larger bs in the window means higher velocity of the data stream, and vice versa.

Fig. 5 illustrates an example of sensing velocity. At a given time, the sliding window has buffered bs data, where

TABLE 1
Characteristics of K20M

#	fc (MHz)	fm (MHz)	tr	condition
1	324	324	0.43	$bs < 0.43 \times size$
2	614	2,600	0.81	$0.43 \times size \leq bs < 0.81 \times size$
3	640	2,600	0.84	$0.81 \times size \leq bs < 0.84 \times size$
4	666	2,600	0.88	$0.84 \times size \leq bs < 0.88 \times size$
5	705	2,600	0.93	$0.88 \times size \leq bs < 0.93 \times size$
6	758	2,600	NA	$bs \geq 0.93 \times size$

bs can be calculated by Eq. (2). Then *bs* compares with the thresholds. If the value of *bs* locates in the middle of two thresholds, i.e., $threshold_i \leq bs < threshold_{i+1}$, then processors will scale to the corresponding frequency f_{i+1}.

As shown in Fig. 5, *bs* of the buffer data locates in the middle of $threshold_2$ and $threshold_3$, so the processor will scale its frequency to f_3. Generally, assume that a GPU can work at n fixed frequencies $F = \{f_1, \ldots, f_n\}$ and $n-1$ thresholds can be set up to determine which one frequency f should be chosen. Eq. (7) describes the calculation method in detail

$$f = \begin{cases} f_1, \quad if\ bs < threshold_1 \\ f_{i+1}, if\ threshold_i \leq bs < threshold_{i+1} \land i \in \{1, \ldots, n-2\} \\ f_n, \quad if\ bs \geq threshold_{n-1}. \end{cases} \tag{7}$$

With the change of velocity, *bs* will change and result in the change of frequency dynamically. By this way, the energy consumption will reduce. The rational will be described in next Section 3.5. The *bs* of sliding window can affect the current GPU frequency. Because a GPU must work at some fixed frequencies $F = \{f_1, \ldots, f_n\}$ and a certain frequency f_i represents a certain computation speed, a relationship can be established as

$$\frac{f_x}{f_y} \approx \frac{threshold_x}{threshold_y}, \tag{8}$$

where f_x and f_y represent two different fixed frequencies, respectively; whereas $threshold_x$ and $threshold_y$ represent two different thresholds, respectively. If knowing a fixed frequency f_x and the corresponding $threshold_x$, for any f_y in the fixed frequency set, the corresponding $threshold_y$ can be figured out. Through this ratio relationship, a threshold can be roughly determined by another.

A GPU has two sorts of different frequencies, one of which is core frequency fc, and the other is memory frequency fm. For example, K20M has six pairs of frequencies as listed in Table 1.

When calculating threshold ratio tr, we should consider these two sorts of frequencies together, but AES is a compute-intensive application as analysed in Section 3.2. Therefore, we can grasp the main contradiction, i.e., we can consider only the core frequency. Therefore we can get five different threshold ratios tr_i using Eq. (9) on K20M:

$$tr_i = \frac{fc_i}{fc_6}, i = 1, \ldots, 5. \tag{9}$$

These tr_i are calculated and listed in the fourth column of Table 1. If the sliding widow is large as $size$, then we can get the conditions for setting different frequency according to current *bs*. For K20M, these conditions are listed in the fifth column of Table 1. Specifically, because $\frac{fc_1}{fc_6} = \frac{324}{758} = 0.43$, when $bs < 0.43 \times size$, GPU core frequency and memory frequency are set to fc_1 and fm_1, respectively.

3.5 Saving Energy Model

The data stream produced by massive users has the properties of burstiness, variance, realtime, etc. These properties will bring the possibility of reducing the energy consumption of streaming encryption. When the velocity of the data stream becomes slow, scaling down the frequency of processors appropriately will reduce energy consumption E. The rationale will be discussed as follows.

The power of a processor P is consist of static power *Ps* and dynamic power *Pd*, i.e., $P = Ps + Pd$. *Ps* is determined by some factors, such as circuit technology, chip layout, etc. *Ps* is not impacted by the frequency of the processor. Whereas, *Pd* is determined by the power formula [30] of CMOS (Complementary Metal Oxide Semiconductor) circuits shown in

$$Pd = ACV^2 f, \tag{10}$$

where A is the switching activity factor, C is the capacitance, V is the supply voltage, and f is the clock frequency. Furthermore, V and f have the relationship as shown in

$$f = K\frac{(V - V_T)^\gamma}{V}, (1 \leq \gamma \leq 2), \tag{11}$$

where V_T is the threshold voltage while K and γ are the parameters related to manufacturing technique.

Because V_T can be omitted because its value is usually very small compared to V [31]. An approximate relation formula is yielded as

$$f \approx KV^{\gamma-1}. \tag{12}$$

Then,

$$Pd = ACkf^\alpha,\ \alpha = \frac{\gamma+1}{\gamma-1}, k = K^{\frac{2}{1-\gamma}}, \tag{13}$$

can be yielded from Eqs. (10) and (12). Hence, *Pd* is proportional to α power of f, where $\alpha = \frac{\gamma+1}{\gamma-1}$.

By default, GPUs are set at the second highest frequencies to keep high performance and good stability. After sensing velocity of sliding window, energy can be saved by scaling the frequency of GPUs corresponding to the velocity of data stream.

Fig. 6 demonstrates the rational. In the top of the figure, a data stream with variant velocity flows in a sliding window, and then it flows out and is shaped to a fixed velocity.

When the stream is encrypted, two different solutions are shown in the figure. One is non-scaling frequency as shown in the part pointed by the arrow marked *b*. The other is scaling frequency as shown in the part pointed by the arrow marked *a*.

The non-scaling frequency encrypts the data stream in the fixed frequency and consumes energy unchangeably. The scaling frequency encrypts the data stream at different frequencies according to the velocity. Scaling down the

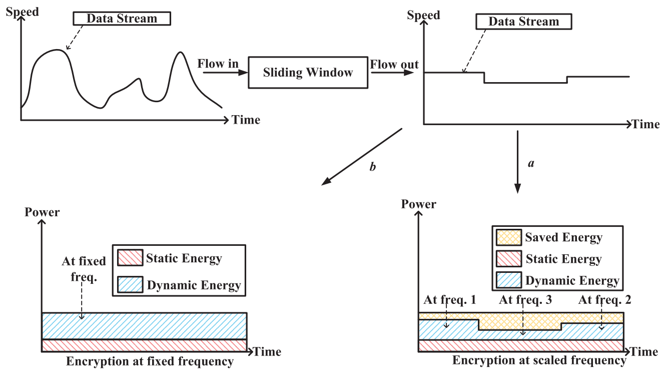


Fig. 6. Comparison of energy consumption between scaled freq. and fixed freq.

frequency will cause low energy consumption according to Eq. (13).

In addition, because the default frequency is the second highest, the non-scaling frequency will consume more energy than scaling frequency. The reddish slash grid represents the static energy consumption, which is the same in the two solutions. The blue back slash grid represents the dynamic energy consumption. Therefore, the yellow crossover grid represents the saved energy.

4 Algorithms

In this section, a Low-Energy Consumption Parallel AES (LECPAES) algorithm is proposed based on the above sections. As comparisons, GPU Parallel, CPU Parallel, and CPU Serial (CS) AES are described in this section also.

4.1 LECPAES

Algorithm 1 describes the details of LECPAES in the pseudo code manner. The algorithm can encrypt users' plain text streams to cipher text streams.

Nvidia Management Library (*nvml*) [32] provides the abilities of setting frequencies of GPUs and retrieving the energy consumption. Therefore, in Line 1, the algorithm initializes *nvml*. If successful, then the algorithm gets the GPU device handle in Line 2. Because a GPU must work at some fixed frequencies, the algorithm retrieves all supported frequencies in Line 3. Based on these frequencies, threshold ratios can be calculated as Eq. (9) in Line 4. In Line 5, the algorithm establishes a sliding window with size of *size* and sets the initialize *head* and *tail* for buffering data and sensing velocity.

Because this algorithm needs to process users' data stream, from Lines 6 to 12, a "while" loop will be performed continuously until a stop command is issued. In Line 7, the algorithm selects a frequency combination f based on the current used ratio of the sliding window using the method in Eq. (7).

In Line 8, the algorithm uses *nvmlDeviceSetApplicationsClocks* in *nvml* to set GPU frequency to f. After that, the current *plainStream* in the window is encrypted by parallel AES on the GPU in Line 9. And then, the *cipherStream* can be flowed out in Line 10.

Because the state of window changes, *head* and *tail* are moved as described in Section 3.3 in Line 11. If *stop* command is issued, the loop will terminate. And then *nvml* should be shutdown as shown in Line 13.

Algorithm 1. LECPAES Algorithm

```
Input:
  Plain text stream, $plainStream$;
  Keys, $keys$;
Output:
  Cipher text stream, $cipherStream$;
1: Initialize $nvml$;
2: Get GPU device handle;
3: Retrieve supported $n$ kinds of frequencies $fc$ and $fm$;
4: Compute threshold ratios using Eq. (9);
5: Initialize a sliding window with size $size$;
6: while (!stop) do
7:    Select frequency $f$ by Eq. (7);
8:    Set GPU frequency to $f$;
9:    Encrypt $plainStream$ in current window using $keys$ on
      GPU in parallel;
10:   Flow out the current encrypted cipher $cipherStream$;
11:   Move $head$ and $tail$ as the description in Section 3.3;
12: end while
13: Shutdown $nvml$.
```

4.2 GP

Algorithm 2 shows the pseudo code of GP. The algorithm is different from LECPAES in scaling GPU frequencies. GP does not need to scale GPU frequencies, so that it does not use *nvml* and does not sense the velocity of the current stream.

Algorithm 2. GP Algorithm

```
Input:
  Plain text stream, $plainStream$;
  Keys, $keys$;
Output:
  Cipher text stream, $cipherStream$;
1: Initialize sliding window with size $size$;
2: while (!stop) do
3:    Encrypt $plainStream$ in current window using $keys$ on the
      GPU in parallel;
4:    Flow out the current encrypted cipher $cipherStream$;
5:    Move $head$ and $tail$ as the description in Section 3.3;
6: end while
```

GP first initializes a sliding window in Line 1, and then circularly executes encryption task until a *stop* command is issued. The encryption task includes encrypting the users' data in current window on the GPU in parallel as shown in Line 3, flowing out the cipher stream as shown in Line 4, and sliding the window as shown in Line 5.

4.3 CP

Algorithm 3 exhibits the pseudo code of CP. The algorithm differs GP on parallelism method. CP employs CPU parallelism rather than GPU parallelism adopted in GP. Therefore, CP derives multiple threads according to the number of CPU cores in Line 3. If the CPU supports hyper-threading, CP will derive twice as many of threads of the number of CPU cores. For example, Two Intel Xeon E5-2640 v2 CPUs support hyper-threading and have 16 cores totally, so that CP derives $\alpha = 16 \times 2 = 32$ threads when running on these CPUs. The derived threads will undertake the encryption evenly in parallel as shown in Line 4. After the encryption finished, the current encrypted cipher $cipherStream$ flows out. And then $head$ and $tail$ are moved as the description in Section 3.3.

Algorithm 3. CP Algorithm

```
Input:
  Plain text stream, $plainStream$;
  Keys, $keys$;
Output:
  Cipher text stream, $cipherStream$;
1: Initialize sliding window with size $size$;
2: while (!stop) do
3:     Derive $\alpha$ threads according to the numbers of CPU cores;
4:     Perform encryption by each thread in parallel for even
       users' data;
5:     Flow out the current encrypted cipher $cipherStream$;
6:     Move $head$ and $tail$ as the description in Section 3.3;
7: end while
```

4.4 CS

Algorithm 4 describes the pseudo code of CS, which differs CP on whether adopting parallelism. CS does not adopt CPU parallelism and serially performs encryption for the buffered data in the window as shown in Line 3.

Algorithm 4. CS Algorithm

```
Input:
  Plain text stream, $plainStream$;
  Keys, $keys$;
Output:
  Cipher text stream, $cipherStream$;
1: Initialize sliding window with size $size$;
2: while (!stop) do
3:     Perform encryption serially for the buffered users' data;
4:     Flow out the current encrypted cipher $cipherStream$;
5:     Move $head$ and $tail$ as the description in Section 3.3;
6: end while
```

5 Experiments

5.1 Experiment Setup

5.1.1 Experiment Environment

Experiments are conducted on a heterogeneous platform with two Intel Xeon CPUs and one Nvidia K20M GPU. The configuration of the platform is listed in Table 2.

Experiments are conducted with four different algorithms for a series of user data streams. The purpose is to compare them and validate the effect of LECPAES, and further find some useful conclusions. The four algorithms are CPU Serial AES, CPU Parallel AES, GPU Parallel AES, and LECPAES. They all employ sliding windows but differentiate in encryption manner. CS executes serial encryption for the buffered data one user by one user on the CPUs, whereas CP executes parallel encryption for the current buffered data in the window on the CPUs. GP executes parallel encryption for the current buffered data in the window on the GPU. LECPAES differs GP in dynamically scaling the frequency to reduce the energy consumption.

TABLE 2
Configuration of the Experimental Platform

GPU	CPU
NVIDIA Tesla K20M (13 multiprocessors)	Two Intel Xeon E5-2,640 v2 (support hyper-threading)
Six groups of supported clock frequencies of core and memory in unit of MHz: (324, 324), (614, 2,600), (640, 2,600), (666, 2,600), (705, 2,600), (758, 2,600)	Clock rate: 2.0 GHz
Total: 2,496 cores	Total: 16 cores

The user data stream is mimicked as 12,000 users submitting their data cyclically. The period, called $latency$, represents different flow velocity. Longer $latency$ means lower stream velocity, and vice versa. Thus, the experiments can implement different streams with different velocity. The users will produce data with length between 35 KB and 150 KB to simulate the data in typical web accesses randomly, since this length range has been gathered statistics by Levering et al. [33].

When the velocity of the stream changes, sometimes the velocity of inflow may be faster than the velocity of outflow. If the sliding window has not enough space to load more data, this will make some data miss, i.e., cannot be encrypted this time. In this situation, these missed data will be retransmitted later. In the experiments, the time for retransmission is set to $2 \times latency$.

In order to get the energy consumption of the GPU, the experiments use *nvmlDeviceGetPowerUsage* in *nvml* to get the instant power *gpower* every 50 ns and *Event* mechanism in CUDA to record the spent time *gtime*.

In another aspect, the experiments use *PowerGadget* provided by Intel to sample the instant power of the CPUs *cpower*. *PowerGadget* bases on RAPL (Running Average Power Limit) library which can acquire the data of power of Intel CPUs recorded in MSRs (energy Model-Specific Registers). The spent time on the CPUs *ctime* are recorded by $gettimeofday$ for Linux or $QueryPerformanceCounter$ for Windows.

Further, the energy consumption Eg on the GPU can be calculated by $Eg = gpower \times gtime$. Similarly, the energy consumption Ec on the CPUs can be calculated by $Ec = cpower \times ctime$. Therefore, the total energy consumption on the CPUs and GPU E can be calculated as: $E = Eg + Ec$.

5.1.2 Experimental Process

The experiments first derives five threads to execute different tasks. The first thread produces the plain text data for users every *latency* to form a data stream. If the sliding window has enough space, the thread will put the data into the window and move $tail$ and then continue; otherwise the thread will wait $2 \times latency$ and then continue.

The second thread is in charge of triggering the encryption task by one of CS, CP, GP, or LECPAES according to different experiments. Once the encryption completes for the buffered data, the thread will move *head* and then continue.

The next two threads acquire the data of power on the CPUs and GPU every 50 ns, respectively. The last thread

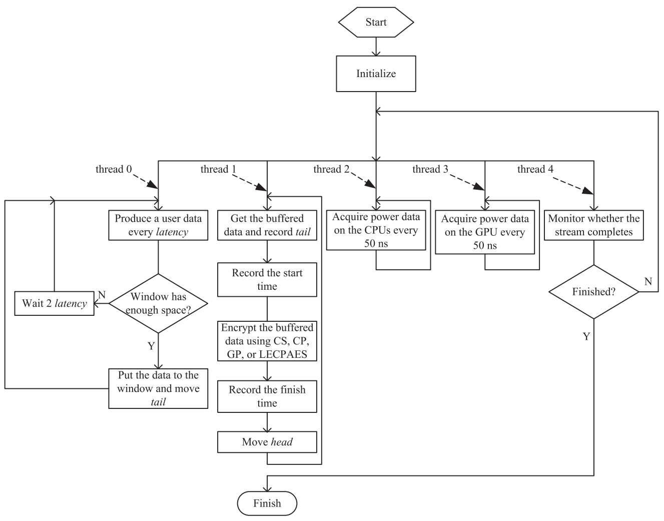

Start
Initialize
thread 0
thread 1
thread 2
thread 3
thread 4
Produce a user data every latency
Get the buffered data and record tail
Acquire power data on the CPUs every 50 ns
Acquire power data on the GPU every 50 ns
Monitor whether the stream completes
Record the start time
Window has enough space?
Wait 2 latency
Encrypt the buffered data using CS, CP, GP, or LECPAES
Finished?
Put the data to the window and move tail
Record the finish time
Move head
Finish

Fig. 7. Experimental process.

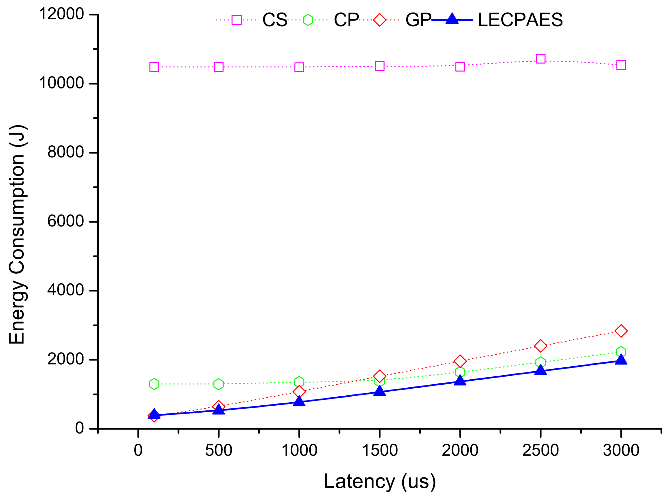


Fig. 8. Energy consumption.

terminates the application when the data stream has been processed completely. The full process flow can be seen in Fig. 7.

The experiments are conducted for the four algorithms separately. For implementing different velocities, the experiments choose seven different latencies, i.e., 100, 500, 1,000, 1,500, 2,000, 2,500, and 3,000 us. The results will be reported in the next Section.

5.2 Experimental Results

The experimental results will be reported from several aspects including energy consumption, throughput, retransmission rate, and energy saving, in this section.

5.2.1 Energy Consumption

In the experiments, CS, CP, GP, and LECPAES encrypt some users' data streams with different *latency* separately. Their energy consumptions are drawn in Fig. 8.

From the figure, CS consumes far more energy than the other three algorithms. Moreover, CS consumes also the same energy for the streams with different *latency*. The average energy consumption of CS is $10{,}520.77$ J.

CP, GP, and LECPAES increase energy consumption with the increase of latency. Specifically, CP increases energy consumption from $1{,}259.91$ J for the stream with latency of 100 us to $2{,}218.47$ J for the stream with latency of $3{,}000$ us. GP increases energy consumption from 381.31 J for the stream with latency of 100 us to $2{,}839.52$ J for the stream with latency of $3{,}000$ us. LECPAES increases energy consumption from 395.95 J for the stream with latency of 100 us to $1{,}974.57$ J for the stream with latency of $3{,}000$ us.

LECPAES consumes less energy than GP for the streams with latency larger than 500 us, but more energy than GP for the streams with latency less than 500 us. For the stream with latency of 500 us, GP and LECPAES consume energy of 641.60 J and 524.89 J, respectively.

GP consumes less energy than CP for the streams with latency less than $1{,}500$ us, but more energy than CP for the streams with latency larger than or equal to $1{,}500$ us. For the stream with latency of $1{,}500$ us, GP and CP consume energy of $1{,}522.14$ J and $1{,}378.80$ J, respectively.

LECPAES consumes less energy than CP for all the streams in the figure, but their difference of energy consumption becomes less with the increase of latency. For the stream with latency of 100 us, LECPAES and CP consume energy of 395.95 J and $1{,}259.91$ J, respectively. Here, LECPAES consumes 899.96 J less energy than CP. Nevertheless, for the stream with latency of $3{,}000$ us, LECPAES and CP consume energy of $1{,}974.57$ J and $2{,}218.47$ J, respectively. Now, LECPAES consumes 243.90 J less energy than CP.

5.2.2 Throughput

In this Section, throughput will be reported. Here, throughput represents the efficiency of a stream passing the encryption server. It is defined as a quotient *throughput* of the data volume of the stream dv dividing the pass time pt, i.e., $throughput = \frac{dv}{pt}$. Note that dv refers to the available data volume, i.e., the same data retransmitted multiple times will be calculated only one time. In addition, pt represents the entire time of from the time of the stream flows in to the time of the stream flows out.

Fig. 9 portrays the throughput of the four algorithms for the streams with different latencies. From Fig. 9, the throughput of LECPAES, GP, and CP decreases as the

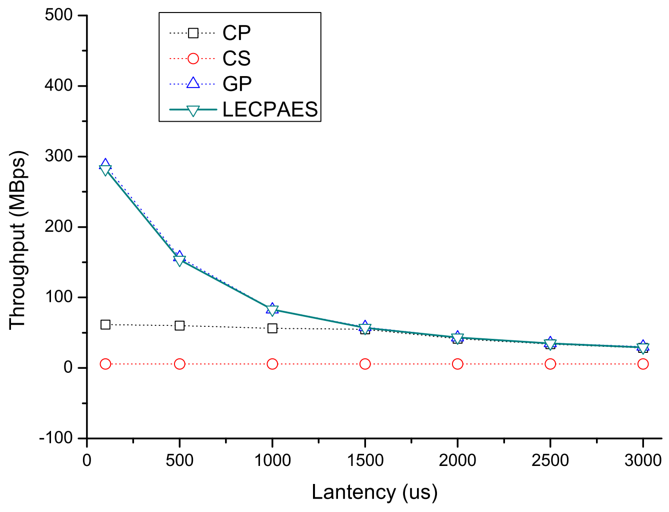


Fig. 9. Throughput.

TABLE 3
Retransmission Rate

latency (us)	CS	CP	GP	LECPAES
100	61.13075	4.60500	0.64567	0.73242
500	14.30458	0.73592	0.01425	0.01492
1,000	6.54617	0.40750	0.00650	0.00667
1,500	4.50083	0.25417	0.00358	0.00458
2,000	3.27467	0.18333	0.00133	0.00258
2,500	2.62733	0.08543	0.00000	0.00000
3,000	2.02175	0.03786	0.00000	0.00000

latency increases, but the throughput of CS keeps stable. Specifically, LECPAES decreases throughput from 282.13 MBps for the stream with latency 100 us to 29.32 MBps for the stream with latency $3{,}000$ us. GP decreases throughput from 287.56 MBps for the stream with latency 100 us to 30.31 MBps for the stream with latency $3{,}000$ us. CP decreases throughput from 61.44 MBps for the stream with latency 100 us to 28.45 MBps for the stream with latency $3{,}000$ us. CS keeps the throughput between 5.55 MBps and 5.62 MBps.

In the experiments, GP has the highest throughput, but LECPAES is only a little less than GP. CS has the lowest throughput. The average throughput of CS, CP, GP, and LECPAES are 5.57 MBps, 48.02 MBps, 99.18 MBps, and 97.59 MBps, respectively.

5.2.3 Retransmission Rate

When a stream has higher velocity of inflow than that of outflow, some users' data can be buffered in the sliding window. However, when the sliding window does not have enough space to load more data, current users' data will be lost and need to be retransmitted.

Retransmission will take place after $2 \times latency$ of the missing service. Retransmission will reduce the QoS (Quality of Service) and impact the experience of users. Retransmission rate is defined as the ratio of retransmission to all transmission and reflects the QoS.

Table 3 lists the retransmission rate processed by different algorithms for the streams with different latencies. The retransmission rates decrease as the latencies decrease. CS has the highest retransmission rate, and GP has the lowest retransmission rate.

5.2.4 Energy Saving

Energy saving ratio Esr_{ij} represents the saving energy ratio of one algorithm A_i over another one A_j in this paper. It can be calculated as $Esr_{ij} = (EA_j - EA_i)/EA_j$, where EA_i and EA_j are the energy consumption of A_i and A_j, respectivley. Table 4 lists the energy saving ratio of LECPAES over the other three algorithms. The second, third, and fourth columns represent the ratios of LECPAES over GP, CP, and CS, respectively. The last row represents the average ratios.

TABLE 4
Energy Saving Ratio of LECPAES

latency	L2G	L2CP	L2CS
100	−0.03842	0.69446	0.96222
500	0.18190	0.59456	0.94992
1,000	0.28982	0.43694	0.92702
1,500	0.29878	0.22588	0.89846
2,000	0.30019	0.16544	0.86940
2,500	0.30272	0.13169	0.84399
3,000	0.30461	0.10994	0.81167
avg	0.23423	0.33699	0.89467

TABLE 5
Energy Saving Ratio of GP and CP

latency	G2CP	G2CS	CP2CS
100	0.70576	0.96362	0.87636
500	0.50442	0.93878	0.87647
1,000	0.20716	0.89723	0.87038
1,500	−0.10396	0.85519	0.86883
2,000	−0.19255	0.81339	0.84352
2,500	−0.24528	0.77626	0.82033
3,000	−0.27995	0.72917	0.78840
avg	0.08509	0.85338	0.84918

The energy saving ratio of LECPAES over GP (L2G) increases as the latency increases. Its average value is 23.4 percent. The energy saving ratio of LECPAES over CP (L2CP) decreases as the latency increases. Its average value is 33.7 percent. The energy saving ratio of LECPAES over CS (L2CS) decreases as the latency increases. Its average value is 89.5 percent.

Table 5 lists the energy saving ratios of GP over CP, GP over CS, and CP over CS, respectively. The last row also represents the average ratios.

The energy saving ratio of GP over CP (G2CP) decreases as the latency increases. Its average value is 8.5 percent. The energy saving ratio of GP over CS (G2CS) decreases as the latency increases also. Its average value is 85.3 percent. The energy saving ratio of CP over CS (CP2CS) decreases as the latency increases. Its average value is 84.9 percent.

5.3 Summary

In summary, the comparison of LECPAES and GP from the aspects of energy consumption, throughput, and retransmission rate can be seen in Table 6. Because energy saving ratio is a relative value and its statistical difference is meaningless, it will not be listed in the following Tables 6 to 8. The differences in the table are calculated by the corresponding value of LECPAES minus that of GP. From the table, LECPAES costs less energy, but with a little lower throughput and a little higher retransmission rate, than GP.

TABLE 6
LECPAES versus GP

latency (us)	Energy Consumption Difference (J)	Throughput Difference (MBps)	Retransmission Rate Difference
100	14.64789	−5.43462	0.00233
500	−116.70678	−3.58103	0.00172
1,000	−312.04820	−0.13804	0.00233
1,500	−454.78482	−0.84615	0.00050
2,000	−587.49921	−0.05752	0.00075
2,500	−726.29618	−0.08636	0.00000
3,000	−864.94611	−0.99010	0.00000
avg	−435.37620	−1.59055	0.00109

TABLE 7
LECPAES versus CP

latency (us)	Energy Consumption Difference (J)	Throughput Difference (MBps)	Retransmission Rate Difference
100	−899.95776	220.68415	−4.40675
500	−769.74009	93.33504	−0.83745
1,000	−593.37253	26.72644	−0.23183
1,500	−311.44435	2.27806	−0.00317
2,000	−271.50469	1.93408	−0.00174
2,500	−253.71760	1.14931	0.00000
3,000	−243.89636	0.86728	0.00000
avg	−477.66191	49.56776	−0.78299

TABLE 8
LECPAES versus CS

latency (us)	Energy Consumption Difference (J)	Throughput Difference (MBps)	Retransmission Rate Difference
100	−10,085.40719	276.50737	−60.13958
500	−9,955.36973	147.90988	−14.66937
1,000	−9,712.35478	77.32657	−7.34900
1,500	−9,444.21666	51.49978	−4.76192
2,000	−9,117.86565	37.80085	−3.47817
2,500	−9,050.31162	29.36779	−2.68150
3,000	−8,509.86151	23.76355	−2.16567
avg	−9,410.76959	92.02511	−13.60646

The trend is that the amount of variation becomes smaller with higher latency. On average, compared to GP, LECPAES saves energy of 435.37620 J, reduces throughput of 1.59055 MBps, and increases retransmission rate of 0.00109.

The comparison of LECPAES and CP can be seen in Table 7, where the differences are calculated by the corresponding value of LECPAES minus that of CP. In this table, the tread has changed compared to that in Table 6. LECPAES costs less energy, but with higher throughput and lower retransmission rate than CP. On average, compared to CP, LECPAES saves 477.66191 J energy, increases throughput of 49.56776 MBps, and reduces retransmission rate of 0.78299.

The comparison of LECPAES and CS can be seen in Table 8, where the differences are calculated by the corresponding value of LECPAES minus that of CS. In this table, the tread is same with that in Table 7, but with larger different value. On average, compared to CS, LECPAES saves energy of 9,410.76959 J, increases throughput of 92.02511 MBps, and reduces retransmission rate of 13.60646.

6 Discussion

Through the experiments, we have the following findings.

- For fast streams, LECPAES consumes little more energy than GP. For slow streams, LECPAES consumes the least energy compared to GP, CP, and CS. This is because, for faster streams, LECPAES will result in higher retransmission rate and lower throughput, but the profits brought by scaling frequency cannot offset them. As the velocity of streams increases, LECPAES gets more profits from scaling frequency and becomes saving more energy than GP. This can be seen that LECPAES, GP, and CP will consume more energy with the increase of the velocity of streams. CS consumes almost the same energy with the increase of the velocity of streams. This is because CS performs encryption in serial and its speed is far slower than the speed of inflow. Moreover, faster streams will cause data retransmission later, thus the energy consumption of CS is determined by the time of draining the stream. In the experiments, even the slowest stream is still faster than the speed of encryption outflow, thus CS will spend almost the same time of encrypting the data for streams with different velocity and consume almost the same energy.
- With the decrease of the velocity of streams, LECPAES, GP, and CP decrease their throughput, but CS keeps stable throughput. Faster streams will cause higher retransmission rate, but when calculating throughput, only the successfully encrypted data are used. That is to say these streams have the same amount of data. Because CS encrypts the streams at the slowest speed and the speed is far slower than the veloctiy of inflow, CS produces the lowest throughput and keeps stable. CP produces the second lowest throughput because of higher encryption speed than CS but slower encryption speed than GP and LECPAES. For slower streams, CP, GP, and LECPAES will spend more time to complete the encryption, thus they decrease throughput for slower streams. Because GP and LECPAES have higher encryption speed than CP, thus CP and LECPAES produce higher throughput than CP. In addition, because LECPAES dynamically scales the frequency, LECPAES has a little slower encryption speed than GP and results in a slightly lower throughput than GP.
- GP and LECPAES are suitable for fast streams, but LECPAES consumes less energy than GP. For fast streams, GP and LECPAES consume less energy, produce higher throughput, and lower retransmission rate compared to CP and CS. This is because, the GPU undertakes full workload and produces better energy efficiency. Because GP and LECPAES encrypt data at high speed, they have higher throughput and lower retransmission rate. Moreover, LECPAES consumes less energy than GP through dynamically scaling frequency, but only slightly decreases throughput and slightly increases retransmission rate. In short, LECPAES is the best algorithm among them for fast and energy-saving stream encryption.

7 Conclusion

In this work, in order to expedite encrypting streams in cloud environment in energy saving manner, a velocity-aware parallel encryption algorithm with low energy consumption, called LECPAES, is proposed.

In LECPAES, a sliding window is adopted and used to sense the velocity of current stream. Then a scaling frequency scheme is employed based on the current sensed stream velocity. The encryption is performed on the GPU in many threads fashion. As comparison, CS, CP are also designed for suiting stream data and are compared with GP and LECPAES from the aspects of energy consumption, throughput, and retransmission rate.

Some experiments are conducted in a server with one K20M GPU and two Xeon CPUs and show the following results. (1) LECPAES can reduce energy consumption, slightly reduce throughput, and slightly increase retransmission rate compared with GP. (2) LECPAES can reduce energy consumption, increase throughput, and reduce retransmission rate compared with CP. (3) LECPAES can reduce energy consumption largely, increase throughput largely, and reduce retransmission rate largely compared with CS. Therefore, LECPAES is suitable for fast stream encryption in energy saving manner through sensing velocity.

This paper represents our initial work in stream encryption with low energy consumption. We consider only the stable streams with fixed inflow speed. In future, we plan to extend this work to suit streams with complicated inflow speed.

ACKNOWLEDGMENTS

The authors would like to thank the editors and the anonymous reviewers for their comments and suggestions to improve the manuscript. The research was partially funded by the Key Program of National Natural Science Foundation of China (Grant No. 61432005), the National Outstanding Youth Science Program of National Natural Science Foundation of China (Grant No. 61625202), the National Natural Science Foundation of China (Grant Nos. 61370095, 61472124, 61572175), International Science & Technology Cooperation Program of China (2015DFA11240), National High-tech R&D Program of China (2015AA015303).

REFERENCES

[1] D. H. Woo and H.-H. S. Lee, "Extending Amdahl's law for energy-efficient computing in the many-core era," *IEEE Comput.*, vol. 41, no. 12, pp. 24–31, 2008.

[2] top500.org, "The top500 list," [Online]. Available: http://www.top500.org/

[3] P. FIPS, "197: Specification for the Advanced Encryption Standard, 2001," 2009. [Online]. Available: http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[4] H. Lipmaa, P. Rogaway, and D. Wagner, "Comments to NIST concerning aes modes of operations: Ctr-mode encryption," (2000). [Online]. Available: http://www.cs.ucdavis.edu/rogaway/papers/ctr.pdf

[5] Y. Abe, H. Sasaki, M. Peres, K. Inoue, K. Murakami, and S. Kato, "Power and performance analysis of gpu-accelerated systems," in *Proc. USENIX Conf. Power-Aware Comput. Syst.*, vol. 12, 2012, pp. 1–5.

[6] R. Ge, R. Vogt, J. Majumder, A. Alam, M. Burtscher, and Z. Zong, "Effects of dynamic voltage and frequency scaling on a K20 GPU," in *Proc. 42nd Int. Conf. Parallel Process.*, 2013, pp. 826–833.

[7] K. Ma, X. Li, W. Chen, C. Zhang, and X. Wang, "Greengpu: A holistic approach to energy efficiency in GPU-CPU heterogeneous architectures," in *Proc. 41st Int. Conf. Parallel Process.*, 2012, pp. 48–57.

[8] M. Arora, S. Manne, I. Paul, N. Jayasena, and D. M. Tullsen, "Understanding idle behavior and power gating mechanisms in the context of modern benchmarks on CPU-GPU integrated systems," in *Proc. 21st Int. Symp. High Performance Comput. Archit.*, 2015, pp. 366–377.

[9] C. Tang, D. O. Wu, A. T. Chronopoulos, and C. S. Raghavendra, "Efficient multi-party digital signature using adaptive secret sharing for low-power devices in wireless networks," *IEEE Trans. Wireless Commun.*, vol. 8, no. 2, pp. 882–889, Feb. 2009.

[10] Z. Liu, E. Wenger, and J. Groschadl, "MoTE-ECC: Energy-scalable elliptic curve cryptography for wireless sensor networks," in *Proc. 12th Int. Conf. Appl. Cryptography Netw. Security*, vol. 8479, 2014, pp. 361–379.

[11] X. Ma, M. Dong, L. Zhong, and Z. Deng, "Statistical power consumption analysis and modeling for GPU-based computing," in *Proc. ACM SOSP Workshop Power Aware Comput. Syst.*, 2009, pp. 1–5.

[12] H. Nagasaka, N. Maruyama, A. Nukada, T. Endo, and S. Matsuoka, "Statistical power modeling of GPU kernels using performance counters," in *Proc. Green Comput. Conf. Int.*, 2010, pp. 115–122.

[13] S. Hong and H. Kim, "An integrated GPU power and performance model," in *Proc. 37th Annu. Int. Symp. Comput. Archit.*, vol. 38, no. 3, 2010, pp. 280–289.

[14] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "GPGPU performance and power estimation using machine learning," in *Proc. 21st Int. Symp. High Performance Comput. Archit.*, 2015, pp. 564–576.

[15] K. Kasichayanula, D. Terpstra, P. Luszczek, S. Tomov, S. Moore, and G. D. Peterson, "Power aware computing on GPUs," in *Proc. Symp. Appl. Accelerators High Performance Comput.*, 2012, pp. 64–73.

[16] K. Li, W. Yang, and K. Li, "Performance analysis and optimization for SpMV on GPU using probabilistic modeling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 1, pp. 196–205, Jan. 2015.

[17] W. Yang, K. Li, and Z. Mo, "Performance optimization using partitioned SpMV on GPUs and multicore CPUs," *IEEE Trans. Comput.*, vol. 64, no. 9, pp. 2623–2636, Sep. 2015.

[18] S. A. Manavski, "CUDA compatible GPU as an efficient hardware accelerator for AES cryptography," in *Proc. Int. Conf. Signal Process. Commun.*, 2007, pp. 65–68.

[19] P. Maistri, F. Masson, and R. Leveugle, "Implementation of the advanced encryption standard on GPUs with the NVIDIA CUDA framework," in *Proc. Symp. Ind. Electron. Appl.*, 2011, pp. 213–217.

[20] K. Iwai, N. Nishikawa, and T. Kurokawa, "Acceleration of AES encryption on CUDA GPU," *Int. J. Netw. Comput.*, vol. 2, no. 1, pp. 131–145, 2012.

[21] N. Nishikawa, K. Iwai, and T. Kurokawa, "High-performance symmetric block ciphers on CUDA," in *Proc. 2nd Int. Conf. Netw. Comput.*, 2011, pp. 221–227.

[22] X. Fei, K. Li, W. Yang, and K. Li, "Practical parallel AES algorithms on cloud for massive users and their performance evaluation," *Concurrency Comput. Practice Exp.*, vol. 28, pp. 4246–4263, 2016.

[23] Y. Wang, Z. Feng, H. Guo, C. He, and Y. Yang, "Scene recognition acceleration using CUDA and OpenMP," in *Proc. 1st Int. Conf. Inf. Sci. Eng.*, 2009, pp. 1422–1425.

[24] G. Liu, et al., "A program behavior study of block cryptography algorithms on gpgpu," in *Proc. 4th Int. Conf. Frontier Comput. Sci. Technol.*, 2009, pp. 33–39.

[25] F. Shao, Z. Chang, and Y. Zhang, "AES encryption algorithm based on the high performance computing of GPU," in *Proc. 2nd Int. Conf. Commun. Softw. Netw.*, 2010, pp. 588–590.

[26] C. Mei, H. Jiang, and J. Jenness, "CUDA-based AES parallelization with fine-tuned GPU memory utilization," in *Proc. Int. Symp. Parallel Distrib. Process. Workshops Phd Forum*, 2010, pp. 1–7.

[27] K. Iwai, T. Kurokawa, and N. Nisikawa, "AES encryption implementation on CUDA GPU and its analysis," in *Proc. 1st Int. Conf. Netw. Comput.*, 2010, pp. 209–214.

[28] T. Nhat-Phuong, L. Myungho, H. Sugwon, and L. Seung-Jae, "High throughput parallelization of AES-CTR algorithm," *IEICE Trans. Inf. Syst.*, vol. 96, no. 8, pp. 1685–1695, 2013.

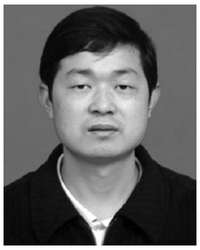
[29] X. Fei, K. Li, W. Yang, and K. Li, "A secure and efficient file protecting system based on SHA3 and parallel AES," *Parallel Comput.*, vol. 52, no. 2, pp. 106–132, 2016.

[30] T. D. Burd and R. W. Brodersen, "Energy efficient CMOS microprocessor design," in *Proc. 28th Hawaii Int. Conf. Syst. Sci.*, vol. 1, pp. 288–297, 1995.

[31] Y. Lin, X. Yang, T. Tang, G. Wang, and X. Xu, "An integrated energy optimization approach for CPU-GPU heterogeneous systems based on critical path analysis," *Chin. J. Comput.*, vol. 35, no. 1, pp. 123–133, 2012.

[32] N. Corp, "Nvml api reference manual," (2012). [Online]. Available: http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/NVML/nvml.pdf

[33] R. Levering and M. Cutler, "The portrait of a common HTML web page," in *Proc. ACM Symp. Document Eng.*, 2006, pp. 198–204.



Xiongwei Fei received the MS degree in computer science from Hunan University, China, in 2008. He is currently working toward the PhD degree at Hunan University, China. He is an associate professor of computer science and technology at Hunan City University, China. His research interests include parallel computing, cryptography algorithms, and cloud computing.

Kenli Li received the PhD degree in computer science from Huazhong University of Science and Technology, China, in 2003. He was a visiting scholar at the University of Illinois at Urbana-Champaign from 2004 to 2005. He is currently a full professor of computer science and technology at Hunan University and the deputy director in the National Supercomputing Center in Changsha. His major research areas include parallel computing, high-performance computing, and grid and cloud computing. He has published more than 130 research papers in international conferences and journals such as *IEEE Transactions on Computers, IEEE Transactions on Parallel and Distributed Systems, Journal of Parallel and Distributed Computing*, ICPP, CCGrid. He is an outstanding member of CCF. He serves on the editorial board of the *IEEE Transactions on Computers*. He is a senior member of the IEEE.

Wangdong Yang received the MS degree from Central South University, China, in 2006. He is currently working toward the PhD degree at Hunan University, China. He is a professor of computer science and technology, Hunan City University, China. His research interests include modeling and programming for distributed computing systems, parallel algorithms, grid and cloud computing.

Keqin Li is a SUNY distinguished professor of computer science. His current research interests include parallel computing and high-performance computing, distributed computing, energy-efficient computing and communication, heterogeneous computing systems, cloud computing, big data computing, CPU-GPU hybrid and cooperative computing, multicore computing, storage and file systems, wireless communication networks, sensor networks, peer-to-peer file sharing systems, mobile computing, service computing, Internet of things and cyber-physical systems. He has published more than 470 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. He is currently or has served on the editorial boards of *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, *IEEE Transactions on Cloud Computing*, *IEEE Transactions on Services Computing*, *IEEE Transactions on Sustainable Computing*. He is a fellow of the IEEE.