CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2016; **28**:4246–4263
Published online 17 December 2015 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3734

SPECIAL ISSUE PAPER

Practical parallel AES algorithms on cloud for massive users and their performance evaluation

Xiongwei Fei[1,2], Kenli Li[1,2,*,†], Wangdong Yang[1,2] and Keqin Li[1,2,3]

[1] *College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China*
[2] *National Supercomputing Center in Changsha, Hunan University, Changsha 410082, China*
[3] *Department of Computer Science, State University of New York, New Paltz, NY 12561, USA*

SUMMARY

Many e-business or social network servers have been constructed on cloud. On such open environments, private data of massive users have to be protected by encrypting, such as using Advanced Encryption Standard (AES), and furthermore, this process must be finished in a short time for users' better experience. This gives huge pressure on cloud servers, especially common servers, such as web servers. We urgently need an inexpensive and highly efficient method to relieve cloud servers' pressure. Fortunately, many cores of a graphics processing unit (GPU) can undertake this hard mission because of stronger computing power and lower price. The GPU environments can be virtualized on demand by cloud through the vCUDA technology. Of course, for those clouds not equipped with a GPU, a central processing unit (CPU) can still work as multithreads in parallel. Thus, in a cloud, AES can be parallelized using many cores of a GPU or multicores of a CPU with high efficiency and low cost. For typical cloud applications, such as web services, there are massive users and each one has short plaintext. If we simply parallelize AES in such an application, we cannot obtain better performance because of the GPU's extra data transferring cost. Thus, we coalesce the massive users' data and cut these data into same-length slices for improving the performance of parallel AES as much as possible. So we design six parallel AES algorithms using GPU parallelism or CPU parallelism, which differ in parallel scope and whether data are coalesced or cut to slices. Specifically, they are coalescent and sliced GPU (GCS), coalescent and unsliced GPU, uncoalescent GPU, coalescent and sliced CPU, coalescent and unsliced CPU, and uncoalescent CPU. Moreover, we implement them on two representative platforms and evaluate their performance. Through comparing their performance, GCS has the best performance among these algorithms. In a cloud with Nvidia GPUs, GCS is a more powerful algorithm for massive users' data encrypting, relatively. Copyright © 2015 John Wiley & Sons, Ltd.

Received 6 January 2015; Revised 31 August 2015; Accepted 11 November 2015

KEY WORDS: Advanced Encryption Standard; cloud; coalescent; CPU parallelism; GPU parallelism; performance evaluation; slice

1. INTRODUCTION

Currently, more and more people use e-business or social network services. Thus, the servers are suffering heavy workloads on one side and must offer data security for users' private data on the other side, usually by encrypting with Advanced Encryption Standard (AES). However, the servers are hard to upgrade for satisfying these constantly increasing demands because of extra investment or difficulties of redeploying.

Fortunately, many servers have been constructed on the clouds with CPUs and graphics processing units (GPUs). Through vCUDA[1], a cloud can virtualize a suitable GPU environment on

[*] Correspondence to: Kenli Li, College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China.

[†] E-mail: lkl@hnu.edu.cn

demand for an application. Of course, for those clouds not equipped with GPUs, a CPU can still work as multithreads for high-performance computing. Thus, in a cloud, AES can be parallelized using many cores of a GPU or multicores of a CPU with high efficiency and low cost. Furthermore, the servers' embarrassing situation can be relieved by GPU parallelism or CPU parallelism.

In some typical applications, massive users' data are usually short but may be different in length. For example, in a secure socket layer (SSL), transferring data usually varies from 35 to 150 KB, but different user's data may have a different length [2]. If we just use GPU parallelism or CPU parallelism to accelerate encrypting in this situation, it does not fully utilize the computing performance of a CPU or GPU, because a GPU or CPU often encounter these short plaintexts. Especially, GPU parallelism will fall into the difficulty of transferring massive little plaintexts.

This paper endeavors to solve the effective encrypting problem for massive users with short plaintexts. We propose six efficient parallel AES algorithms and evaluate their performance on two different platforms.

In short, our main contributions are summarized as follows.

- For the first time in existing research, we consider parallel AES encrypting for common applications with massive users, where each user has short but possibly different length data.
- We improve the encrypting efficiency of such applications by the methods of coalescing and/or slicing.
- We design a series of parallel AES algorithms for cloud either equipped with a GPU or CPU. These parallel AES algorithms cover all aspects, such as parallel scope, coalescing, and slicing.
- We implement these six algorithms and evaluate their performance on two representative platforms.

The rest of this paper is organized as follows. Section 2 reviews some related work on parallel AES algorithms. In Section 3, we present and analyze six different CPU-parallel or GPU-parallel AES algorithms for massive users' data encrypting. Next, Section 4 describes our experiments and compares the performance of these practical parallel AES algorithms and then discusses and analyzes the experimental results. In Section 5, we make a conclusion for all the work and look forward to our future work.

2. RELATED WORK

Advanced Encryption Standard (AES) [3, 4] is widely used in enterprises or organizations for encrypting/decrypting data, which is chosen as a standard for higher efficiency and stronger security than its competitors, such as RC6, etc.

In the past decade, the CPU has had multicores and continues to increase the number of cores, while the GPU has developed to many cores as well, and more GPUs have evolved to support general-purpose computing. Thus, we can use many threads in a CPU or GPU to quickly solve some problems in parallel. Without exception, there are many researches about improving AES efficiency by using GPU parallelism or CPU parallelism, especially the latter.

There are mainly two reasons for the popularity of GPU parallelism. One reason is that GPU programming becomes easy, because it is supported by new frameworks, such as Computing Unified Device Architecture (CUDA) offered by Nvidia. The other reason is that GPU parallelism can produce powerful computing, because a GPU usually has hundreds (currently thousands on high-end GPU) of processing cores and very-high data bandwidth. For example, Li *et al.* [5] improved the performance of sparse matrix-vector multiplication on a GPU using probabilistic modeling.

The first paper [6], which uses CUDA to parallelize AES on a GPU, achieved about 20 times speedup for OpenSSL (Open Secure Socket Layer) data. Li *et al.* [7] achieved the maximum performance of around 60 Gbps throughput using CUDA on an NVIDIA Tesla C2050 GPU. Maistri *et al.* [8] implemented parallel AES using CUDA and obtained good performance with excellent price/performance ratio. In addition, some further analysis and optimization for parallel AES on GPU can be found in [9–17].

If a cloud is not equipped with a GPU, another possible parallelism is CPU parallelism. Currently, a CPU usually has multicores and these cores can perform some workloads in parallel. In such

DOI: 10.1002/cpe

a CPU, shared memory parallel programs can be designed by the mature OpenMP (Open Multiprocessing) parallel technology. OpenMP can also be easily used to develop parallel programs. This is because multithreads can be supported by a set of simple directives in OpenMP. AES of course can be parallelized on a CPU by means of multiple threads, which are indicated by OpenMP directives.

This attracts the attention of researchers as well. For instance, Navalgund *et al.* [18] implemented a parallel AES using OpenMP and achieved attractive performance improvement. Duta *et al.* [19] parallelized AES using CUDA, OpenMP, and OpenCL (Open Computing Language), respectively, and the rank of their performance is CUDA, OpenCL, and OpenMP in descending order. Besides, Pousa *et al.* [20] implemented three parallel AES algorithms by means of using OpenMP, MPI, and CUDA, respectively, and parallel AES by CUDA also exhibits higher efficiency. Recently, Nagendra and Sekhar [21] used OpenMP to implement a parallel AES, which costs 40–45% less time than the sequential one. Ortega *et al.* [22] parallelized AES on multicores CPU using OpenMP and on GPUs using CUDA, respectively, and observed that the latter is superior to the former. These research progresses consolidate that parallelizing AES using CUDA is a strong and effective method.

In addition, Banu *et al.* [2] accelerated AES by combining hardware parallelism made by field-programmable gate array and software parallelism (i.e., OpenMP). Currently, Liu and Baas [23] parallelized AES on an Asynchronous Array of Simple Processors and gained better performance with higher energy efficiency. The hardware parallelism techniques demonstrate higher efficiency than the software techniques but have the drawbacks of lacking of flexibility of implementation and needing extra costly investment.

All in all, for a cloud equipped with Nvidia GPUs, it is a better method to parallelize AES using CUDA. Whereas, for a cloud server not deploying NVIDIA GPUs, it can still accelerate AES by CPU parallelism using OpenMP.

The final aim of improving AES efficiency by parallel techniques is to apply it to practical applications, such as SSL or HTML (Hyper Text Markup Language) services. This is a key step and should consider the actual situations as well. Some researchers have explored the applications of parallel AES. As mentioned before, the first paper [6] using CUDA to parallelize AES on GPU is finally used to accelerate encrypting of the data of OpenSSL applications. Besides, Jang *et al.* [24] implemented a parallel SSL on a GPU, including AES, RSA, and HMAC-SHA1, and obtained the speed of 21.5K SSL transactions per second. Moreover, in database applications, Fazackerley *et al.* [25] implemented Cypher Block Chained parallel AES on GPU.

These researches do not concern the massive users' data. But with the number of network users increasing stably, definitely, the massive users' data should be effectively protected by encrypting in open environments. Among them, one of the typical applications is HTML, whose features include that the data usually vary from 35 KB to 150 KB, and that massive users offer data at the same time, such as e-order for buying train tickets (e.g., on a common holiday) or e-business for goods on sales (e.g., during an important festival), and so on. The data of one user are little and must be quickly encrypted for good experience of users. Thus, we cannot avoid the fact of cloud servers' suffering from the demands of massive users' data encrypting. So, we consider adopting CPU parallelism and GPU parallelism to solve this problem effectively.

To the best of our knowledge, this is the first time to consider how to effectively design parallel AES for practical applications with massive users.

3. PRACTICAL PARALLEL ADVANCED ENCRYPTION STANDARD ALGORITHMS

In this section, we will propose and analyze six different GPU-parallel or CPU-parallel AES algorithms for massive users' little plaintext encrypting.

3.1. Introduction to Advanced Encryption Standard

Advanced Encryption Standard (AES) is a symmetric group cipher. In details, it uses the same key and the same operating structures for encrypting and decrypting, and its group is as long as 16 bytes. Its key can be as long as 16, 24, or 32 bytes, so AES-128, AES-192, and AES-256 are named respectively after the number of bits of key. A different length of key means different operating rounds.

DOI: 10.1002/cpe

AES-128 runs 10 rounds, while AES-192 and AES-256 run 12 and 14 rounds, respectively. Each round includes four phrases: SubBytes, ShiftRows, MixColumns, and AddRoundKey, except before the first round there is an extra AddRoundKey and the last round does not have a MixColumns. The key is expanded to round keys for AddRoundKey. Because there is an extra AddRoundKey before the first round, AES-128, AES-192, and AES-256 expand their 16-byte keys to 11, 13, and 15 round keys, respectively. Each group of plaintext is called *state* and is expressed as a 4×4 array. AddRoundKey performances XOR operation on round key and *state* for blinding. SubBytes executes non-linear byte substitution and can be implemented by SBox, which is a fixed table. ShiftRows cyclically shifts each row of *state* by 0, 1, 2, and 3 bytes, respectively. MixColumns performs multiplications on each column of the *state* by a polynomial $'03'x^3 +' 01'x^2 +' 01'x +' 02'$ modulo $x^4 + 1$. Obviously, this is a time-consuming process but can be accelerated by precomputed T tables.

Once AES expands its key to round keys, these round keys can be used by all the *state* of the plaintext needed to be encrypted. The encrypting of each *state* is independent and the round operators are major workloads. In addition, GPUs offered by Nvidia can support general integer computing. For GPUs, the key expansion only executes once in serial and the *states* of plaintext that have the same encrypting workloads can be encrypted on GPU in parallel. Thus, AES can perform on many-core GPU efficiently in parallel. Of course, AES can also run on a CPU in parallel but has less parallelism because of less cores.

In practical, AES employs a certain mode to add extra security for large amounts of data. There are five modes [26], including Electronic Code Book, Cipher Feedback, Output Feedback, CTR (Counter), and CBC (Cipher Block Chaining). CTR and CBC are more notable, but CBC does not support parallel encrypting because of data dependence within groups. CTR is a new added mode and is strongly recommended by Lipmaa *et al.* [27] for provable security and supporting parallel operation. Thus, we adopt the CTR mode to achieve our parallel AES algorithms.

3.2. Parallel Advanced Encryption Standard Algorithms

Parallel AES algorithms adopt different parallel schemes. They are coalescent and sliced GPU (GCS), coalescent and unsliced GPU (GCNS), uncoalescent GPU (GNC), coalescent and sliced CPU (CCS), coalescent and unsliced CPU (CCNS), and uncoalescent CPU (CNC). GCS means such a parallel scheme, which coalesces the massive users' little plaintexts and cuts them into slices evenly, and then a GPU encrypts these slices simultaneously. GCNS first coalesces massive users' little plaintexts but does not cut them into slices, then uses GPU to encrypt these data in parallel. Whereas, GNC does not coalesce the massive users' little plaintexts and directly executes encrypting on GPU in parallel.

In fact, GNC is based on the approach proposed by Duta *et al.* [19], which neither coalesces nor slices the data of users. We adapt it to accommodate the environment of massive users. So, we name it as GNC for the convenience of comparing.

Similarly, CCS, CCNS, and CNC work by CPU parallelism comparing with the former three algorithms. Likewise, CNC neither coalesces nor slices the data of users. In fact, CNC is based on the approach proposed by Navalgund *et al.* [18]. We adapt it to make it suitable for the data of massive users and name it as CNC for the convenience of comparing.

The details of coalescing and slicing can be seen in Figures 1. For better showing these algorithms' difference, we list their main features in Table I. In practical applications, a server could face massive users' encrypting requests, and it can buffer these users' plaintexts and adopt these six parallel AES algorithms to encrypt them.

Coalescent and sliced GPU (GCS) coalesces the buffered users' plaintexts and cuts them into same-length slices and then sends these slices to a GPU. A GPU executes these encryption works simultaneously by using its many cores. Because these slices have almost the same length and one slice is in charged of one block, threads in a block have about the same amount of workload. This is beneficial to fully utilize GPU computing resources.

Whereas, GCNS coalesces users' plaintexts but does not cut them into same-length slices. Thus, GCNS arranges one user's data to one block, and each thread of block encrypts some groups of

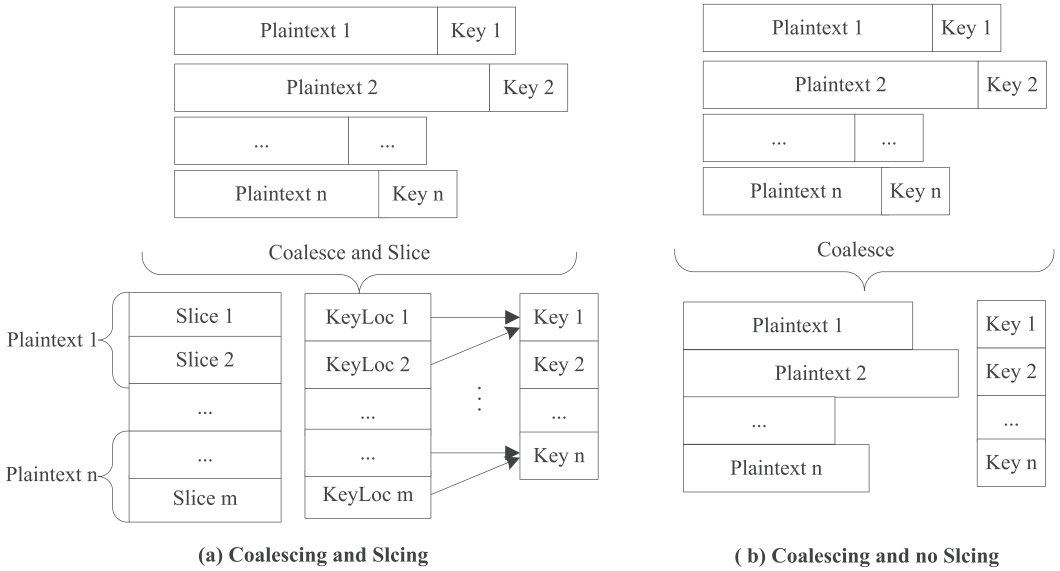

Figure 1. (a) The buffered users' data are coalesced and cut into slices of the same length. The key location of each slice is stored in KeyLocs. (b) The buffered users' data are just coalesced but not cut into slices.

Table I. Comparison of six practical parallel Advanced Encryption Standard algorithms.

Name	Parallel scope	Coalescent?	Sliced?
GCS	Buffered users' data on GPU	Yes	Yes
GCNS	Buffered users' data on GPU	Yes	No
GNC	One user's data on GPU after another user's	No	No
CCS	Buffered users' data on CPU	Yes	Yes
CCNS	Buffered users' data on CPU	Yes	No
CNC	One user's data on CPU after another user's	No	No

GPU, graphics processing unit; GCS, coalescent and sliced GPU; GCNS, coalescent and unsliced GPU; GNC, uncoalescent GPU; CCS, coalescent and sliced CPU; CCNS, coalescent and unsliced CPU; CNC, uncoalescent CPU.

16-byte plaintext. Because different blocks may have different encrypting workload, some blocks may spend more time to encrypt larger data.

Uncoalescent GPU (GNC) has a simpler work process that sends the data of one user to a GPU once, and then this user's data are encrypted in parallel on a GPU. Then the other users' data repeat the same process one by one. The encrypting works are distributed to many blocks and each block is in charge of an equal part. For example, each thread of a block encrypts a group of 16-byte plaintext. If a block is organized as 256 threads/block, it will be in charge of the encrypting of 16 bytes × 256 = 4 KB plaintext.

All CCS, CCNS, and CNC work as CPU parallelism and are similar to the previous GCS, GCNS, and GNC but can be used on the cloud not equipped with Nvidia GPUs. For conciseness, we omit their descriptions as readers can infer them from the descriptions of previous GCS, GCNS, and GNC algorithms.

3.3. *Coalescent and sliced graphics processing unit*

The GCS algorithm is shown in Algorithm 1. In practice, servers can buffer massive users' little plaintexts easily if network users are sufficient. Thus, the GCS algorithm first coalesces and cuts these data into some equal slices with length of *sliceLen*.

Obviously, a user's data can be divided into several slices, so GCS uses an *exkeyLocs* array to record the extended key location of each slice for finding the extended key in later GPU encrypting. In AES-128, the 16-byte key should be extended to 176 bytes for 11 AddRoundKeys. Because a

DOI: 10.1002/cpe

Algorithm 1 GCS algorithm

```
Require:
    number of users, $users$;
    buffered users' plaintext data, $uData$;
    users' data length, $uLens$;
    users' keys, $uKeys$;
    slice length, $sliceLen$;
Ensure: $uData$ are the AES ciphertext
1: coalesce users' data to $coalesceData$ and divide them to pieces as long as $sliceLen$;
2: extend each users' key to $exKeys$ and store the extended key locations $exKeyLocs$ for each slice
   data in CPU parallelism;
3: allocate device memory for storing $exKeys$ as $dExKeys$, $T$ Table as $dT$, $coalesceData$ as $dInput$,
   $exKeyLocs$ as $dExKeyLocs$;
4: transfer $exKeys$, $T$, $coalesceData$ ,and $exKeyLocs$ from main memory to GPU global memory
5: define dimension of block and grid as $dimBlock$ and $dimGrid$ respectively by device properties
6: execute AES parallel kernel $GCS\_Cipher <<< dimGrid, dimBlock >>>$
   $(dInput, dExKeys, dT, dExKeyLocs, tLen, users, sliceLen)$;
7: copy ciphertext from $dInput$ to $coalesceData$
```

user's data use the same extended key, GCS extends *uKeys* to *exKeys* for all users using CPU parallelism as shown in Line 2, and then one of *exKeys* will be used by each slice. Next, GCS allocates sufficient device memory to store the data from the main memory, including *dExKeys*, which stores *exKeys*, *dT*, which stores T Table (a 4 KB constant table, which is used to accelerate AES), *dInput*, which stores *coalesceData*, and *dKeyLocs*, which stores *exKeyLocs*.

After this, the users' data are transferred from the main memory to the GPU global memory, and then AES is performed by calling the kernel function *GCS_Cipher* on GPU simultaneously. The *GCS_Cipher* works by using the T tables to accelerate speed and allocating shared memory *sT* for T tables and *sKey* for the extended key for alleviating access latency. The T tables have the length of 4 KB and the length of the extended key is 176 bytes. Thus, the shared memory is enough for storing them without any problem. Through *dExkeys* and *dKeyLocs*, we can get the extended key as *dExKeys*[*dExKeyLocs*[*sliceID*]] for this slice.

The *GCS_Cipher* organizes the threads as follows. Each block has a size of $Bs = sliceLen/16$ threads and is in charge of the encrypting of a slice. In addition, GCS employs two methods to improve performance as follows:

- using the granularity of 16B/Thread, which is the best granularity discussed in Ref. [16].
- using the block size of 100% occupancy.

Here we describe how to choose the block size *Bs* of 100% occupancy for GPU parallel AES algorithms. The occupancy *Oc* is calculated by

$$Oc = Aw/Mw \times 100\%, \tag{1}$$

where *Aw* represents the active warps in a multiprocessor, and *Mw* denotes the maximum possible warps in a multiprocessor. *Aw* is limited by maximum warps or blocks per multiprocessor, registers per multiprocessor, and shared memory per multiprocessor. In other words, the number of active blocks is limited by the aforementioned factors and must be the minimum of these limitations. For example, GPUs with computing capacity 3.0 or 3.5 have the parameters as shown in Table II. In GCS, each thread needs 19 registers and each block needs 4344 bytes shared memory according to compilation information. There are three possible block sizes whose occupancy is 100%, that is, 256, 512, and 1024. For block size 256, the occupancy is computed as follows:

Firstly, the number of active blocks limited by registers, N_{br}, is calculated as:

$$N_{br} = \frac{Rm}{Bs \times \frac{Rt+Ru-1}{Ru} \times Ru} = \frac{65536}{256 \times \frac{19+7}{8} \times 8} = 10, \tag{2}$$

Table II. Parameters of graphics processing units with computing capacity 3.0 or 3.5.

Maximal warps per multiprocessor	64
Maximal thread blocks per multiprocessor	16
Registers per multiprocessor	65,536
Register allocation unit size	8
Shared memory per multiprocessor	49,152
Shared memory allocation unit size	256
Maximum thread block size	1024
Threads per warp	32

Table III. Occupancy of graphics processing units with computing capacity 3.0 or 3.5 for coalescent and sliced graphics processing unit.

Block size	Register lim.	SM lim.	Max-blocks-warps lim.	Warps per block	Active warps	Occupancy
256	10	11	8	8	64	100%
512	5	11	4	16	64	100%
1024	2	11	2	32	64	100%

where *Rm* represents registers per multiprocessor, *Bs* denotes the size of block, *Rt* represents registers per thread, *Ru* represents the size of register allocation unit, and the divisions are exact divisions.

Secondly, the number of active blocks limited by shared memory, N_{bsm}, is calculated as:

$$N_{bsm} = \frac{Sm}{\frac{Sb+Su-1}{Su} \times Su} = \frac{49152}{\frac{4344+255}{256} \times 256} = 11, \tag{3}$$

where *Sm* represents shared memory per multiprocessor, *Sb* denotes shared memory per block, and *Su* represents the size of shared memory allocation unit.

Thirdly, the number of active blocks limited by max-blocks-warps, N_{bmbw}, is calculated as:

$$N_{bmbw} = \min\left(Mb, \frac{Mw \times Ws}{Bs}\right) = \min\left(16, \frac{64 \times 32}{256}\right) = 8, \tag{4}$$

where *Mb* represents maximum thread blocks per multiprocessor, *Mw* represents maximum warps per multiprocessor, *Ws* represents the size of warp, and *Bs* denotes the size of block.

Fourthly, the number of maximum active blocks *Ab* is the minimum of the above three numbers, that is,

$$Ab = \min(N_{br}, N_{bsm}, N_{bmbw}) = 8. \tag{5}$$

Fifthly, the total active warps is

$$Aw = \frac{Ab \times Bs}{Ws} = \frac{8 \times 256}{32} = 64. \tag{6}$$

Finally, the occupancy is $Aw/Mw \times 100\% = 64/64 \times 100\% = 100\%$.

The occupancy of block sizes of 512 and 1024 can be computed similarly as shown in Table III. However, a larger block size means more data will be padded into slices in GCS. Therefore, GCS employs a block size of 256 for GPUs with computing capacity 3.0 or 3.5 in our implementation. That means $sliceLen = 16 \times Bs = 16 \times 256 = 4$ KB. The grid has *tLen/sliceLen* blocks, where *tLen* denotes the length of *coalesceData*. Each thread in a block encrypts a group of 16-byte plaintext.

In the kernel *GCS_Cipher*, each block is in charge of encrypting a slice with the ID of $sliceID = blockIdx.x + blockIdx.y * gridDim.x$. Each block allocates shared memory *sT* and *sKey* for *dT* and *dExKeys*[*dExKeyLocs*[*sliceID*]], respectively. *dExKeyLocs*[*sliceID*] represents the location of extend

DOI: 10.1002/cpe

key for the slice, while *dExKeys*[*dExKeyLocs*[*sliceID*]] denotes the extend key for the slice. Each thread of a block executes AES encrypting by using T Tables *sT* and key *sKey*.

Once the kernel is finished, the buffered users' data have been encrypted, so we can copy them through PCIe (Peripheral Component Interconnect Express) bus from *dInput* to *coalesceData*. Naturally, the latter is the ciphertext of the buffered users' data. Because we know every user's data length through *uLens*, GCS can easily separate the ciphertext of each user. By now, GCS has finished all the users' data encrypting and obtained each user's ciphertext. Each block encrypts one slice by means of many threads.

3.4. Coalescent and unsliced graphics processing unit

Now, we present our second algorithm GCNS. There is only one difference, that is, 'Sliced?', between GCNS and GCS as shown in Table I. The details of GCNS can be seen in Algorithm 2.

Massive users' data can be buffered in a server. Then GCNS coalesces the users' data and keys to *coalesceData* and *coalesceKeys*, respectively. Because of no cutting the data to slices, GCNS allocates device spaces *dKeys* to store *coalesceKeys*, *dT* to store T tables, *dinput* to store *coalesceData*, and *duLen* to store *uLen*, respectively. After this, *coalesceData* etc. can be transferred from main memory to these spaces.

Next, GCNS defines *users* blocks as *dimGrid*(*users*) and *Bs* threads in a block, where *Bs* is defined by GPU properties for 100% occupancy. According to compilation information, each thread needs 23 registers and each block needs 4592 bytes shared memory. Similarly, the best block size is calculated by the previous formulae 1 to 6. There are still three block sizes of 100% occupancy for GPUs with computing capacity 3.0 or 3.5, that is, 256, 512, and 1024. For comparing, GCNS defines the block size as 256 as well. Next, GCNS will call the kernel function *GCNS_Cipher* for encrypting on GPU simultaneously.

In the kernel *GCNS_Cipher*, each block is in charge of encrypting a user's data. Each block allocates shared memory *sT* for *dT* and *sKey* for the extended key. Then, it computes the block ID as $bid = blockIdx.x + blockIdx.y * gridDim.x$ and then assigns a thread, such as $threadIdx.x = 0$, to extend the key from *dKeys*[*bid*] to *sExKey*. After these, each thread of a block executes AES encrypting by using T Tables *sT* and *sExKey*, but could encrypt several groups of 16-byte plaintext according to the size of the user's data.

There are several differences in the kernel function of GCS and GCNS. The first one is that GCNS uses a block to encrypt a user's data, thus different user's data may result in different workloads of blocks. The second one is that GCNS will compute the extended key *dKeys*[*bid*] on a GPU by one thread of a block, where *bid* is the ID of blocks. In addition, for the threads in a block, a thread will encrypt multiple groups of 16-byte plaintext. For instance, if a user has a 64-KB plaintext and a block has 256 threads, each thread should encrypt sixteen 16-byte plaintext as $16\times256\times16$ byte $=$ 64 KB.

Algorithm 2 GCNS algorithm

```
Require:
    number of users, users;
    buffered users' plaintext data, uData;
    users' data length, uLens;
    users' keys, uKeys;
Ensure: uData are the AES ciphertext
  coalesce users' data uData to coalesceData and uKeys to coalesceKeys ;
  allocate device memory for storing coalesceKeys as dKeys, T Tables as dT, coalesceData as
      dInput, uLens as dLens;
  transfer coalesceKeys, T, coalesceData, and uLens from main memory to corresponding GPU
      global memory
  define dimension of block and grid as dimGrid and dimBlock respectively by device properties;
  execute AES parallel kernel GCNS_Cipher <<< dimGrid, dimBlock >>>
      (dInput, dKeys, dLens, dT, users);
  copy ciphertext from dInput to coalesceData
```

DOI: 10.1002/cpe

3.5. Uncoalescent graphics processing unit

Next, we will describe our third algorithm GNC, which encrypts massive users' data without coalescing or cutting them. GNC encrypts one user's data on GPU in parallel after another user's data are encrypted. This means that each user's data are encrypted on a GPU simultaneously one user by one user, but different user's data are dealt in different times. Algorithm 3 shows the parallel process of GNC. It encrypts all the user's data orderly as shown in the loop (Line 1). In the loop body,

Algorithm 3 GNC algorithm

```
Require:
    number of users, $users$;
    buffered users' plaintext data, $uData$;
    users' data length, $uLens$;
    users' keys, $uKeys$;
Ensure: $uData$ are the AES ciphertext
1: for $i = 1$ to $users$ do
2:     extend $uKeys[i]$ to $uExKey$;
3:     allocate device memory for storing $uExKey$ as $dExKey$, $T$ Table as $dT$, $uData[i]$ as $dInput$;
4:     transfer $uExKey$ and $uData[i]$ from main memory to GPU global memory;
5:     define dimension of grid and block as $dimGrid$ and $dimBlock$ by device properties;
6:     execute AES parallel kernel $GNC\_Cipher <<< dimGrid, dimBlock >>>$
       $(dInput, dExKey, dT, uLens[i])$;
7:     copy ciphertext from $dInput$ to $uData[i]$
8: end for
```

GNC first extends the current user's key *uKeys*[i] to *uExKey*, then allocates device memory *dExKey*, *dT*, and *dInput* for storing *uExKey*, T tables, and current user's data *uData*[i]. Next, GNC transfers *uExKey*, T, and *uData*[i] from the main memory to the GPU device memory.

Sequentially, it defines block dimension and grid dimension by GPU properties. According to the compilation information, each thread needs 27 registers and each block needs 4320 bytes shared memory for T tables, *dExKey* and so on. For computing capacity 3.0 or 3.5 GPUs, GNC employs the block size of 256 for high occupancy according to the previous formulae 1 to 6. GNC adopts the grid size of $uLen[i]/(16 \times Bs)$.

In the kernel *GNC_Cipher*, all blocks on GPU are in charge of encrypting a user's data. Each block allocates shared memory sT for *dT* and *sExKey* for the extended key *dExKey*. Each thread of a block executes AES encrypting for a group of 16-byte plaintext of *dInput* by using T Tables sT and *sExKey*.

After the kernel function finishes, GNC can get current user's ciphertext through copying data from *dInput* to *uData*[i]. If the loop finished, all the buffered users' data have been encrypted.

3.6. Coalescent and sliced CPU

In this subsection, we will propose our fourth algorithm CCS. CCS coalesces and cuts massive users' data to the same long slices and then encrypts these slices on a CPU simultaneously by means of multiple threads. If a server is not equipped with Nvidia GPUs, it can use this algorithm to execute the encrypting of massive users' data.

Algorithm 4 demonstrates the details of CCS. Firstly, this algorithm coalesces and cuts the users' data to slices equally and computes the key location *keyLocs*[i] of each slice in *uKeys* as shown in Line 3. Then CCS assigns *WORK_THREAD* threads to encrypt some slices simultaneously by using OpenMP. Each thread should first extend the slice's key *uKeys*[*keyLocs*[i]] to get the extended key *exKeys*[i] as shown in Line 6, then encrypts a slice's data *coalesceData*[$sliceLen \times i$]. Then, CCS uses T tables and *exKeys*[i] to encrypt the slice. T tables can accelerate AES encrypting as well. The workloads of each thread will be allocated by OpenMP and can be kept almost same.

Algorithm 4 CCS algorithm

```
Require:
    number of users, users;
    buffered users' data, uData;
    users' data length, uLens;
    users' keys, uKeys;
    slice length, sliceLen;
Ensure: uData are the AES ciphertext
    coalesce users' data to coalesceData
    cut coalesceData to numSlices slices as long as sliceLen
    compute KeyLocs for each slice
    #pragma omp parallel for num_threads(WORK_THREADS);//where WORK_THREAD represents the number of threads in CPU. Usually it equals the number of cores of CPU.
    for $i = 1$ to numSlices do
        extend uKeys[keyLocs[$i$]] to get a slice's extended key exKeys[$i$];
        encrypt a slice's data coalesceData[$sliceLen * i$] using exKeys[$i$] and $T$ tables;
    end for
```

3.7. Coalescent and unsliced CPU

The CCNS algorithm will be proposed here. Similarly to GCNS, the difference between CCS and CCNS is whether the coalesced data are cut into slices. CCNS misses the process of cutting and looks simpler. The details of CCNS can be seen in Algorithm 5.

The algorithm first coalesces buffered users' data but does not cut them. Then it defines an array *plainLocs* to store the start position of each user's data in *coalesceData*. After computing these locations in Line 2, CCNS uses OpenMP to derive *WORK_THREADS* threads and these threads will be assigned some concrete encrypting workloads. The workloads include extending the user's key *uKeys*[i] to get the corresponding extended key *exKeys*[i] and encrypting the user's data, which start from *coalesceData*[*plainLocs*[i]], with the length *uLens*[i]. These threads simultaneously run on a CPU and fully use the computing resources of a CPU.

Algorithm 5 CCNS algorithm

```
Require:
    number of users, users;
    buffered users' data, uData;
    users' data length, uLens;
    users' keys, uKeys;
Ensure: uData are the AES ciphertext
 1: coalesce users' data to coalesceData;
 2: compute the start position of each user's data plainLocs;
 3: #pragma omp parallel for num_threads(WORK_THREADS); //where WORK_THREAD represents the number of threads in CPU. Usually it equals to the number of cores of CPU.
 4: for $i = 1$ to users do
 5:     extend uKeys[$i$] to get a user's extended key exKeys[$i$];
 6:     encrypt a user's data coalesceData[PlainLocs[$i$]], whose size is uLens[$i$], with exKeys[$i$] and $T$ Tables;
 7: end for
```

3.8. Uncoalescent CPU

Now, we put forward the last algorithm CNC of this work. CNC does not coalesce massive users' data, so this means that it will deal one user's data after another one. But for a certain user's

DOI: 10.1002/cpe

data, CNC will encrypt them simultaneously by multiple threads of CPU. Obviously, the difference between CCNS and CNC is that CCNS lets all users' data to be dealt simultaneously by multiple threads of CPU, but CNC just lets one user's data.

We list the details of CNC in Algorithm 6. CNC circularly deals the user's data one user by one user as shown in Line 1. In the loop (Line 1), CNC first extends the current user's key *uKeys*$[i]$ to obtain the extended key *exKeys*$[i]$ and then derives *WORK_THREADS* threads of CPU to perform AES encrypting on *uData*$[i]$ with the extended key *exKeys*$[i]$ simultaneously as shown in Line 5. Each thread j encrypts a part of the workloads with the size of *uLens*$[i]/$*WORK_THREADS*. These workloads are scheduled by OpenMP, thus can be kept in balance. Of course, in the process of encrypting, CNC also uses T tables to accelerate AES. After the internal loop ends at Line 6, which means a user's data has been encrypted, i will be increased by 1 to continue encrypting another users' data. When the external loop ends in Line 7, all the users' data have been encrypted.

Algorithm 6 CNC algorithm

```
Require:
    number of users, users;
    buffered users' data, uData;
    users' data length, uLens;
    users' keys, uKeys;
Ensure: uData are the AES ciphertext
1: for $i = 1$ to users do
2:   extend uKeys[$i$] to get a user's extended key exKeys[$i$];
3:   #pragma omp parallel for num_threads(WORK_THREADS);//where WORK_THREAD represents the number of threads in CPU. Usually it equals the number of cores of CPU.
4:   for $j = 1$ to WORK_THREADS do
5:     encrypt a part of the $i$-th user's data uData[$i$][$j \times$ uLens[$i$]/WORK_THREADS], whose size is uLens[$i$]/WORK_THREADS;
6:   end for
7: end for
```

3.9. Analysis of parallel Advanced Encryption Standard algorithms

In this subsection, the overhead and encrypting time of parallel AES will be analyzed. The overhead will be analyzed firstly. GNC transfers users' data one by one to the device memory and then encrypts them on a GPU in parallel. Thus, the overhead of GNC for massive N users can be calculated as:

$$N \times \left(\frac{2 \times PL + KL}{BW} + 2 \times delay \right), \tag{7}$$

where PL represents the length of plaintext, KL represents the length of key, BW denotes the bandwidth of transferring, and *delay* is the communication delay of transferring.

Coalescent and sliced GPU (GCNS) and GCS differ in slicing. If the users' data are regular, which means every user's data are multiple length of slice, the overhead of GCS and GCNS for massive N users can be calculated as:

$$\frac{N \times (2 \times PL + KL + PL/SliceLen \times KeyLoc)}{BW} + 2 \times delay, \tag{8}$$

$$\frac{N \times (2 \times PL + KL)}{BW} + 2 \times delay \tag{9}$$

respectively, where *sliceLen* represents the length of a slice, *KeyLoc* represents the length of key location. If the users' data are irregular, which means users' data may be not multiple length of slice, the overhead of GCS for massive N users can be calculated as:

$$\frac{N \times (2 \times PL + sliceLen/2 + KL + PL/SliceLen \times KeyLoc)}{BW} + 2 \times delay, \qquad (10)$$

where $sliceLen/2$ denotes the average padded data length for a user; the overhead of GCNS is still calculated by formula (9). In a word, the transferring overhead is decreased by GNC, GCS, and GCNS. CNC, CCNS, and CCS do not need transfer data, thus coalescing and slicing do not cause communication overhead.

Next, encrypting data time will be analyzed here. GNC encrypts a user's data by a grid (kernel) in parallel. But different users' data are encrypted one by one serially. For small users' data, GPU computing resources will not be fully used. However, GCNS and GCS encrypt all buffered users' data by a grid. This will fully use the GPU computing resources and reduce encrypting time. Furthermore, GCNS encrypts each user's data by a block of threads, while GCS encrypts each slice data by a block of threads in parallel. Because different users may have different length of data, GCNS may lead to imbalanced workloads for different users. Whereas, each block has the same amount data to encrypt in GCS, and GCS can finish at almost the same time. Thus, GCS will spend less time in encrypting buffered users' data. In a word, for massive users' buffered data, the encrypting time is decreased by GNC, GCNS, and GCS.

As for CPU parallelism, CNC encrypts buffered users' data one by one serially but each user's data by multiple threads. This means that each thread will encrypt almost the same amount data. But CNC needs more overhead of thread schedule and management. Both CCNS and CNS encrypt buffered users' data in parallel and their difference is whether the data has been sliced. CCNS does not slice users' data, which means that different thread may encrypt different long data. Whereas, CCS cuts users' data to identical length slices, which makes different threads encrypt identical length data also. But CCS will incur extra padded data such as key locations. So in theory, the threads of CNC and CCS have more balanced workload than CCNS, but both CNC and CCS need more overhead at the same time as analyzed before.

4. EXPERIMENTS AND PERFORMANCE EVALUATION

In this section, we will implement all our proposed six algorithms and evaluate their performance. Our experiments are performed on two respective platforms, whose specifications are introduced in Subsection 4.1. The experimental results and performance evaluation are discussed in 4.3 and 4.4, respectively.

4.1. Experimental platforms

In order to exhibit the effects of our proposed practical parallel AES algorithms for massive users' data and evaluate their performance, some experiments are operated on two different platforms whose specifications are shown in Table IV. The platform 1 is a general computer representing low-end server, but the platform 2 is a powerful computing platform representing high-end server.

Table IV. Configurations of two experiment platforms.

Platform	GPU	CPU
1	NVIDIA GeForce GT 640M Clock Rate: 709 MHz Total: 384 Cores	Intel i5 3220 (2 Cores) Clock Rate: 2.6 GHz Total: 2 Cores
2	NVIDIA Tesla K20M Clock Rate: 706 MHz Total: 2496 Cores	2 Intel Xeon E5-2640 v2 (8 Cores) Clock Rate: 2.0 GHz Total: 16 Cores

GPU, graphics processing unit.

DOI: 10.1002/cpe

4.2. *Experimental assumptions*

- Assumption 1 (Users' data assumption): We consider the users' data coming from typical SSL transactions, whose sizes vary from 35 to 150 KB.
- Assumption 2 (The length of users' data assumption): First, we assume that the lengths of user's data satisfy a normal distribution function.
- Assumption 3 (Normal distribution function assumption): For simulating the behavior of a normal distribution function, we use the method of 'Box Muller' to produce the normally distributed data.

4.3. *Experimental results on platform 1*

According to the six algorithms presented in Section 3, we implement them and evaluate their performance by the number of users from 5 to 10,000. We choose regular data and irregular data as users' plaintext and give their results in the following 4.3.1 and 4.3.2, respectively.

4.3.1. Regular data. Figure 2 demonstrates the encrypting speed of seven algorithms on platform 1 for regular data whose length is a multiple of *sliceLen*. Among them, SR represents the serial AES algorithm for massive users, which encrypts one user's data serially after another user's by means of a single thread. All the experimental results are obtained as the mean value of 100 experiments.

From the figure, SR are slower than the other parallel algorithms. CNC, CCNS, CCS, GNC, and SR basically keep stable encrypting speed, while GCS and GCNS accelerate their encrypting speeds with the number users increasing from 5 to 200 and then keep steady encrypting speeds.

Quantitatively, for all the test cases from 5 users to 10,000 users on platform 1, the average encrypting speed of GCS, GCNS, GNC, CCS, CCNS, CNC, and SR are 0.537388 Gbps, 0.500823 Gbps, 0.279641 Gbps, 0.172115 Gbps, 0.166311 Gbps, 0.162538 Gbps, and 0.075087 Gbps, respectively. In other words, GCS is faster than GCNS by 7.3% at average speed, which is calculated by

$$(SGCS - SGCNS)/SGCNS \times 100\%,$$

where *SGCS* and *SGCNS* represent the encrypting speed of GCS and GCNS. Similarly, GCS is faster than GNC, CCS, CCNS, CNC, and SR by 92.17%, 212.23%, 223.12%, 230.62%, and 615.69%, respectively, at average speed on platform 1.

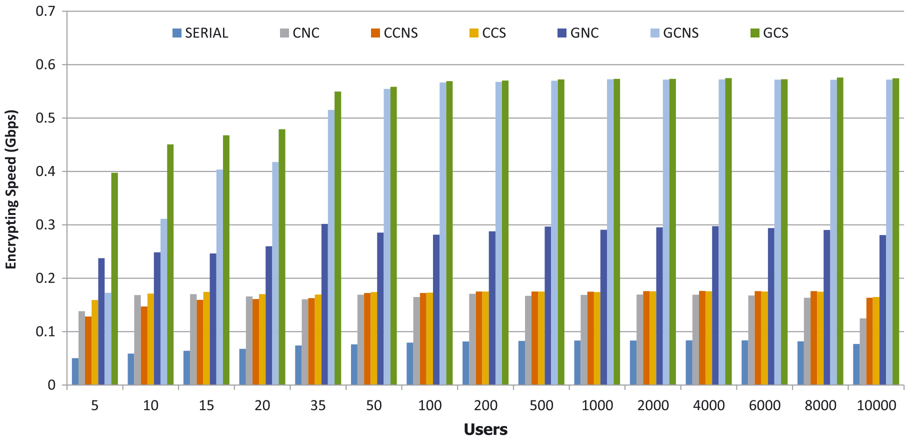

Figure 2. The encrypting speed of seven algorithms on platform 1 for regular data: coalescent and sliced graphics processing unit (GCS), coalescent and unsliced graphics processing unit (GCNS), uncoalescent graphics processing unit (GNC), coalescent and sliced CPU (CCS), coalescent and unsliced CPU (CCNS), and uncoalescent CPU (CNC) represent our proposed six practical parallel algorithms for massive users, while serial (SR) denotes CPU serial Advanced Encryption Standard algorithm, which serially encrypts each user's data one by one in a single thread.

DOI: 10.1002/cpe

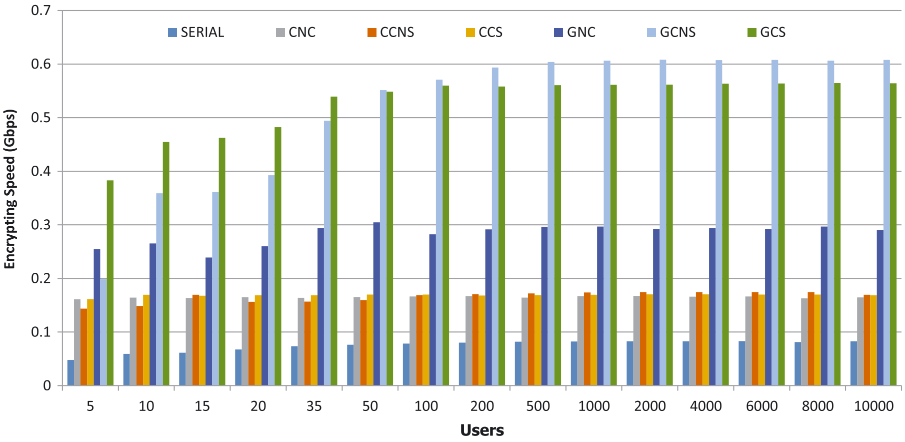

Figure 3. The encrypting speed of seven algorithms on platform 1 for irregular data: coalescent and sliced graphics processing unit (GCS), coalescent and unsliced graphics processing unit (GCNS), uncoalescent graphics processing unit (GNC), coalescent and sliced CPU (CCS), coalescent and unsliced CPU (CCNS), and uncoalescent CPU (CNC) represent our proposed six practical parallel algorithms for massive users, while serial (SR) denotes CPU serial Advanced Encryption Standard algorithm, which serially encrypts each user's data one by one in a single thread.

4.3.2. Irregular data. For normal distributed users' data whose length is not a multiple of *sliceLen*, we do the same experiments for 100 times to obtain the average encrypting speed of AES algorithms. The experimental results are shown in Figure 3.

From the figure, we still observe that SR are slower than the other parallel algorithms. CNC, CCNS, CCS, GNC, and SR basically keep stable encrypting speed, while GCS and GCNS accelerate their encrypting speeds with the number users increasing from 5 to 200 and then keep steady encrypting speeds.

Quantitatively, the average encrypting speed of GCS, GCNS, GNC, CCS, CCNS, CNC, and SR are 0.528550Gbps, 0.517939Gbps, 0.283295Gbps, 0.168726Gbps, 0.165758Gbps, 0.164906Gbps, and 0.074718Gbps respectively. In other words, GCS is faster than GCNS, GNC, CCS, CCNS, CNC, and SR by 2.05%, 86.57%, 213.26%, 218.87%, 220.52%, and 607.39%, respectively, at average speed on platform 1 for irregular data.

4.4. Experimental results on platform 2

In the previous Subsection 4.3, we give the experimental results on platform 1, which is a notebook with limited computing resources. We can see the effects of our six algorithms, especially GCS. To reflect the reality closely, we urgently want to know the effects of the six algorithms on high-end servers. Thus, we choose platform 2 to evaluate these algorithms. We also choose regular data and irregular data as users' plaintext and give their results in the following 4.4.1 and 4.4.2, respectively.

4.4.1. Regular data. The experimental results on platform 2 for regular data are shown in Figure 4. From the figure, we can find obvious speed improvements over platform 1 and basically the same trends as platform 1.

Quantitatively, for regular data on platform 2, the average encrypting speed of GCS, GCNS, GNC, CCS, CCNS, CNC, and SR are 2.536342 Gbps, 0.941182 Gbps, 0.570270 Gbps, 0.688243 Gbps, 0.566662 Gbps, 0.680594 Gbps, and 0.046192 Gbps, respectively. From the aspect of performance improvement, GCS is faster than GCNS, GNC, CCS, CCNS, CNC, and SR by 30.66%, 344.76%, 268.52%, 347.59%, 272.67%, and 5390.92%, respectively, at average speed for regular data on platform 2.

DOI: 10.1002/cpe

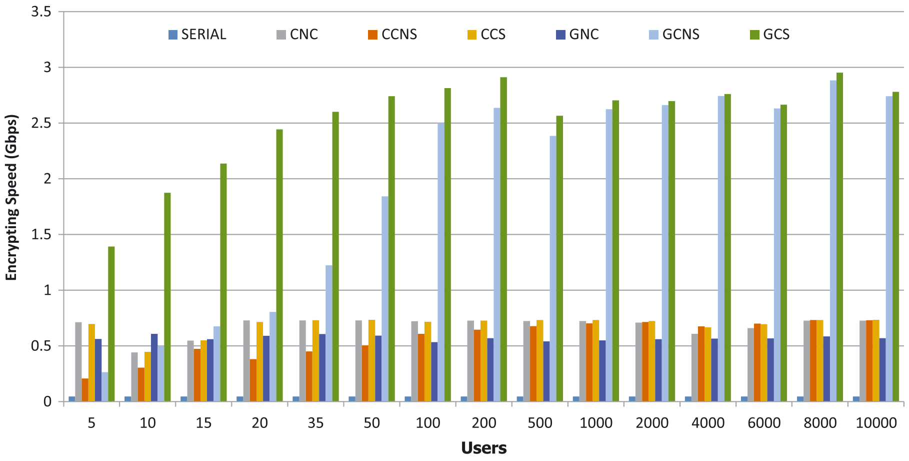

Figure 4. The encrypting speed of seven algorithms on platform 2 for regular data: coalescent and sliced graphics processing unit (GCS), coalescent and unsliced graphics processing unit (GCNS), uncoalescent graphics processing unit (GNC), coalescent and sliced CPU (CCS), coalescent and unsliced CPU (CCNS), and uncoalescent CPU (CNC) represent our proposed six practical parallel algorithms for massive users, while serial (SR) denotes CPU serial Advanced Encryption Standard algorithm, which serially encrypts each user's data one by one in a single thread.

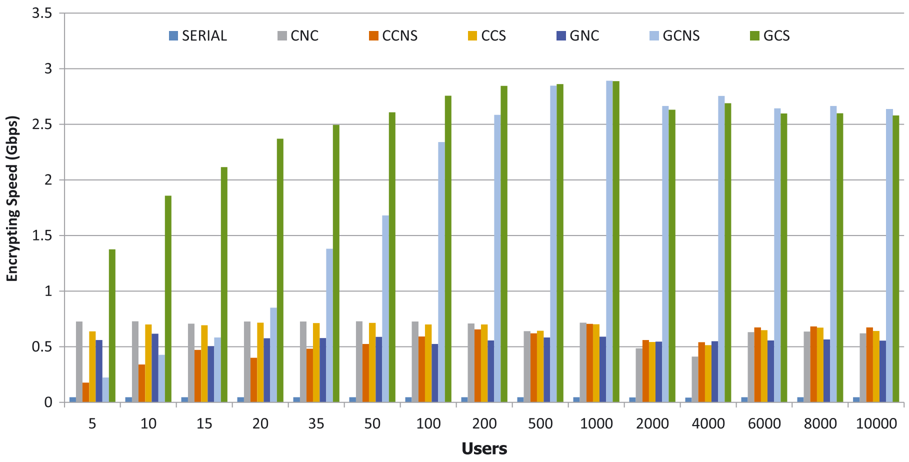

Figure 5. The encrypting speed of seven algorithms on platform 2 for irregular data: coalescent and sliced graphics processing unit (GCS), unsliced graphics processing unit (GCNS), uncoalescent graphics processing unit (GNC), coalescent and sliced CPU (CCS), coalescent and unsliced CPU (CCNS), and uncoalescent CPU (CNC) represent our proposed six practical parallel algorithms for massive users, while serial (SR) denotes CPU serial Advanced Encryption Standard algorithm, which serially encrypts each user's data one by one in a single thread.

4.4.2. Irregular data. The experimental results on platform 2 for irregular data are shown in Figure 5. From the figure, we can find obvious speed improvements over platform 1 and basically the same trends as platform 1.

Quantitatively, the average encrypting speed of GCS, GCNS, GNC, CCS, CCNS, CNC, and SR are 2.485290 Gbps, 1.945289 Gbps, 0.563285 Gbps, 0.662577 Gbps, 0.539771 Gbps, 0.661487 Gbps, and 0.045802 Gbps, respectively. From the aspect of performance improvement, GCS is faster than GCNS, GNC, CCS, CCNS, CNC, and SR by 27.76%, 341.21%, 275.09%, 360.43%, 275.71%, and 5326.17%, respectively, at average speed for irregular data on platform 2.

DOI: 10.1002/cpe

4.5. Experiment analysis and discussion

We analyze and discuss the experimental results as follows:

1. All SR, CNC, CCNS, CCS algorithms keep stable encrypting speed whether for regular data or irregular data. This is because they have not data transferring overhead.
2. The encrypting speed of GCS and GCNS increase with the number of users before reaching their peak performance, while GNC almost keeps its encrypting speed stable. For GCS and GCNS, this is because coalesced users' data hide the data transferring overhead with the increasing of users, and enough workloads can fully use GPU resources. Whereas, GNC encrypts user's data one by one in parallel, so GNC does not have enough data to hide the overhead and fully use GPU resources.
3. For regular data, GCS is faster than GCNS, while for irregular data, GCS becomes a little slower than GCNS when the number of users is enough large. This can be explained as follows. For regular data, although GCS has more transferring overhead as discussed in 3.9, GCS has more balanced workloads. Because extra overhead that lies in transferring *KeyLoc* (4 bytes/slice) is low, GCS can achieve the benefit from balanced workloads. For irregular data, GCS needs transfer *keyLocs* and padded data to make users' data multiple of *sliceLen*. Thus when the number of users is less, GCS still does not fully use GPU resources. GCS can outperform GCNS because of less computing time as analyzed in 3.9. But when the number of users is large enough to fully use GPU resources, GCS will be slower than GCNS because of more data (including padded data) to be encrypted.
4. Uncoalescent GPU (GNC) is faster than CCS, CNC, and CCNS on platform 1 but slower than them on platform 2. This can be explained as follows. The average data transferring speed between the device memory and the main memory in platform 1 is 8405.55 MB/s, while in platform 2 is 6348.53 MB/s. GNC costs more transferring overhead in platform 2 than platform 1. In addition, because of powerful CPUs of 16 cores in platform 2, this causes the performance of GNC to be even worse than CCS, CNC, and CCNS.
5. The speed of CCS, CCNS, and CNC is close. CNC is slower than CCS in our experiments. CCNS usually has the worst speed among them in our experiments. For the former, because CCS coalesces massive uses' data to encrypt in slices, this makes each thread having balanced workload. CNC encrypts users' data one by one, each thread also has balanced workloads. But this needs extra thread managing and schedule overhead. For the latter, because CCNS coalesces users' data but a thread encrypts some users's data, different threads may have different workloads. Thus, CCNS may spend more time in encrypting.
6. Serial (SR) is slowest among these algorithms whether for regular or irregular data, because SR does not fully use CPU resources and encrypts users' data using only one thread in serially. The best speedup of parallel AES to SR is 7.15 in platform 1 and 54.90 in platform 2, calculated as $SGCS/SSR$, where SGCS and SSR are the speed of GCS and SR, respectively.

To summarize, GCS has the best performance at average speed in two respective platforms whether for regular data or irregular data. This proves that coalescing and slicing massive users' data on a GPU can produce excellent efficiency.

5. CONCLUSION

We present six practical parallel AES algorithms for massive users' data encryption on cloud. For some typical services (e.g., HTML data), users' data are usually small but quickly need encrypting. Our algorithms aim at encrypting these data effectively by means of GPU parallelism or CPU parallelism. They are classified and named by parallel scope and the means of coalescing and slicing. Among them, all GCS, GCNS, and GNC use GPU parallelism to accelerate AES speed. All CCS, CCNS, and CNC use CPU parallelism to improve the speed of encrypting in CPU. GCS works by coalescing and cutting the massive users' data to same-length slices, and then these slices are encrypted on a GPU simultaneously. GCNS works by coalescing but no cutting users' data, and then a GPU encrypts the coalesced data in parallel. GNC works by encrypting one user's data on a GPU

DOI: 10.1002/cpe

in parallel after another user's data. CCS works by coalescing and cutting the massive users' data to same-length slices, and then the multiple threads of CPU encrypt these slices simultaneously. CCNS works by coalescing massive users' data, and then the multiple threads of CPU encrypt the coalesced data. In CCNS, each thread is in charge of a part of users' data, which is distributed by OpenMP. CNC works by one user's data are encrypted by the multiple threads of a CPU after another user's.

These six algorithms are implemented on two different but typical platforms. We evaluate their encrypting speed through the experiments by the number of users from 5 to 10,000 on these platforms. The results of evaluation are listed as follows. For regular data, GCS is faster than GCNS, GNC, CCS, CCNS, CNC, and SR by 7.3%, 92.17%, 212.23%, 223.12%, 230.62%, and 615.69%, respectively, at average speed on platform 1. For irregular data, GCS is faster than GCNS, GNC, CCS, CCNS, CNC, and SR by 2.05%, 86.57%, 213.26%, 218.87%, 220.52%, and 607.39%, respectively, at average speed on platform 1. Whereas, for regular data, GCS is faster than GCNS, GNC, CCS, CCNS, CNC, and SR by 30.66%, 344.76%, 268.52%, 347.59%, 272.67%, and 5390.92%, respectively, at average speed on platform 2. For irregular data, GCS is faster than GCNS, GNC, CCS, CCNS, CNC, and SR by 27.76%, 341.21%, 275.09%, 360.43%, 275.71%, and 5326.17%, respectively, at average speed on platform 2. Using GCS-parallel AES for massive users' data, encrypting can produce higher efficiency than the other algorithms whether on general servers or on high-end servers.

This work represents our initial work to deal with massive users' data encryption on cloud and to correspondingly design a series of GPU-parallel or CPU-parallel AES algorithms for improving efficiency. In the future, we plan to further study the AES encryption of massive users' data, such as considering the users' dynamic data flows and combining the CPU parallelism and the GPU parallelism at the same time.

ACKNOWLEDGEMENTS

The authors would like to thank 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 Nos. 61133005, 61432005), the National Natural Science Foundation of China (Grant Nos. 61370095, 61472124, 61572175), International Science & Technology Cooperation Program of China (2015DFA11240), the Scientific Research Fund of Hunan Provincial Education Department (Grant No. 13A011), and the Scientific Research Project of Hunan Provincial Education Department (Grant No. 15C0254).

REFERENCES

1. Shi L, Chen H, Sun J, Li K. vcuda: Gpu-accelerated high-performance computing in virtual machines. *IEEE Transactions on Computers* 2012; **61**(6):804–816.
2. Banu JS, Vanitha M, Vaideeswaran J, Subha S. Loop parallelization and pipelining implementation of AES algorithm using OpenMP and FPGA. In *2013 International Conference on Emerging Trends in Computing, Communication and Nanotechnology (ICE-CCN)*. IEEE: New York, NY, 2013; 481–485.
3. Daemen J, Rijmen V. Aes proposal: Rijndael (Available from: http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf) [accessed on 15 November 2014].
4. FIPS P. 197: Specification for the Advanced Encryption Standard. *National Technical Information Service* 2011: 5–47. (Available from: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf) [accessed on 16 November 2014].
5. Li K, Yang W, Li K. Performance analysis and optimization for spmv on gpu using probabilistic modeling. *IEEE Transactions on Parallel and Distributed Systems* 2014; **26**(1):196–205.
6. Manavski SA. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In *IEEE International Conference on Signal Processing and Communications, 2007. ICSPC 2007*. IEEE: New York, NY, 2007; 65–68.
7. Li Q, Zhong C, Zhao K, Mei X, Chu X. Implementation and analysis of aes encryption on gpu. In *IEEE 14th International conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012*. IEEE: New York, NY, 2012; 843–848.
8. Maistri P, Masson F, Leveugle R. Implementation of the advanced encryption standard on gpus with the nvidia cuda framework. In *2011 IEEE Symposium on Industrial Electronics and Applications (ISIEA)*. IEEE: New York, NY, 2011; 213–217.
9. Bos JW, Osvik D, Stefan D. Fast implementations of AES on various platforms. *IACR Cryptology ePrint Archive* 2009. (Available from: http://eprint.iacr.org/2009/501.pdf) [accessed on 19 January 2015].

DOI: 10.1002/cpe

10. Iwai K, Kurokawa T, Nisikawa N. AES encryption implementation on CUDA GPU and its analysis. In *2010 First International Conference on Networking and Computing (ICNC)*. IEEE: New York, NY, 2010; 209–214.
11. Le D, Chang J, Gou X, Zhang A, Lu C. Parallel aes algorithm for fast data encryption on gpu. In *2010 2nd International Conference on Computer Engineering and Technology (ICCET)*, Vol. 6. IEEE: New York, NY, 2010; 1–6.
12. Shao F, Chang Z, Zhang Y. Aes encryption algorithm based on the high performance computing of gpu. In *Second International Conference on Communication Software and Networks, 2010. ICCSN'10*. IEEE: New York, NY, 2010; 588–590.
13. Mei C, Jiang H, Jenness J. CUDA-based AES parallelization with fine-tuned GPU memory utilization. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and phd Forum (IPDPSW)*. IEEE: New York, NY, 2010; 1–7.
14. Wang Y, Feng Z, Guo H, He C, Yang Y. Scene recognition acceleration using CUDA and OpenMP. In *2009 1st International Conference on Information science and engineering (ICISE)*. IEEE: New York, NY, 2009; 1422–1425.
15. Liu G, An H, Han W, Xu G, Yao P, Xu M, Hao X, Wang Y. A program behavior study of block cryptography algorithms on gpgpu. In *fourth international conference on d/Frontier of computer science and technology, 2009. fcst'09*. IEEE: New York, NY, 2009; 33–39.
16. Iwai K, Nishikawa N, Kurokawa T. Acceleration of AES encryption on CUDA GPU. *International Journal of Networking and Computing* 2012; **2**(1):131–145.
17. Nhat-Phuong T, Myungho L, Sugwon H, Seung-Jae L. High throughput parallelization of AES-CTR algorithm. *IEICE Transactions on Information and Systems* 2013; **96**(8):1685–1695.
18. Navalgund SS, Desai A, Ankalgi K, Yamanur H. Parallelization of AES algorithm using OpenMP. *Lecture Notes on Information Theory* 2013; **1**(4):144–147.
19. Duta C-L, Michiu G, Stoica S, Gheorghe L. Accelerating encryption algorithms using parallelism. In *2013 19th International Conference on Control Systems and Computer Science (CSCS)*. IEEE: New York, NY, 2013; 549–554.
20. Pousa A, Sanz V, De Giusti AR. Performance analysis of a symmetric cryptographic algorithm on multicore architectures. In *2012 Computer Science & Technology Series-XVII Argentine Congress of Computer Science-selected Papers. EDULP*: La Plata, Argentina, 2012; 57–66.
21. Nagendra M, Sekhar MC. Performance improvement of Advanced Encryption Algorithm using parallel computation. *International Journal of Software Engineering and Its Applications* 2014; **8**(2):287–296.
22. Ortega J, Trefftz H, Trefftz C. Parallelizing AES on multicores and GPUs. In *Proceedings of the IEEE International Conference on Electro/Information Technology (EIT)*. IEEE: New York, NY, 2011; 15–17.
23. Liu B, Baas BM. Parallel aes encryption engines for many-core processor arrays. *IEEE Transactions on Computers* 2013; **62**(3):536–547.
24. Jang K, Han S, Han S, Moon S, Park K. Accelerating ssl with gpus. In *ACM SIGCOMM Computer Communication Review*, Vol. 40. ACM: New York, NY, 2010; 437–438.
25. Fazackerley S, McAvoy SM, Lawrence R. Gpu accelerated aes-cbc for database applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. ACM: New York, NY, 2012; 873–878.
26. Dworkin M. Recommendation for block cipher modes of operation. methods and techniques, DTIC Document, 2001. (Available from: http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA400014) [accessed on 23 November 2014].
27. Lipmaa H, Wagner D, Rogaway P. Comments to nist concerning aes modes of operation: Ctr-mode encryption. (Available from: http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/workshop1/papers/lipmaa-ctr.pdf) [accessed on 18 November 2014].

DOI: 10.1002/cpe