A hybrid CPU/GPU Scheme for Optimizing ChaCha20 Stream Cipher

Ziheng Wang, Heng Chen*, Weiling Cai
School of Computer Science and Technology, Xi’an Jiaotong University, Xi’an, China
wzh009888@stu.xjtu.edu.cn, hengchen@xjtu.edu.cn, withinmiaov@stu.xjtu.edu.cn

Abstract—The secure transmission of large-scale data has attracted more and more attention. In the widely recognized security protocol TLSv1.3, the only algorithms that support large-scale data en-/decryption are ChaCha20 and the Advanced Encryption Standard (AES). Although AES has a higher usage rate, ChaCha20 still has the advantage of speed and security on many platforms, and has a better performance against post-quantum attacks. However, for a CPU/GPU platform, compared to the AES algorithm, no work has fully described the application scheme of ChaCha. This paper proposes an optimization scheme to optimize the performance of the ChaCha20 algorithm on a CPU/GPU platform. On a CPU platform, we provide a parallelization implementation that is better than that of OpenSSL. On a single GPU, our implementation of ChaCha20 achieves peak throughput of 211.41GB/s, which is better than any previous implementation of ChaCha20 and AES algorithms on GPU. More importantly, we are the first to detail the optimization of ChaCha on GPU. When considering the interconnection between CPU and GPU, we use the 87.76% peak bidirectional bandwidth of a PCIe channel. Finally, we also provide a scheme for the application of ChaCha20 on a CPU/GPU platform.

Index Terms—ChaCha20, GPU, MPI

I. INTRODUCTION

With the development of the Internet, the issue of network security has attracted more and more attention. At the same time, the era of big data makes the amount of data increase rapidly. Whether on storage systems [1], web servers, and federated learning systems [2], the amount of data is accelerating and the demand for security is higher. It is becoming more common to transmit large amounts of encrypted data. In this case, it is urgent to improve the speed of en-/decryption.

In the widely recognized Transport Layer Security (TLS) v1.3 protocol [3], the only algorithms that support large-scale data encryption are ChaCha20 and the Advanced Encryption Standard (AES) [4]. Although AES has a higher usage rate, ChaCha20 still has the advantage of speed and security on many platforms [5], [6], and has a better performance against post-quantum attacks [7], [8]. However, on a CPU/GPU platform, compared to the AES algorithm, the work on the application scheme of ChaCha has not been introduced in depth. There is not even a work detailing the optimization of ChaCha on GPU.

ChaCha20 [9], a variant of Salsa20 [10], is a 256-bit stream cipher [11]. According to the report in [12], the ChaCha stream cipher is expected to remain secure in 10-50 year lifetime. Salsa-related cipher such as ChaCha can be used to replace RC4, which has theoretically proved unsafe [13]. Besides, in recent years, ChaCha20 has been used as an alternative to the AES block cipher algorithm to increase en-/decryption speed [5]. From the perspective of en-/decryption efficiency, reference [14] described in detail whether ChaCha20 or AES should be used in different scenarios.

This paper proposes several methods to optimize the performance of the ChaCha stream cipher on a CPU/GPU platform. In order to apply the ChaCha algorithm in practice, we also provide a scheme on how to use it more effectively on some platforms. Specifically, this paper makes the following contributions:

- On multi-core CPU, we use Message Passing Interface (MPI) and provide a parallelization implementation of ChaCha20 that is better than that of OpenSSL and any previous implementation of ChaCha20.
- We are the first to optimize ChaCha by utilizing its intrinsic characteristics. Specifically, the ChaCha’s block function is accelerated with a combination of encryption granularity, coalesced memory access, branch operations (rather than index method), and inline Parallel Thread Execution (PTX) techniques. On a single GPU, our implementation of ChaCha20 achieves peak throughput of 211.41GB/s, which is better than any previous implementation of ChaCha20 and AES on GPU.
- When considering the interconnection between CPU and GPU, we use multi-copy technique and achieve a 87.56% peak bidirectional bandwidth of a PCIe channel. Similarly, the throughput in this case is better than any other previous effort.
- We provide a scheme for using ChaCha20 on storage systems, web servers, and federated learning systems, so that the ChaCha20 stream cipher can achieve higher en-/decryption efficiency on CPU/GPU platforms.

The rest of this paper is organized as follows. Section II introduces the ChaCha stream cipher and reviews the related work. Section III describes scheme about optimize ChaCha20 on CPU/GPU platform. Section IV evaluates our methods, and Section V concludes this paper.

* Corresponding author
This research has been supported by the China National Key R&D Program during the 13th Five-year Plan Period (Grant No. 2018YFB1700405).
Algorithm 1 ChaCha’s round

<table>
<thead>
<tr>
<th>Input:</th>
<th>x[16] (initial matrix as array of 4-bytes values)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>x[16] (updated matrix)</td>
</tr>
</tbody>
</table>

1: \#define QUARTERROUND(a, b, c, d) \|
2: \{ x[a] += x[b]; x[d] ^= x[a]; x[d] \ll= 16; \|
3: \{ x[c] += x[d]; x[b] ^= x[c]; x[b] \ll= 12; \|
4: \{ x[a] += x[b]; x[d] ^= x[a]; x[d] \ll= 8; \|
5: \{ x[c] += x[d]; x[b] ^= x[c]; x[b] \ll= 7; \|
6: for i ← 0 to 10 do
7: \{ QUARTERROUND(0, 4, 8, 12)
8: \{ QUARTERROUND(1, 5, 9, 13)
9: \{ QUARTERROUND(2, 6, 10, 14)
10: \{ QUARTERROUND(3, 7, 11, 15)
11: \{ QUARTERROUND(0, 5, 10, 15)
12: \{ QUARTERROUND(1, 6, 11, 12)
13: \{ QUARTERROUND(2, 7, 8, 13)
14: \{ QUARTERROUND(3, 4, 9, 14)

ChaCha Stream Cipher

ChaCha is a stream cipher, which means in encryption it firstly need to generate a keystream, then use the plaintext to encrypt in sequence with the keystream and get the ciphertext. In ChaCha, XOR operation is used in the encryption process (1). In the process of decryption, the keystream is also generated firstly, and then it is XORed with the ciphertext to generate the plaintext (2). In addition, because the encryption overhead is only related to the data size, the algorithm complexity is \(O(n)\). The generation of the keystream is the core of the ChaCha algorithm, also known as the ChaCha block function. It is divided into three parts: matrix initialization, ChaCha’s round and keystream creation.

\[
\text{ciphertext} = \text{plaintext} \oplus \text{keystream} \quad (1)
\]

\[
\text{plaintext} = \text{ciphertext} \oplus \text{keystream} \quad (2)
\]

OpenSSL [15], the de facto standard for secure data transmission, also integrates ChaCha20 stream cipher. Let’s take the scheme used in OpenSSL as an example to illustrate the initial matrix, which consists of 64 bytes, and its form is

\[
\begin{pmatrix}
\text{constant}[0] & \text{constant}[1] & \text{constant}[2] & \text{constant}[3] \\
\text{key}[0] & \text{key}[1] & \text{key}[2] & \text{key}[3] \\
\text{key}[4] & \text{key}[5] & \text{key}[6] & \text{key}[7] \\
\text{counter} & \text{nonce}[0] & \text{nonce}[1] & \text{nonce}[2]
\end{pmatrix}
\]

The first 16 bytes are constants. Under the Request For Comments (RFC) standard [11], they are the hexadecimal notation of the string of “expand 32-byte k”. The next 32 bytes are the key of the symmetric en-/decryption algorithm. Then, the next 4 bytes are the counter, which starts from zero and is used to locate the position of the keystream. This parameter enables the keystream to be accessed randomly, regardless of previous computation. The last 12 bytes, nonce (number-used-once), are a number unique to each keystream.

After the initial matrix has been read, the ChaCha’s round needs to be processed. Algorithm 1 shows the efficient and concise form of ChaCha20’s round. The body of the computation is the 10 iterations, where an iteration includes a row-round and a column-round. There are 20 rounds in total, and each round consists of four quarter-rounds, hence the name ChaCha20. The ChaCha algorithm is also defined as 8 and 12 rounds, which are called ChaCha8 and ChaCha12, respectively, but ChaCha20 is often used in practical applications because of its high security.

Quarter-rounds consist of add, XOR and rotate operations. After a quarter-round, the 16 bytes of a matrix will be updated. After four quarter-rounds, all data of the matrix will be updated. During the update process, the internal execution of a quarter-round is dependent and its order cannot be changed. However, there is no dependence between the four quarter-rounds, so the order can be changed and four quarter-rounds can be computed simultaneously by vectorization [16]. The 20 rounds are sequential and it is not possible to change the order. After running a ChaCha’s round, the updated matrix is added to the initial matrix to get the keystream. This paper refers to this step as keystream creation.

Related Work

As a symmetric encryption algorithm widely used in practice, ChaCha has been tuned on a variety of platforms. On the x86/64 platform, 128-bit vectorization was first applied to speed up a round of ChaCha. The four quarter-rounds are independent, so the 128-bit vector can be used to compute four quarter-rounds simultaneously. With the advent of AVX2, which can handle 256-bit vectors at the same time, M. Goll and S. Gueron [16] noticed that two block functions using 128-bit vectorization could be processed at the same time. Then came AVX512, which could handle 512-bit vectors. ChaCha is extended to handle four block functions simultaneously. As AVX512 introduces more features, ChaCha stream cipher becomes more efficient.

There are a number of optimized implementations of ChaCha on other platforms as well. On FPGA, N. At et al. [17] implemented BLAKE and ChaCha on a Xilinx Virtex-6 device and achieved competitive throughput. On ARM, F. D. Santis et al. [6] optimized the ChaCha20 in two ways, one is the ChaCha’s rounds optimization by rearranging the order of the quarter-rounds and minimizing the amount of memory operations, the other is the quarter-rounds optimization by exploiting the “flexible second operand”. On GPU, R. Velea et al. [18] accelerated the ChaCha20 algorithm. As far as we know, this is the only work on GPU and does not provide any optimizations related to the ChaCha20’s feature.

ChaCha has shown high competitiveness in many fields. B. D. III et al. [5] took advantage of the new features of mobile hardware to apply ChaCha20-poly1305 on the full-drive encryption, improving read performance by as much as 2.36× and providing stronger integrity guarantees than AES-XTS. In 2014, D. J. Bernstein [7] proposes SPHINCS, a high-security post-quantum stateless hash-based signature scheme, where the key generation algorithm is only instantiated with
ChaCha12. Recently, S. Sun [6] implemented SPHINCs on GPU, and instantiated with AES-based hash function Haraka [19] and ChaCha. All the signature implementation based on Chacha are more than 2 times faster than that of Haraka.

Acceleration of cryptographic algorithms has been studied for years. A lot of work has been done to parallelize AES, e.g., multi-core CPU [20], [21], FPGA [22]–[24] and GPU [25]–[28], and the practical performance of AES has been improved continuously. In addition, there is a lot of parallel work on asymmetric encryption. The well-known RSA algorithm has achieved super high throughput on CPU [29], GPU [30] and FPGA [31]. Many studies have also been conducted on Elliptic Curve Cryptography on GPU [32], [33].

III. PARALLEL CHACHA IMPLEMENTATION

For a CPU/GPU platform, it is necessary to choose a scheme to make better use of the algorithm’s performance under different data sizes and platforms. We first describe the optimization approach under CPU and GPU, and then show a CPU/GPU hybrid scheme. In addition, because the en-/decryption processes of the ChaCha algorithm are similar, only the encryption process is described here.

A. CPU Parallel Scheme

The ChaCha algorithm encrypts 64 bytes of data after a block function, which is a small granularity. Because of the dependence of the algorithm execution order, the acceleration of a block function can only be done by the simultaneous computation of four quarter-rounds, which can be completed by the 128-bit vectorization technique. Because the existence of AVX2 and AVX512 vectorization, by executing two or four block functions simultaneously, it becomes more efficient to encrypt 128- or 256-byte plaintext.

To encrypt plaintext larger than 256 bytes, we can consider multi-core acceleration. Commonly used multi-core solutions are OpenMP (pthread) and MPI. Compared with MPI, OpenMP is easy to deploy and has good compatibility, which can achieve multi-core acceleration with fewer changes to the existing system. However, OpenMP is difficult to cope with the situation of multiple computing nodes, and when an encryption process runs on multiple NUMA nodes with virtual cores, it has poor scalability and cannot run on multiple computing nodes at the same time.

Compared with OpenMP, MPI is widely used in multi-node systems because of its high scalability. Here, we use MPI to parallelize CPU implementation. We use the interface of ChaCha20 in OpenSSL to implement the MPI version, where process mapping is the key to improving the encryption performance when running on multiple nodes.

The encryption process consists of three steps, read the plaintext, XOR with the keystream, and write back the ciphertext. So the encryption process is a memory-intensive task. Therefore, when encryption process doesn’t exhaust all cores, in order to reduce memory contention, it is necessary to distribute processes evenly to each computing node and different NUMA nodes on a computing node. It can be realized by using

MPI itself or job scheduling system on different computing nodes. Inside a computing node, the sched_setaffinity function can be used to set the affinity of cores to map the processes to cores.

B. Tuning CUDA Implementation

For such a memory-intensive task, the interconnection between CPU and GPU is often the bottleneck. In general-purpose systems, PCIe is usually used for the interconnection. For higher bandwidth, the faster PCIe should be used for interconnection between GPU and CPU. Therefore, for the implementation of ChaCha20 on GPU, we need to pay attention to both GPU kernel and interconnection between CPU and GPU at the same time. We first discuss GPU optimization without considering interconnection. To tune the GPU kernel, we use the same optimization strategy as that on CPU, that is data parallelism, which is parallelized by executing multiple ChaCha’s block functions by multiple threads. Here, we optimized some implementation details of GPU kernel.

Using integer granularity. Because encryption process requires reading the plaintext from GPU memory and writing back the ciphertext, we need to determine the granularity of reading and writing back data. Our work is based on the original c code of OpenSSL, where the initial granularity is byte. This is not a good granularity for a GPU, because the granularity of GPU memory access is at least 4 bytes. We use memory copy operation memcpy to read / write, and use unsigned integer as the unit for processing. The results show a significant increase in throughput. We also test unsigned long integers and find no acceleration.

Coalesced memory access. We notice that a ChaCha’s block function processes 64-byte plaintext at a time. If a thread encrypts 64 bytes at a time, then the coalesced memory access technology cannot be used. In this case, for ChaCha encryption, such a memory-intensive task, memory access is a bottleneck. Coalesced memory access technology can maximize GPU memory bandwidth, but it requires each thread to access 4 bytes, 8 bytes, or 16 bytes of data each time. Although 4 bytes and 8 bytes can reach the upper limit of GPU memory bandwidth each time, 16 bytes can still approach the upper limit of memory bandwidth if some computation exists between reading and writing back data in recent GPU architecture. In some older GPU architectures, 16-byte coalesced memory access does not work well, which explains why there was very little chacha work on the GPU.
We notice that coalesced memory access technology can be used by this way that a group of threads performs the same computation and generates the keystream, then each thread in a group encrypts part of the data separately. For example, eight threads can each encrypt different eight bytes of data to complete an encryption process. If this technology can take effect, the maximum amount of data a thread can encrypt is 16 bytes. If the actual bandwidth of GPU kernel cannot be close to the upper limit of GPU memory bandwidth, then a single encryption of 16 bytes is a fixed solution.

As shown in Fig. 1, four threads are used to finish a ChaCha’s block function. Firstly, four threads need to perform the same computation. Then each thread reads four integer plaintext from GPU memory, XOR these plaintext with the keystream, generate ciphertext and write them back to GPU memory.

Through Nsight Compute, a Nvidia performance analysis tool, we know that the bottleneck of the ChaCha kernel at this point is the computation time. We note that four quarter-rounds can be computed in parallel using four threads, potentially reducing the time required for a single ChaCha’s block function. But it is unfortunate that because this technology requires the use of shared memory, and each thread needs to perform different computations, there is branch operation or register selection/permutation, and combined with the overhead of synchronization between threads, this method cannot speed up computation.

Branch. Because there are always four threads performing the same computation, so there is a question of whether to create the entire 64-byte keystream for every thread. Fig. 2 shows two schemes to deal with the issue of creating keystream. Index method shows a scheme. If we create all presents the scheme we choose. The define QUARTERROUND(a, b, c, d) shows a scheme. If we create all presents the scheme we choose. The 

```
#define QUARTERROUND(a, b, c, d)
xor.b32 %3, %3, %3, %6;
xor.b32 %1, %1, %2;
xor.b32 %1, %1, %2;
xor.b32 %1, %1, %1, %7;
```

Algorithm 2 Quarter-round using inline PTX

```
1: #define QUARTERROUND(a, b, c, d) 
2: asm ("add.u32 %0, %0, %1; \n\t" \n3: "xor.b32 %3, %3, %3, %6; \n4: "shf.l.clamp.b32 %3, %3, %3, %4; \n\t" \n5: "add.u32 %2, %2, %3; \n\t" \n6: "xor.b32 %1, %1, %2; \n\t" \n7: "shf.l.clamp.b32 %1, %1, %1, %5; \n\t" \n8: "add.u32 %0, %0, %1; \n\t" \n9: "xor.b32 %3, %3, %3, %6; \n\t" 
10: "shf.l.clamp.b32 %2, %3, %2, %3; \n\t" 
11: "add.u32 %2, %2, %3; \n\t" 
12: "xor.b32 %1, %1, %2; \n\t" 
13: "shf.l.clamp.b32 %1, %1, %1, %7; \n\t" 
14: ":+r”(x[a]),“+r”(x[b]),“+r”(x[c]),“+r”(x[d])” \n15: ":r”(16), “r”(12), “r”(8), “r”(7) 
```

reduce the number of registers according to the index: this solution directly adds the corresponding 16-byte updated matrix to the 16-byte initial matrix to generate the corresponding 16-byte keystream. However, due to the increasing frequency of index calls, the performance of this solution is half of the branch operation solution.

This scheme of Branch operation can reduce the number of registers used at the same time. We can limit the upper limit of the number of registers through the -maxrregcount option. Limiting the critical number of registers usually allows more blocks to be mobilized, resulting in more selectable warps. In the implementation of ChaCha’s algorithm, we test both the upper limit of the 24 and 32 registers, and a register upper limit of 32 is found to be optimal, which can improve the maximum throughput.

Inline PTX. In order to reduce the computation time, we use inline PTX to speed up the computation. We find that the rotate instruction can be used to replace the original software implementation of the rotate operation. We also noticed that merging inline PTX statements can improve efficiency, so we merge a quarter-round operation into one statement. The realization of quarter-round is shown in Algorithm 2. Here, shf.l.clamp.b32 is the rotate operation of 32-bit data, which is key step of the algorithm. Moreover, add.u32 and xor.b32 are addition and xor operations of integers, respectively. The r at the end of the algorithm represents the register and is used in order.

C. Multi-copy for CPU/GPU System

In order to achieve the peak transmission bandwidth between GPU and CPU, we use page locked memory to deal with memory allocation request. At the same time, because CPU and GPU are interconnected through PCIe, the multi-copy technology is used to improve the throughput. The PCIe has two channel, which means that we can overlap the process of reading plaintext and write back ciphertext.

The multi-copy technology is implemented using stream technology, which is non-blocking and consists of three com-
ponents: reading data, computing data, and writing data back. The three parts are overlapped by a non-blocking technique. So the task needs to be divided into multiple subtasks, each of which is completed by a stream. In this way, the process of reading data and writing data back can overlap, reducing transmission costs. Generally, as the number of streams increases, the peak throughput will increase, and when the number of streams reaches a certain value, the peak throughput will begin to decrease, so peak throughput tends to have a peak.

In addition, the GPU and CPU can encrypt in parallel, thus improving encryption efficiency. This method only needs to plan the respective encryption amounts of CPU and GPU in advance, and then encrypt their respective data. However, due to the memory contention between the data transmission with GPU and CPU encryption process and the specific way of data distribution, the actual throughput is not simply the addition of the two.

D. CPU/GPU Hybrid Scheme

Various implementation schemes have been described above, but if the ChaCha20 encryption algorithm is applied in practical applications, some factors need to be considered. The following describes a CPU / GPU hybrid scheme from the aspect of application scenarios.

On storage systems, large amounts of data usually need to be preserved in encryption and decrypted when used. In such cases, an efficient encryption scheme is often required. In general, the storage system does not have a GPU, but has multiple processors, where MPI en-/decryption takes advantage. MPI can maximize the advantages of multicore, greatly improve the efficiency of en-/decryption.

Web servers need to encrypt large-scale data in many cases. But because there are often multiple users who need to respond, the server usually has limitations on multi-core acceleration. Considering that the encryption of large-scale data is not frequent, at this time, a CPU / GPU hybrid encryption that use only one core can be used for acceleration, so as to effectively use the idle GPU. Or when the server is relatively idle, multi-core encryption is used. In this case, multi-core encryption can adopt a thread-level method to flexibly change the number of cores that need to be used.

In essence, federated learning is a distributed machine learning technology, which aims to achieve joint modeling and improve the effect of artificial intelligence models on the basis of ensuring data privacy security and legal compliance. It needs to encrypt data for transmission and decrypt data locally. Because the process of machine learning is usually carried out on GPU, the data can be encrypted on GPU after the completion of training and can also be decrypted on GPU before training. This scenario reflects the advantages of en-/decryption on GPU.

Through the above three scenarios, we have learned that we need to choose the encryption scheme according to the specific scenario. In addition, the en-/decryption method should be selected according to the size of the data. Generally speaking, CPU scheme is chosen when the data size is small, while GPU or CPU/GPU mixed encryption is chosen when the data size is large. However, further decisions should be made according to the actual configured hardware environment.

IV. Evaluation

The experimental platform is shown in Table I. For the CPU platform, multi-core processors are used to test the scalability. Our platform has two NUMA nodes, where each has 20 physical cores and 40 logical cores. Also, because memory and PCIe also affect encryption performance, information about them is presented. For the GPU platform, Nvidia GeForce RTX 3090 is used to evaluate the actual effect of the method. PCIe also is tested about the bidirectional peak bandwidth using the program in the CUDA sample, which is 22.47GB/s.

A. CPU Parallel Implementation

Fig. 3 shows the comparison between our method and the OpenSSL method. For general-purpose processors, the ChaCha20 algorithm implemented by OpenSSL is generally considered to be optimal in most cases. Our method uses MPI, combined with the process mapping method, can get close to ideal scalability. For OpenSSL, we use the speed application in OpenSSL to evaluate the throughput, and use the -multi option to specify the number of threads to run. There are 40 logical physical CPU cores used totally. When fewer than 40 logical cores are run, there is little difference between the two methods. However, when the number of cores used by OpenSSL exceeds the number of physical cores, the scalability is poor.

However, there is data reuse in our work and in OpenSSL during evaluating, which is not representative of actual throughput. On a multi-core processor, memory access often limits throughput. We compare the actual throughput with previous work. The table II shows CPU-related work. We only enumerate the throughput on a single processor to demonstrate the validity of our work. In order to achieve greater throughput, we bind the cores to the physical cores when data reuse exists, and to all logical cores when there is no data reuse.

B. GPU Parallel Implementation

This section evaluates the throughput of GPU kernel, which contains three parts: generating the keystream, reading the

<table>
<thead>
<tr>
<th>CPU</th>
<th>Intel Xeon Gold 5218R at 2.1GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>Nvidia GeForce RTX 3090</td>
</tr>
<tr>
<td>Mainboard</td>
<td>X11DA-N</td>
</tr>
<tr>
<td>RAM Memory</td>
<td>32GB DDR4 at 2666MHz × 8</td>
</tr>
<tr>
<td>PCIe</td>
<td>PCI-E 3.0 X16</td>
</tr>
<tr>
<td>Tool Chain</td>
<td>CUDA 11.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table II: Comparison of our work with other works on CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChaCha20</td>
</tr>
<tr>
<td>AES</td>
</tr>
<tr>
<td>AES</td>
</tr>
<tr>
<td>ChaCha20</td>
</tr>
<tr>
<td>ChaCha20</td>
</tr>
</tbody>
</table>
TABLE III
COMPARISON OF OUR WORK WITH OTHER WORKS ON CPU

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>paper</th>
<th>Device</th>
<th>Device throughput</th>
<th>CPU/Device throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Salsa20</td>
<td>2013 [35]</td>
<td>Nvidia GTX 590</td>
<td>27.65GB/s</td>
<td>2.8GB/s</td>
</tr>
<tr>
<td>ChaCha20</td>
<td>2016 [18]</td>
<td>Radeon HD 6850</td>
<td>160GB/s</td>
<td></td>
</tr>
<tr>
<td>AES-ECB</td>
<td>2019 [26]</td>
<td>Nvidia GTX 1080</td>
<td>65.18GB/s</td>
<td>—</td>
</tr>
<tr>
<td>AES-ECB</td>
<td>2019 [26]</td>
<td>Nvidia GTX 1050 Ti</td>
<td>37.09GB/s</td>
<td>—</td>
</tr>
<tr>
<td>AES-ECB</td>
<td>2019 [26]</td>
<td>Tesla P100</td>
<td>104.07GB/s</td>
<td>—</td>
</tr>
<tr>
<td>AES-ECB</td>
<td>2019 [26]</td>
<td>Tesla V100</td>
<td>173.06GB/s</td>
<td>—</td>
</tr>
<tr>
<td>AES</td>
<td>2020 [27]</td>
<td>Nvidia GTX 2070</td>
<td>—</td>
<td>6.35GB/s</td>
</tr>
<tr>
<td>AES-CTR</td>
<td>2020 [28]</td>
<td>Nvidia GTX 1060</td>
<td>—</td>
<td>1.94GB/s</td>
</tr>
<tr>
<td>ChaCha20</td>
<td>This work</td>
<td>Nvidia GTX 3070</td>
<td>119.32GB/s</td>
<td>—</td>
</tr>
<tr>
<td>ChaCha20</td>
<td>This work</td>
<td>Nvidia GTX 3090</td>
<td>211.41GB/s</td>
<td>9.86GB/s</td>
</tr>
<tr>
<td>ChaCha12</td>
<td>This work</td>
<td>Nvidia GTX 3090</td>
<td>322.94GB/s</td>
<td>10.05GB/s</td>
</tr>
<tr>
<td>ChaCha8</td>
<td>This work</td>
<td>Nvidia GTX 3090</td>
<td>401.22GB/s</td>
<td>10.17GB/s</td>
</tr>
</tbody>
</table>

plaintext on GPU memory, and writing the ciphertext back to GPU memory. We have tuned some parameters. Four threads are used to execute an block function in parallel, and the total number of threads is 64. The upper limit of the registers used is 32. Fig. 4 shows the throughput on a GeForce RTX 3070 / 3090, where different data sizes have different throughput and the throughput increases little after 16MB.

This section also evaluates the proposed methods to prove their effectiveness. After each method is added, the total number of threads and the number of threads to processing a single encryption need to be adjusted to get the best performance. As an example, Fig. 5 shows the throughput of GTX 3090. Here, origin represents the result of running origin code of OpenSSL on GPU. Then, +integer means that the plaintext reads and the ciphertext writes at integer granularity instead of byte granularity in origin code. Next, +four shows the result using four thread to finished a single encryption. The fourth legend +branch means that we use the branch operation rather than the index method when selecting the plaintext that needs to be encrypted for a ChaCha’s block function, and at the same time we also limit the maximum of register number. Finally, we use the inline PTX code to replace the C code, and the results are shown in +PTX.

Device throughput in table III shows recent work on ChaCha, Salsa20 and AES algorithm acceleration. Among them, ChaCha8 represents that the number of round is 8, so the compute workload is less. In this way, our method reaches 400GB/s, and almost reaches the limit of GPU memory bandwidth. This also shows from the side that computation is the bottleneck of our method.

Our implementation of ChaCha20 is faster than previous work. But it also shows that on GPU, ChaCha20 does not have too many advantages over AES. Nevertheless, for the encryption system using ChaCha20, GPU acceleration can only be performed for ChaCha20 instead of AES. Therefore, GPU acceleration for ChaCha20 is still very necessary.

C. CPU/GPU Scheme

Here, when we measure GPU time, we consider the data copy time between host memory on CPU and global memory on GPU. On CPU, we will avoid data reuse and obtain real throughput. We only measure the respective time of CPU encryption and GPU encryption, not the time of mixed CPU and GPU encryption. Because the ratio of the amount of encrypted data allocated is not fixed when CPU and GPU are all used to en-/decrypt, it is difficult to ensure the optimal performance.

CPU/Device throughput in Table III shows recent works considering the interconnection between CPU and device.
where the GTX 3070 and the GTX 3090 are in different hardware environments, so the numerical value of the GTX 3070 is not shown. In this case, CHACHA20 can reach a throughput of 9.86 GB/s on the GTX3090, during which both the input and output transmission are performed on the PCIe, utilizing 87.76% (9.86 × 2 / 22.47) of the PCIe bidirectional bandwidth.

Fig. 6 shows the throughput with different number of streams. With different data sizes, there is no certain number of streams that always have the highest throughput. As the amount of data increases, the number of streams with the largest throughput is 1, 2, 4, 8, and 16. However, when the number of streams increases to 32, the throughput no longer follows this rule, and it is almost completely worse than the remaining number of streams.

Fig. 7 shows respective throughput of CPU and GPU, where 16 streams are used to maximum the throughput of GPU. Obviously, CPU encrypts more efficiently when the message is less than 1M, while GPU encrypts more efficiently when the message is greater than 1M. However, CPU implementation may not be able to reach this value in practice because of memory contention among multiple cores. GPUs can also suffer performance degradation when they are not idle. A simple scheme is that CPU encryption can be used when messages are less than 1M and GPU encryption can be used when data is greater than 1M. But we need to adjust our strategy according to the actual situation. In addition, Because the throughput of both is relatively stable when the message size exceeds 2M, encryption can be carried out by CPU and GPU at the same time, and the amount of data is distributed to CPU and GPU at a ratio of 1:2.

V. CONCLUSION

In this paper, we present a hybrid CPU/GPU scheme for optimizing CHACHA20 Stream Cipher. On CPU, We use MPI to provide a better parallel implementation than that of OpenSSL. On GPU, we accelerate CHACHA with a combination of encryption granularity, coalesced memory access, branch operations (rather than index method), and inline PTX techniques. When considering the interconnection between CPU and GPU, we use multi-copy technology to make full use of the bidirectional bandwidth of the PCIe channel. Finally, we provide a scheme to use CHACHA20 on the CPU/GPU platform. The results show that our work outperforms any previous work.

ACKNOWLEDGMENT

This research has been supported by the China National Key R&D Program during the 13th Five-year Plan Period (Grant No. 2018YFB1700405).

REFERENCES


