Parallel Computing 52 (2016) 106–132

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

A secure and efficient file protecting system based on SHA3 and parallel AES

Xiongwei Fei[a,b], Kenli Li[a,b,*], Wangdong Yang[a,b], Keqin Li[a,b,c]

[a] College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China
[b] National Supercomputing Center in Changsha, Hunan University, Changsha 410082, China
[c] Department of Computer Science, State University of New York, New Paltz, NY 12561, USA

ARTICLE INFO

Article history:
Received 20 August 2015
Revised 23 November 2015
Accepted 3 January 2016
Available online 11 January 2016

Keywords:
Advanced Encryption Standard
Confidentiality
CPU parallelism
Efficiency
GPU parallelism
Integrity

ABSTRACT

There are many private or confidential files stored in computers or transferred on the Internet. People worry and even fear their security problems, such as stealing, breaking, forging, and so on, and urgently need a both secure and highly efficient (for better experience) file protecting system. Thus, we propose and implement a secure and efficient file protecting system (SEFPS) for this demand. SEFPS based on advanced SHA3 (Secure Hash Algorithm 3) and parallel AES (Advance Encryption Standard) can provide the protection of both confidentiality and integrity, and produce high performance by GPU (Graphics Processing Unit) parallelism or/and CPU (Central Processing Unit) parallelism. Correspondingly, SEFPS has three variants for GPU parallelism, CPU parallelism, and both of them. The first variant includes CPU Parallel Protecting (CPP) and CPU Parallel Unprotecting (CPUP). The second variant includes GPU Parallel Protecting (GPP) and GPU Parallel Unprotecting (GPUP). The third variant includes Hybrid Parallel Protecting (HPP) and Hybrid Parallel Unprotecting (HPUP). We design and implement them, and evaluate their performance on two representative platforms by some experiments. HPP and HPUP outperform GPP and GPUP, respectively, and outperform CPP and CPUP more. For those computers not equipped with Nvidia GPUs, CPP and CPUP can be employed, because they still outperform CSP and CSUP, respectively, where CSP and CSUP denote the serial implementation of SEFPS for protecting and for unprotecting, respectively. Moreover, we also prove the security of SEFPS in terms of integrity and confidentiality. Thus, SEFPS is a secure and efficient file protecting system, and can be used in computers no matter whether equipped with Nvidia GPUs or not.

1. Introduction

We are in an era of information explosion now, and we constantly produce, store, or transfer information by computer systems everyday. There is much information that is stored in computers or is transferred on networks as files. If these files are private or confidential, we must protect them against stealing or destroying by security technologies, such as encrypting, tamper-proofing, and so on. More importantly, we must also ensure that the secure technologies should work as fast as possible for our better experience and the pursuit of speed. Especially, for large files, we almost cannot tolerate too much

* Corresponding author at: College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China. Tel.: +86073188664161.
E-mail addresses: feixiongwei@gmail.com, feixiongwei@hnu.edu.cn, feixiongwei@qq.com (X. Fei), lkl@hnu.edu.cn (K. Li), yangwangdong@163.com (W. Yang), lik@newpaltz.edu (K. Li).

http://dx.doi.org/10.1016/j.parco.2016.01.001
0167-8191/

time spent on protecting them, but such files will be more general. Thus, on one side we should protect our private or confidential files by secure technologies inevitably; on the other side we should improve the performance of these technologies as high as possible.

Fortunately, CPUs (Central Processing Units) have multiple cores and GPUs (Graphics Processing Units) have many cores now. GPUs are more suitable for efficiently dealing compute-intensive tasks, such as SpMV (Sparse Matrix-Vector multiplication) [1,2]. Data encryption is also a compute-intensive task due to complex operations. Naturally, GPUs are also suitable for efficient encrypting, because GPUs support integer computations, which are main operations of encrypting, and CUDA (Computing Unified Device Architecture) framework offered by Nvidia makes parallel programming on GPUs easily [3,4]. Moreover, a GPU has more and cheaper cores than a CPU and these cores are good at the encrypting operations without too much control. This makes it a good scheme to improve encrypting performance by GPU parallelism. Without loss of generality, for those computers not equipped with Nvidia GPUs, it can still improve encrypting performance by CPU parallelism.

For encrypting technologies, AES (Advance Encrypting Standard) [5,6] is widely used as a standard for encrypting or decrypting data by enterprises and organizations due to more security and higher efficiency than its competitors, such as RC6. AES is a symmetry cryptography and it encrypts/decrypts data as groups of 16-byte. These groups do not depend on each other in ECB (Electronic Code Book) or CTR (Counter) mode, but share the common key. Thus it can be executed in GPUs or CPUs in parallel by means of many or multiple threads.

For tamper-proofing technologies, message digest algorithms can check whether the information has been altered. Message digest algorithms are based on hash (one way) functions. One can compute a message digest easily and quickly, but it is hardly possible to get the original message just knowing the message digest or to find a conflict of two different messages which have the same message digest. Thus, one can match the old message digest and new computing message digest to identify whether the message has been changed. If the two message digests are equal, this states that the message has not been altered; otherwise the message must have been changed. This is a suitable method of obtaining encrypting and tamper-proofing by combining encrypting algorithms and message digest algorithms. That is to say, we compute the message digest of plaintext first, and then encrypt both of them. Thus, we can keep the plaintext secret and identify any modifying attacks on data as well.

The message digest algorithms mainly include MD (Message Digest) series (such as MD5 [7]) and SHA (Secure Hash Algorithm [8]) series (such as SHA-0). The successful attacks on MD5 and SHA-0 [9], and theoretical attacks on SHA-1 [10] and SHA-2 [11], trigger the demand of securer SHA3. Currently, NIST chooses Keccak as the winner of NIST hash function competition [12], namely SHA3 (Secure Hash Algorithm 3). Despite higher security, SHA3 also has better performance, whose authors declared that it works in 12.5 cpb (cycles per byte) on an Intel Core2 CPU. So we have enough reasons to choose it to implement our fast file protecting system.

However, some existing works, such as [13–17], etc., just consider the improvement of AES performance by CPU or GPU parallelism. Whereas, some other existing works, such as [10,18–21], and so on, have some weaknesses in protecting files and do not consider the performance. The details can be seen in Section 2. Thus, there is need to develop a file protecting system with both high performance and security. Our work in this paper is to achieve such a system by combining the parallel techniques and security techniques. In summary, our contributions are listed as follows.

- We propose a secure and efficient file protecting system (SEFPS) that uses currently excellent SHA3 and parallel AES.
- We prove the security of SEFPS in terms of confidentiality and integrity.
- We design and implement the three variants of SEFPS adopting GPU parallelism or/and CPU parallelism.
- We evaluate the SEFPS' performance on two representative platforms.

The rest of this paper is organized as follows. In Section 2, we review some related work on parallel AES algorithms, SHA algorithms, and file protecting systems. In Section 3, we present the algorithms of our file protecting system. In Section 4, we prove the security of our file protecting system. Next, in Section 5, we describe our experimental environment and methods. In Sections 6 and 7, we give and discuss the experimental results on two platforms, and then we evaluate the performance of our file protecting system. In Section 8, we make a conclusion for all the work and look forward to our future work.

2. Related work

Performance is a forever topic in computer fields, not to speak of encrypting or decrypting, for the pursuit of speed and better experiences. But the speed is limited by the frequency of CPUs because of high power consumption and poor reliability [22]. Thus, people have to change the direction of accelerating speed to more cores. In the past decade, CPUs have developed to multiple cores and GPUs have evolved to many cores with general-purpose computing. Many applications have obtained the gain of CPU or GPU cores. For example, Li et al. [23] optimized the 0–1 knapsack problem using CPU and GPU parallelism; Shi et al. [24] parallelized a face contour detecting application using CPUs and GPUs also; Tan et al. [25] quantified the interplay between energy efficiency and resilience for large-scale high-performance computing systems with many cores. As a frequently used application, AES also enjoys the benefits. Many researchers have improved the efficiency of AES by using GPU parallelism or/and CPU parallelism.

Manavski first [13] used CUDA to parallelize AES on GPUs and achieved about 20 times speedup. Iwai et al. [14] implemented parallel AES by use of CUDA on a GPU and analyzed the effectiveness of AES implementation from the conditions of parallel processing granularity, memory allocation, etc. They found that the granularity at 16 bytes/thread, and the shared

memory of allocating T table and round keys are to be effective. Maistri et al. [15] also implemented parallel AES using CUDA and got good performance with excellent price/performance ratio. And some further analysis and optimization about parallel AES on GPUs can be found in [26–28], and [29].

If a computer is not equipped with GPUs, another possible parallelism is to use CPUs in parallel. AES of course can be parallelized on CPUs by use of their multiple threads. This attracts the attention of researchers as well. For instance, Duta et al. [16] parallelized AES using CUDA, OpenMP, and OpenCL, respectively, and found that their performance descended by CUDA, OpenCL, and OpenMP. Pousa et al. [17] implemented parallel AES algorithm by using OpenMP, MPI, and CUDA, respectively, and the last one also outputs higher efficiency. These researches prove that parallelizing AES by CUDA is a strong and effective method. Moreover, some other researches implemented parallel AES using OpenMP in [30,31], and so on.

Another research direction of accelerating AES is employing hardware. Hossain et al. [32] presented a very low power and high throughput AES processor on the FPGA (Field Programmable Gate Array) platform. Moh'd et al. [33] presented an FPGA architecture of AES-512 algorithm for higher security and evaluated its performance. Banu et al. [34] accelerated AES by combining FPGA and OpenMP. Currently, Liu and Baas [35] parallelized AES on Asynchronous Array of Simple Processors and gained better performance with higher energy efficiency. The hardware parallelism techniques exhibit higher efficiency than the software techniques, but cannot avoid the shortcoming of lacking of flexibility of implementation and spending extra investment.

In a word, for a computer equipped with Nvidia GPUs, parallelizing AES by GPUs is a better method; otherwise by CPUs is still a good method.

As a useful security technique, SHA continually absorbs the attention of researches. Malik et al. [36] evaluated the performance of the software implementation of Keccak (the winner of SHA3) on two platforms (i.e., a resource-efficient platform and a resource-constrained platform). Bayat-Sarmadi et al. [37] presented and designed an efficient concurrent error detection scheme for Keccak. Through ASIC analysis, it shows acceptable complexity and performance overhead. Moreira et al. [38] proposed the utilization of new SHA3 based MAC (Message Authentication Code) and compared with both AES-128 and SHA3 hardware implementations in the context of PTP (Precision Time Protocol) networks. Dong et al. [39] proposed an RFID (radio frequency identification) mutual authentication protocol on the Keccak algorithm for its security and lightweight properties.

As a widely used application, file (data) protecting has fascinated many researches' attention as well. Chen and Lee [18] proposed and implemented a scheme which can offer integrity protection to outsourced data in cloud storage. Yandji et al. [19] proposed a file encryption and decryption system by AES and MD5, but AES works sequentially and MD5 still has some weakness [9]; moreover it deals some information in plaintext that will result in some possible attacks. Biglaril et al. [40] presented a high throughput AES encryption/decryption based on FPGA, which must update existing computers for applying it. JunLi et al. [20] combined the characteristics of AES and ECC (Elliptic Curve Cryptosystem) and designed a mixed email encryption system. Obviously, it does not protect the integrity of the data and would be attacked by tampering. Zhang and Jin [21] developed an encryption system by mixing DES and SHA-1, which provides the security and integrity of data to a certain degree. But because of the weakness of DES and SHA-1 [10], this system will be attacked potentially.

In the above researches, they all did not consider the protecting of integrity and improvement of performance in parallel by exiting computing resources, such as GPUs or CPUs. Thus, there is need to combine the parallel techniques and security techniques to develop a file encryption/decryption system. Our work in this paper is to achieve a file protecting system with both high security and efficiency.

3. File protecting system algorithms

In this section, we will propose a file protecting system based on SHA3 and parallel AES. Because it can achieve the effect of high security and efficiency, we name it as SEFPS for the sake of conciseness. Its securities include integrity and confidentiality, while its efficiency is mainly achieved by the parallel encrypting or decrypting. We propose both the CPU parallel or/and GPU parallel implementation for general purpose. In Section 3.1, we present the work flows for the system. In Section 3.2, we will propose the CPU parallel algorithms for the system, while the GPU parallel algorithms will be put forward in Section 3.3 and the hybrid parallel algorithms will be proposed in Section 3.4.

3.1. Flowchart

The work flows of SEFPS consist of protecting and unprotecting. For protecting, it executes integrity protecting by SHA3 first, and then performs parallel encrypting by parallel AES. As for unprotecting, it performs parallel decrypting by parallel AES, and then executes integrity checking by SHA3 for tamper-proofing. All types of files can be protected by SEFPS, because it deals files as binary format.

The work flow of protecting can be divided into three main steps as demonstrated in Fig. 1. The first step is padding some bytes to ensure that the total file has the length of multiple of 16 bytes. This is because the data in file will be encrypted in groups of 16 bytes in the latter encrypting phrase. The padding method is computing the number of padded bytes as $32 - X\%16$ firstly, where X represents the number of bytes of this file and $X\%16$ means the remainder of X exactly dividing 16, and then stuffing 0 to the previous $16 - X\%16$ bytes and $16 - X\%16$ to the next 16 bytes. The stuffing of 0 makes

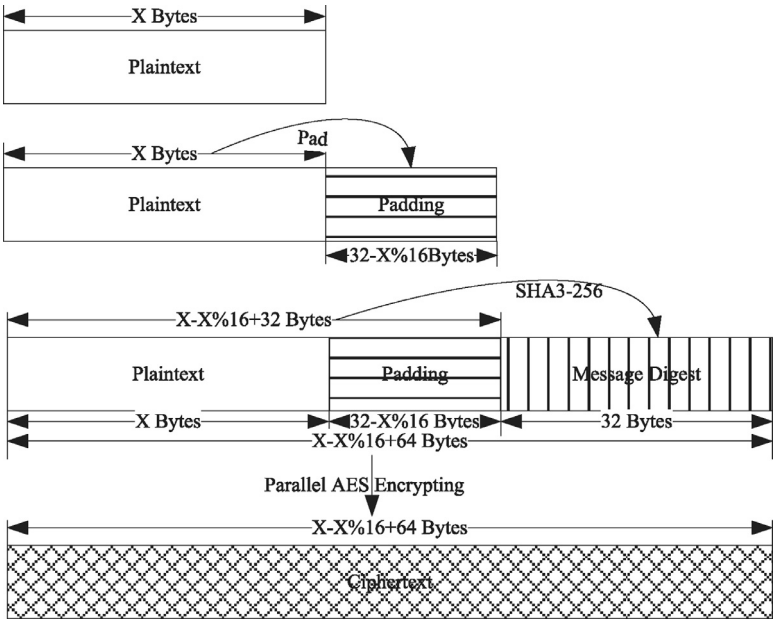

Fig. 1. Work flow of protecting. A file to be protected is first padded some bytes to ensure that its length is a multiple of 16 bytes. Then this padded file is hashed by SHA3 and then attached the corresponding message digest. Next, this file is encrypted by parallel AES. After this, the cipher file has been protected in encrypting and tamper-proofing and can be stored in a hard disk or transferred on networks safely.

sure that the file is enlarged to multiple bytes of 16, while the stuffing of $16 - X\%16$ expresses how many bytes have been stuffed in the previous group of 16 bytes.

For example, if a file has 123 bytes, it should be padded 21 bytes totally to make its length (144 bytes) a multiple of 16 bytes, i.e., stuffing 5 bytes with 0 and 16 bytes with the length value of 5. The stuffed data will be used to recover the original file in unprotecting.

After this, the padded file will be hashed by SHA3-256 and attached the message digest in the end. The reason for choosing SHA3-256 is that it compromises performance and security and its message digest is as long as 32 bytes (twice of 16 bytes). Next, this file will be encrypted by parallel AES for high performance in step three. When these three steps are finished, the cipher file has been protected by means of confidentiality (encrypting) and integrity (message digest), and can be stored in hard disk or transferred on network safely. At the same time, because encrypting works in parallel, SEFPS also produces high performance.

The work flow of unprotecting comprises three main steps too as exhibited in Fig. 2, but has different operations and order. The first step works as decrypting the ciphertext file by parallel AES and getting the corresponding plaintext file. In the plaintext file, the last 32 bytes data is the message digest. As demonstrated in Fig. 2, step two uses SHA3-256 to hash the plaintext except the message digest and gets a new message digest, and then compares this new message digest with the attached message digest. For instance, if the ciphertext has Y bytes, step two hashes the previous $Y - 32$ bytes by SHA3-256 and gets a new 32 bytes message digest.

If these two MDs are equal, the flow will continue to step three; otherwise it will be ended and a message of "The data may be altered or decrypted with wrong password" is reported to warn users. If the match is successful, this plaintext file will be deleted the last attached message (32 bytes) and the padded bytes in step three. The padded bytes are computed as $16 + Plaintext[Y - 32]$, where $Plaintext$ represents the base address of plaintext. Thus, $Plaintext[Y - 32]$ means the number of bytes padded and stuffed with 0, as stating in work flow of protecting. Naturally, $16 + Plaintext[Y - 32]$ represents the total bytes padded to plaintext file. After these three steps, the plaitext file has been recovered to the original file.

3.2. CPU parallel algorithms

CPU parallel algorithms of SEFPS use multiple threads of CPUs to execute AES in parallel for high performance. According to the work flows described in Section 3.1, there are two algorithms that take charge of protecting and unprotecting, respectively. For conciseness, we name the former as CPU Parallel Protecting (CPP) algorithm of SEFPS and the latter as CPU Parallel Unprotecting (CPUP) algorithm of SEFPS. We will describe CPP in Section 3.2.1 and CPUP in Section 3.2.2 in detail.

3.2.1. CPP

The details of CPP can be seen in Algorithm 1. CPP requires the file name and the key as input. It opens this file (*pPlainFile*) in binary format for supporting any type of files.

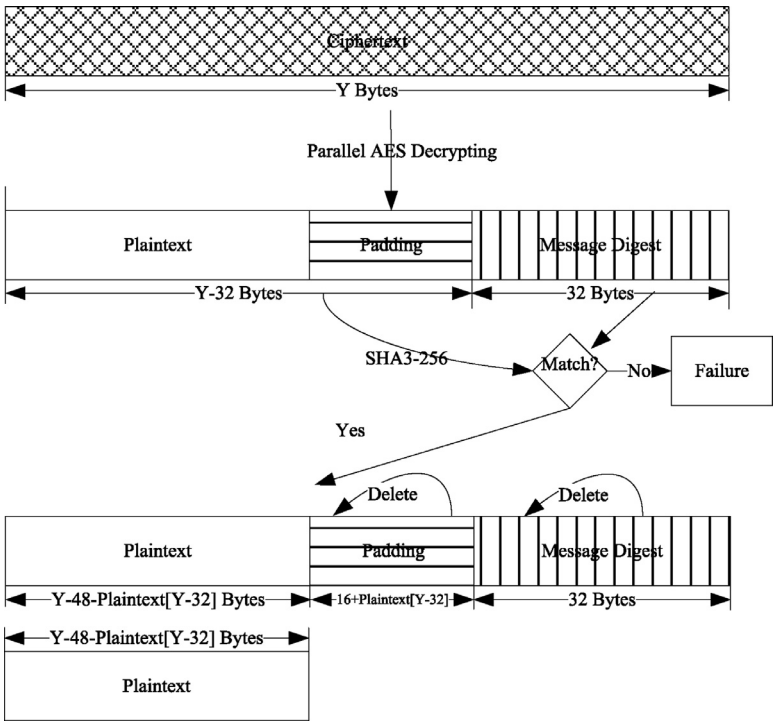

Fig. 2. Work flow of unprotecting. The protected file is firstly decrypted by parallel AES. Then this plaintext file except the message digest (the last 32 bytes) is hashed by SHA3 and then the new message digest will be compared with the message digest attached in the plaintext file. If these two message digests are equal, then the third step will continue; otherwise the process will be ended and a message is reported to warn users. If step two is okay, this plaintext file will be deleted the attached message bytes (32 bytes) and the padded bytes in step three. After this, the plaintext file has been recovered to the original file.

After obtaining the length of *pPlainFile* as *fileLen*, it calculates the total length of protected file as *plainLen* by means of Section 3.1. Next, it allocates *plainLen* bytes space and sets them to 0, which implies padding some bytes with 0. Then it reads data from *pPlainFile* in block with size of *IO_BLOCK* as shown in Line 9, where *IO_BLOCK* represents how many bytes data is read in a time. Of course, the remainder bytes just are read one byte by one byte as Line 11, if they exist.

In Line 13, it assigns the value, which represents the number of bytes padded in the previous group, to the last group of padded data. In Line 14, it executes SHA3-256 to hash the plaintext with the padded bytes and gets the corresponding message digest *hashValue*.

In Lines 16–31, it evenly divides the total bytes (including plaintext, padded bytes, and message digest) to *WORK_THREADS* threads of CPUs, where *WORK_THREADS* represents the number of threads of CPUs. The dividing method is to compute the length of each thread as an array *PtLenCPU* and the location of each thread as an array *StLocCPU*. In Lines 32–36, it uses OpenMP directions to drive *WORK_THREAD* threads, which perform AES encrypting by the arranged workload in parallel. The AES encrypting employs *T* lookup tables to accelerate.

After this, it writes the cipher to *pPlainFile* with new length *plainLen* in blocks. Thus, we get a corresponding cipher file, which has been protected within integrity and confidentiality.

3.2.2. CPUP

Algorithm 2 demonstrates the details of CPUP. CPUP requires the name of the file to be unprotected and the key as input. It unprotects the protected file with corresponding key to get the original file.

For stressing the key points, we omit some details and emphasize on the difference of CPUP with CPP, because readers can consult the description of CPP in Section 3.2.1. It first decrypts the file by CPU parallel AES as shown in Lines 7–27, and then verifies the integrity with SHA3-256 as shown in Lines 28–34. In Line 28, it computes the message digest as *hashValue*, and in Line 29, it compares *hashValue* with the last 32 bytes of *cipher*, which is the old message digest computed in protecting file. If these two message digests can match, it deletes the message digest, pads bytes by shortening *cipherLen* in Line 30, and overwrites *pCipherFile* to get the original file. If they do not match, it reports a message of “The data may be altered or decrypted with wrong password.” to warn users and then exits.

All in all, if legal users execute CPUP with the right password, they will get the original file; otherwise they will fail.

Algorithm 1 CPU Parallel Protecting algorithm of SEFPS.

Require:
the name of file to be protected, $fileName$;
the key of AES, key;
Ensure:
1: int $hashByteLen = 32$;
2: FILE *$pPlainFile$ = fopen($fileName$,"wb+");
3: size_t $fileLen$ = ftell($pPlainFile$);
4: size_t $plainLen$=$fileLen + 16 + (16 - fileLen\%16) + hashByteLen$;
5: unsigned char *$hashValue$=(unsigned char*)malloc($hashByteLen$);
6: unsigned char * $plain$= (unsigned char *)malloc($plainLen$);
7: memset($plain$, 0, $plainLen$);
8: size_t $ioBlocks = fileLen/IO\_BLOCK$, $rm = fileLen\%IO\_BLOCK$;
9: fread($plain$, IO_BLOCK, $ioBlocks$, $pPlainFile$);
10: **if** rm!=0 **then**
11: fread($plain + ioBlocks * IO\_BLOCK$, rm, 1, $pPlainFile$);
12: **end if**
13: memset($plain + plainLen - hashByteLen - 16$, $16 - fileLen\%16$, 16);
14: Hash($hashByteLen * 8$, $plain$, $(plainLen - hashByteLen) * 8$, $hashValue$);//hash the plaintext file with padded bytes by SHA3-256;
15: memcpy($plain + plainLen - hashByteLen$, $hashValue$, $hashByteLen$);
16: unsigned int $PtLenCPU[WORK\_THREADS]$;
17: unsigned char *$StLocCPU[WORK\_THREADS]$;
18: unsigned int $rmd = plainLen/16\%(WORK\_THREADS)$;
19: unsigned int $MeanPtLen = plainLen/16/(WORK\_THREADS) * 16$;
20: $PtLenCPU[0] = MeanPtLen$;
21: $StLocCPU[0] = plain$;
22: **for** int $i = 0; i < WORK\_THREADS; i++$ **do**
23: **if** $rmd > i$ **then**
24: $PtLenCPU[i] = MeanPtLen + 16$;
25: **else**
26: $PtLenCPU[i] = MeanPtLen$;
27: **end if**
28: **end for**
29: **for** int $i = 1; i < WORK\_THREADS; i++$ **do**
30: $StLocCPU[i] = StLocCPU[i-1] + PtLenCPU[i-1]$;
31: **end for**
32: #pragma omp parallel num_threads($WORK\_THREADS$)
33: #pragma omp for
34: **for** int $i = 0; i < WORK\_THREADS; i++$ **do**
35: AesTTableUnroll($StLocCPU[i]$, $PtLenCPU[i]$, key);
//execute AES encrypting by T table;
36: **end for**
37: rewind $pPlainFile$ and write $plain$ to $pPlainFile$ in blocks.

3.3. GPU parallel algorithms

GPU parallel algorithms of SEFPS use many threads of GPUs to perform AES in parallel for high performance without extra investment. Similarly to the previous Section 3.2, there are also two algorithms that take charge of protecting and unprotecting, respectively. We name the former as GPU Parallel Protecting (GPP) algorithm of SEFPS and the latter as GPU Parallel Unprotecting (GPUP) algorithm of SEFPS for the sake of simplicity. The details of GPP are described in Section 3.3.1 and GPUP in Section 3.3.2.

3.3.1. GPP

Algorithm 3 demonstrates the details of GPP. GPP requires the name of the file to be unprotected and the key as input. It protects this file with the key for storing or transferring safely.

Similarly, we omit some details and emphasize on the difference of GPP with CPP. GPP first reads the file in block and pads some bytes as shown in Lines 2–13, and then computes the message digest *hashValue* with SHA3-256 as shown in Line 14. Next the file with the padded bytes and the message digest will be encrypted on GPUs in parallel. Firstly, the key

Algorithm 2 CPU Parallel Unprotecting algorithm of SEFPS.

Require:
the name of file to be unprotected, $fileName$;
the key of AES, key;
Ensure:
1: int $hashByteLen$ =32;
2: FILE *$pCipherFile$ = fopen($fileName$,"wb+");
3: size_t $CipherLen$=ftell($pCipherFile$);
4: unsigned char *$hashValue$=(unsigned char*)malloc($hashByteLen$);
5: unsigned char * $cipher$= (unsigned char *)malloc($cipherLen$);
6: read data from file $pCipherFile$ to $cipher$ in blocks;
7: unsigned int $CtLenCPU[WORK\_THREADS]$;
8: unsigned char *$StLocCPU[WORK\_THREADS]$;
9: unsigned int $rmd = cipherLen/16\%(WORK\_THREADS)$;
10: unsigned int $MeanCtLen = cipherLen/16/(WORK\_THREADS) * 16$;
11: $CtLenCPU[0] = MeanCtLen$;
12: $StLocCPU[0] = cipher$;
13: **for** int $i = 0; i < WORK\_THREADS; i++$ **do**
14: **if** $rmd > i$ **then**
15: $StLenCPU[i] = MeanCtLen + 16$;
16: **else**
17: $StLenCPU[i] = MeanCtLen$;
18: **end if**
19: **end for**
20: **for** int $i = 1; i < WORK\_THREADS; i++$ **do**
21: $StLocCPU[i] = StLocCPU[i-1] + CtLenCPU[i-1]$;
22: **end for**
23: #pragma omp parallel num_threads($WORK\_THREADS$)
24: #pragma omp for
25: **for** int $i = 0; i < WORK\_THREADS; i++$ **do**
26: AesInvTTableUnroll($StLocCPU[i], CtLenCPU[i]$, key);//execute AES decrypting by T table
27: **end for**
28: Hash($hashByteLen * 8, cipher, (cipherLen - hashByteLen) * 8, hashValue$);//hash the plaintext file with padded bytes by SHA3-256;
29: **if** $hashValue$ is equal to the last 32 bytes of $cipher$ **then**
30: $cipherLen = cipherLen - hashByteLen - 16 - cipher[cipherLen - 1]$;//delete message digest and padded bytes by shortening $cipherLen$
31: rewind $pCipherFile$ and write $cipherLen$ byte from $cipher$ to $pCipherFile$ in blocks.
32: **else**
33: report "The data may be altered or decrypted with wrong password" and exit.
34: **end if**

is extended in serial as shown in Line 16. Then, GPP defines and allocates device space to store plaintext *plain*, extended key *exKey*, and lookup tables *T*. In Lines 18–24, GPP arranges the organization of threads on GPUs. The dimension of block *dmblock* is *BK*, where *BK* is multiple of 64 for high thread occupancy. Then, GPP computes the number of blocks *blocks* of plaintext *plain* in a thread corresponding a group of 16 bytes, and arranges a one-dimensional grid of *blocks*. If *blocks* is larger than or equal to 65 536, i.e., the limitation of dimension, GPP uses two-dimensional grid, which is calculated in Lines 22 and 23.

After transferring the data from main memory to global memory in device as shown in Line 25, the kernel function is called to execute AES in parallel as organization as the thread grid. When the kernel function finished, GPP copies *dinput* back to *plain*, and then writes to *pPlainFile*, which is the protected file.

3.3.2. GPUP

Algorithm 4 shows the details of GPUP. GPUP requires the name of the file to be unprotected and the key as input. It aims to unprotect this file to recover the original file.

For conciseness, we will emphasize on the difference of GPUP with GPP. GPUP first decrypts the file by GPU parallel AES as shown in Lines 7–19, and then verifies the integrity with SHA3-256 as shown in Lines 20–26. For the AES decryption, it first executes a different key extending function *InvKeyExpansion*, and transfers a different sbox *invBox* and a different

Algorithm 3 GPU Parallel Protecting algorithm of SEFPS.

Require:
the name of file to be protected, $fileName$;
the key of AES, key;
Ensure:
1: int $hashByteLen$ =32;
2: FILE *$pPlainFile$ = fopen($fileName$,"wb+");
3: size_t $fileLen$ = ftell($pPlainFile$);
4: size_t $plainLen$=$fileLen$ + 16 + (16 − $fileLen$%16) + $hashByteLen$;
5: unsigned char *$hashValue$=(unsigned char*)malloc($hashByteLen$);
6: unsigned char * $plain$= (unsigned char *)malloc($plainLen$);
7: memset($plain$, 0, $plainLen$);
8: size_t $ioBlocks$ = $fileLen$/IO_BLOCK, rm = $fileLen$%IO_BLOCK;
9: fread($plain$, IO_BLOCK, $ioBlocks$, $pPlainFile$);
10: **if** rm!=0 **then**
11: fread($plain$ + $ioBlocks$ ∗ IO_BLOCK, rm, 1, $pPlainFile$);
12: **end if**
13: memset($plain$ + $plainLen$ − $hashByteLen$ − 16, 16 − $fileLen$%16, 16);
14: Hash($hashByteLen$ ∗ 8, $plain$, ($plainLen$ − $hashByteLen$) ∗ 8, $hashValue$);//hash the plaintext file with padding bytes by SHA3-256;
15: memcpy($plain$ + $plainLen$ − $hashByteLen$, $hashValue$, $hashByteLen$);
16: extend key to get $exKey$;
17: define $dinput$, $dkey$, dT to store $plain$,$exkey$ and T tables respectively and allocate space for them in device memory;
18: dim3 $dmblock(BK)$; //block size
19: unsigned int $blocks$ = (($plainLen$ >> 4) + BK − 1)/(BK);
20: dim3 $dmgrid(blocks)$; //num of blocks
21: **if** $blocks$ >= (1 << 16) **then**
22: $dmgrid.x$ = (1 << 15);
23: $dmgrid.y$ = ($blocks$ + (1 << 15) − 1) >> 15;
24: **end if**
25: copy data from $plain$, $exKey$, T tables to $dinput$, $dkey$, dT respectively;
26: CipherTTable<<< $dmgrid$, $dmblock$ >>>
($dinput$, $dkey$, dT, $plainLen$);//execute AES encrypting on the GPU in parallel;
27: copy data from $dinput$ to $plain$;
28: release the allocated memory in device;
29: rewind $pPlainFile$ and write $plain$ to $pPlainFile$ in blocks.

lookup table $invT$, whereas in GPP, $sbox$ can be gotten by T table. Of course, the kernel function $invCipherTTable$ executes decrypting in parallel and is accelerated by $invT$ table.

After these, if legal users execute GPUP with the right password, they will get the original file; otherwise they will fail.

3.4. Hybrid parallel algorithms

Hybrid parallel algorithms of SEFPS use many threads of GPUs and multiple threads of CPUs simultaneously to perform AES for higher performance. There are two algorithms that take charge of protecting and unprotecting, respectively, too. We name the former as Hybrid Parallel Protecting (HPP) algorithm of SEFPS and the latter as Hybrid Parallel Unprotecting (HPUP) algorithm of SEFPS for the sake of simplicity. The details of HPP and HPUP are described in Sections 3.4.1 and 3.4.2, respectively.

3.4.1. HPP

Algorithm 5 demonstrates the details of HPP. HPP adopts the programming model of CUDA + OpenMP [41]. In the model, CPUs derive multiple threads using OpenMP, one of which is used to transfer a part of a file and call the kernel and the others of which are used to protect the rest part of the file. The kernel will be executed on GPUs in parallel to protect data under the help of CUDA. In this way, CPUs and GPUs can work simultaneously to protect the entire file. HPP requires the name of the file to be unprotected and the key as input. It protects this file with the key for storing or transferring safely.

HPP combines both GPP and CPP in encrypting. Its main difference with GPP and CPP is allocating data from $plain$ in the proportion of $ratio$, which is calculated by the computing capacity of GPUs and CPUs in Line 17. The calculation can be seen in Eq. (1):

$$ratio = \frac{Gmp \times Ws \times Gf}{(Cc - 1) \times Cf}, \tag{1}$$

Algorithm 4 GPU Parallel Unprotecting algorithm of SEFPS.

Require:
the name of file to be unprotected, *fileName*;
the key of AES, *key*;
Ensure:
1: int *hashByteLen* =32;
2: FILE **pCipherFile* = fopen(*fileName*,"wb+");
3: size_t *CipherLen*=ftell(*pCipherFile*);
4: unsigned char **hashValue*=(unsigned char*)malloc(*hashByteLen*);
5: unsigned char * *cipher*= (unsigned char *)malloc(*cipherLen*);
6: read data from file *pCipherFile* to *cipher* in blocks;
7: execute InvKeyExpansion(*key*) to get *invExKey*;
8: define *dinput*, *dkey*, *dT*, *diBox* to store *plain*,*invExkey*, *invT* tables and *invBox* respectively and allocate space for them in device memory;
9: dim3 *dmblock*(*BK*); //block size
10: unsigned int *blocks* = ((*cipherLen* >> 4) + *BK* − 1)/(*BK*);
11: dim3 *dmgrid*(*blocks*); //num of blocks
12: **if** *blocks* >= (1 << 16) **then**
13: *dmgrid.x* = (1 << 15);
14: *dmgrid.y* = (*blocks* + (1 << 15) − 1) >> 15;
15: **end if**
16: copy data from *cipher*, *invExKey*, *invT*, *invBox* tables to *dinput*, *dkey*, *dT*, *diBox* respectivly;
17: InvCipherTTable<<< *dmgrid*, *dmblock* >>>
(*dinput*, *dkey*, *dT*, *diBox*, *cipherLen*)//execute AES decrypting on the GPU in parallel;
18: copy data from *dinput* to *cipher* ;
19: release the allocated memory in device;
20: Hash(*hashByteLen* ∗ 8, *cipher*, (*cipherLen* − *hashByteLen*) ∗ 8, *hashValue*);//hash the plaintext file with padding bytes by SHA3-256;
21: **if** *hashValue* is equal to the last 32 bytes of *cipher* **then**
22: *cipherLen* = *cipherLen* − *hashByteLen* − 16 − *cipher*[*cipherLen* − 1];//delete message digest and padded bytes by shortening *cipherLen*
23: rewind *pCipherFile* and write *cipherLen* byte from *cipher* to *pPlainFile* in blocks.
24: **else**
25: report "The data may be altered or decrypted with wrong password" and exit.
26: **end if**

where *Gmp*, *Ws*, and *Gf* denote the number of multiple processors, warp size, and frequency of GPUs, respectively; *Cc* and *Cf* represent the number of cores and frequency of CPUs, respectively. In Eq. (1), the denominator and the numerator represent the computing capacities of CPUs and GPUs, respectively. Because CPUs derive *Cc* threads and one of them is used to serve GPUs, $(Cc-1)\times Cf$ represents the computing capacities of CPUs. On GPUs, each of multiple processor issues and schedules threads in unit of warp, so $Gmp \times Ws \times Gf$ denotes the computing capacities of GPUs. Thus, *ratio* represents the ratio of the computing capacities of GPUs to CPUs. Lines 18 and 19 calculate the encrypting proportion of GPUs and CPUs by $ratio/(ratio+1)$ and $1/(ratio+1)$, respectively. The *ratio* needs to be calculated by Eq. (1), because a greater or less value of the *ratio* will lead to imbalanced workload on GPUs and CPUs and then increase time, i.e., a greater value will result in more workload on GPUs and a less value will cause more workload on CPUs.

Then, HPP derives *WORK_THREADS* CPU threads, one of which executes the kernel function to encrypt a part of the data on GPUs in parallel as the manner of GPP, while the others of which perform encrypting on CPUs simultaneously as the manner of CPP. These can be seen in Lines 21 and 22, respectively. When all the threads finished, HPP get the corresponding parts of ciphertext from GPUs and CPUs, and then writes them to *pPlainFile*, which is the protected file.

3.4.2. HPUP

Algorithm 6 shows the details of HPUP. HPUP requires the name of the file to be unprotected and the key as input. It aims to unprotect this file to recover the original file for the legal users.

HPUP combines both GPUP and CPUP in the process of decrypting. Its main difference with GPUP and CPUP is allocating data from *cipher* in the proportion of *ratio*, which is calculated by the computing capacity of GPUs and CPUs, as shown in Line 8. Then, it drives *WORK_THREADS* CPU threads, one of which executes the kernel function to decrypt a part of the data *PG* on GPUs in parallel as the manner of GPUP, while the others of which perform decrypting on CPUs simultaneously as the manner of CPUP. These can be seen in Lines 12 and 13, respectively. Then, HPUP will verify the integrity with SHA3-256

Algorithm 5 Hybrid Parallel Protecting algorithm of SEFPS.

Require:
the name of file to be protected, $fileName$;
the key of AES, key;
Ensure:
int $hashByteLen$ =32;
FILE *$pPlainFile$ = fopen($fileName$,"wb+");
size_t $fileLen$ = ftell($pPlainFile$);
size_t $plainLen$=$fileLen$ + 16 + (16 − $fileLen$%16) + $hashByteLen$;
unsigned char *$hashValue$=(unsigned char*)malloc($hashByteLen$);
unsigned char * $plain$= (unsigned char *)malloc($plainLen$);
memset($plain$, 0, $plainLen$);
size_t $ioBlocks$ = $fileLen$/IO_BLOCK, rm = $fileLen$%IO_BLOCK;
fread($plain$, IO_BLOCK, $ioBlocks$, $pPlainFile$);
if rm!=0 **then**
  fread($plain + ioBlocks * IO\_BLOCK$, rm, 1, $pPlainFile$);
end if
memset($plain + plainLen - hashByteLen - 16$, $16 - fileLen$%16, 16);
Hash($hashByteLen * 8$, $plain$, $(plainLen - hashByteLen) * 8$, $hashValue$);//hash the plaintext file with padding bytes by SHA3-256;
memcpy($plain + plainLen - hashByteLen$, $hashValue$, $hashByteLen$);
extend key to get $exKey$;
compute the data proportion $ration$ of allocating to GPUs and CPUs by their computing capacity.
unsigned int PC = plainLen*1.0/(ratio+1)/16*16;//PC is the number of bytes of the data, which will be encrypted on CPUs.
unsigned int PG = plainLen - PC;// PC is the number of bytes of the data, which will be encrypted on GPUs.
derive $WORK\_THREADS$ CPU threads;
one of them drives a kernel to encrypt the data from $plain + PC$ on GPUs in parallel, as the manner of Algorithm 3;
the others execute encrypting the data from $plain$ to $plain + PC - 1$ on CPUs in parallel, as the manner of Algorithm 1;
$pPlainFile$ rewind and write $plain$ to $pPlainFile$ in blocks.

Algorithm 6 Hybrid Parallel Unprotecting algorithm of SEFPS.

Require:
the name of file to be unprotected, $fileName$;
the key of AES, key;
Ensure:
int $hashByteLen$ =32;
FILE *$pCipherFile$ = fopen($fileName$,"wb+");
size_t $CipherLen$=ftell($pCipherFile$);
unsigned char *$hashValue$=(unsigned char*)malloc($hashByteLen$);
unsigned char * $cipher$= (unsigned char *)malloc($cipherLen$);
read data from file $pCipherFile$ to $cipher$ in blocks;
execute InvKeyExpansion(key) to get $invExKey$;
compute the data proportion $ratio$ of allocating to GPUs and CPUs by their computing capacity.
unsigned int PC = CipherLen*1.0/(ratio+1)/16*16;//PC is the number of bytes of the data, which will be decrypted on CPUs.
unsigned int PG = CipherLen - PC;// PC is the number of bytes of the data, which will be decrypted on GPUs.
derive $WORK\_THREADS$ CPU threads;
one of them drives a kernel to decrypt the data from $cipher + PC$ on GPUs in parallel, as the manner of Algorithm 4;
the others execute decrypting the data from $cipher$ to $cipher + PC - 1$ on CPUs in parallel, as the manner of Algorithm 2;
Hash($hashByteLen * 8$, $cipher$, $(cipherLen - hashByteLen) * 8$, $hashValue$);//hash the plaintext file with padding bytes by SHA3-256;
if $hashValue$ is equal to the last 32 bytes of $cipher$ **then**
  $cipherLen = cipherLen - hashByteLen - 16 - cipher[cipherLen - 1]$;//delete message digest and padded bytes by shortening $cipherLen$
  $pCipherFile$ rewind and write $cipherLen$ byte from $cipher$ to $pPlainFile$ in blocks.
else
  report "The data may be altered or decrypted with wrong password" and exit.
end if

as shown in Lines 14–19. As a result, if legal users execute GPUP with the right password, they will get the original file; otherwise they will fail.

4. Security proof

SEFPS can provide the securities of integrity and confidentiality at the same time. The integrity can provide tamper-proofing, and the confidentiality can protect data from disclosure. Now we give the proof of these two security. The integrity proof is given in Section 4.1, while confidentiality is proved in Section 4.2.

4.1. Integrity proof

Integrity should ensure that any change of data will be found. Adversaries may destroy data integrity by deleting some data, altering some data, or adding some data, etc., and make the data different from the original data as a result. Naturally, once the data have lost integrity, the legal users will get the wrong data and may suffer some damage or lost.

SEFPS can provide integrity protecting. We will prove that it can survive from three attacks, namely, altering attack, adding attack, and deleting attack. We assume that a file m is protected by SEFPS as $c(m)||c(d)$, where $c(m)$ denotes the cipher of m under AES, while $c(d)$ represents the cipher of MD d of m. We use h to express the hash function (SHA3-256), i.e., $d = h(m)$.

- *Altering attack:* In this attack, adversaries may destroy the protected data $c(m)||c(d)$ by altering $c(m)$, $c(d)$, or both as:

$$alter(c(m)||c(d)) = \begin{cases} c(m)'||c(d) \\ c(m)||c(d)' \\ c(m)'||c(d)' \end{cases}. \tag{2}$$

  For $alter(c(m)||c(d)) = c(m)'||c(d)$, legal users will get the wrong plaintext m' by using the right password to decrypt it, and then they will fail to match the message digests, because $h(m') \neq h(m) = d$. Thus, the legal users can identify this behavior of altering data and will not believe this file. Adversaries just know $c(m)$ and $c(d)$, so they do not know m and d. Thus, they cannot forge m' such that $h(m') = d$.
  For $alter(c(m)||c(d)) = c(m)||c(d)'$, legal users will get the correct plaintext m but the wrong message digest d' by using a right password to decrypt it, and then they will fail to match the message digests, because $h(m) = d \neq d'$. The legal users can also identify this behavior of altering data and will avoid possible loss. Because adversaries just know $c(d)$ and $c(m)$, they cannot know d and m. Thus, they do not find $c(d)'$ such that $c(d)' \neq c(d) \bigwedge d' = h(m)$, because SHA3 can resist conflict.
  For $alter(c(m)||c(d)) = c(m)'||c(d)'$, because adversaries just know $c(d)$ and $c(m)$, they do not know d and m. They change $c(m)$ to $c(m)'$ by encrypting m' and alter $c(d)$ to $c(d)'$ by encrypting $h(c(m)')$, where $h(c(m)')$ is obtained by hashing $c(m)'$. They do not know the correct key of legal users, thus legal users will get wrong plaintext m'' and message digest d'' by using a right password to decrypt, obviously $m'' \neq m' \bigwedge d'' \neq d'$. Now $h(m'')$ is the message digest of m'', but d'' is not the message digest of m'', thus $h(m'') \neq d''$ because the resistance of conflict of hash function. In this situation, the legal users can identify the data has been changed because they can find the mismatch of the message digests.
  All in all, the attacks of altering data will be found by legal users and cannot forge the suitable altering to cheat them without the right key. That is to say, SEFPS can protect the data integrity from altering attacks.
- *Deleting attacks:* In these attacks, adversaries attack the data by deleting some data. The deleted data may locate in ciphertext part $c(m)$, encrypted message digest part $c(d)$, or both as:

$$delete(c(m)||c(d)) = \begin{cases} c(m)'||c(d) \\ c(m)||c(d)' \\ c(m)'||c(d)' \end{cases}. \tag{3}$$

  For $delete(c(m)||c(d)) = c(m)'||c(d)$, this means that the legal users will get shortened plaintext m' but the correct message digest d. Because $d = h(c(m)) \neq h(c(m'))$, the legal users will identify that the data have been changed. Moreover, adversaries do not know the true message digest d, so they cannot selectively delete some data from $c(m)$ for attempting to forge the message digest which matches the shortened data. In this situation, the legal users can identify this behavior of deleting data.
  For $delete(c(m)||c(d)) = c(m)||c(d)'$, this means that the legal users will understand the data as shortened ciphertext m' and an attached message digest d' after encrypting. Because d' does not correspond the message digest of m', i.e., $h(m') \neq d'$. In this situation, the legal users can recognize this behavior as well.
  For $delete(c(m)||c(d)) = c(m)'||c(d)'$, the adversaries delete some data both from $c(m)$ and from $c(d)$. This means that the legal users will get a shortened plaintext m' and a different message digest d'. Because d' is not produced by hashing m', i.e., $d' \neq h(m')$. In this situation, the legal users can also recognize the behavior of deleting data.
  All in all, the attacks of deleting data will be found by legal users and cannot forge the suitable altering to cheat them without the right key. That is to say, SEFPS can protect the data integrity against deleting attacks.

- *Adding attacks:* In these attacks, adversaries modify the data by adding some data. The added data may locate in ciphertext part $c(m)$, encrypted message digest part $c(d)$, or both as:

$$add(c(m)||c(d)) = \begin{cases} c(m)'||c(d) \\ c(m)||c(d)' \\ c(m)'||c(d)' \end{cases} . \tag{4}$$

For $add(c(m)||c(d)) = c(m)'||c(d)$, this means that the legal users will get growing plaintext m' and the correct message digest d. Because $h(m') \neq h(m) = d$, the legal users can be aware of this behavior of adding data.
For $add(c(m)||c(d)) = c(m)||c(d)'$, this means that the legal users will understand the data as increasing ciphertext m' and an attached message digest d'. Because d' is not the message digest of m', i.e., $h(m') \neq d'$. Thus, the legal users can sense this behavior of adding some bytes of message digest too.
For $add(c(m)||c(d)) = c(m)'||c(d)'$, the legal users will get growing plaintext m' and a different message digest d'. Because d' is not the message digest of m', i.e., $h(m') \neq d'$. Thus, the legal users can also sense this behavior of adding data.
All in all, the attacks of adding data will be found by legal users and cannot forge the suitable adding to cheat them without the right key. In other words, SEFPS can protect the data integrity against adding attacks.

4.2. Confidentiality proof

In SEFPS, the confidentiality is provided by parallel AES. Daemen and Rijmen [5] proved that the Rijndael (AES) cipher can provide the security goals of K-secure and Hermetic, whose strength can survive from any known or unknown attacks with the given dimensions. That is why AES is chosen by NIST as a standard encrypting/decrypting algorithm. In the process of protecting files, SEFPS encrypts the data with padded bytes and message digest in parallel. Because each group of 16 bytes can be encrypted independently in ECB or CTR mode, parallel AES works with high performance and keeps the same strong security as the serial AES. In a word, SEFPS efficiently and securely encrypts the data and the message digest which is crucial to implement integrity by SHA3. Thus, SEFPS has confidentiality in protecting the data and the message digest with high performance.

5. Experiment design

We perform some experiments to assess SEFPS on two typical experimental platforms with some files of different sizes. In Section 5.1, we will introduce the experimental environment, including platform configuration, files, and so on. In Section 5.2, we give the process and method of the experiments.

5.1. Experimental environment

The experiments are performed on two platforms with 14 files of different sizes. The first one is a server equipped with special computing card K20M, representing high-end computers. The second one is a computer with general computing card GTX 460, delegating general computers. If SEFPS works well in these platforms, it can be said that SEFPS is universal for users. Their configuration can be seen in Table 1. The files employed in the experiments are listed in Table 2, which have different file types and sizes.

5.2. Experimental method

For the sake of comparing, the experiments not only perform all GPP, GPUP, CPP, CPUP, HPP, and HPUP, but also execute the Serial implementation of SEFPS for Protecting (CSP) files on CPUs and the Serial implementation of SEFPS for Unprotecting (CSUP) files on CPUs.

Without losing generality, these experiments are performed on Platforms 1 and 2 orderly. The steps of the experiments are listed as follows.

Table 1
Configurations of two experimental platforms.

Numbers	GPU	CPU	Hard disk
1	NVIDIA Tesla K20M (13 Multiprocessors) Clock rate: 706 MHz Total: 2496 Cores	2 Intel Xeon E5-2640 V2 (8 Cores) Clock rate: 2.0 GHz Total: 16 Cores	HITACHI HUA723020ALA640 (2TB) Standard: SATA 3 Cache Size: 64 MB
2	NVIDIA Geforce GTX 460 (7 Multiprocessors) Clock rate: 1500 MHz Total: 336 Cores	2 Intel Xeon L5506 (4 Cores) Clock rate: 2.13 GHz Total: 8 Cores	SEAGATE ST31000528AS (1TB) Standard: SATA 2.6 Cache Size: 0 MB

Table 2
Files used in experiments.

Numbers	File name	Size
1	out.cls	5.61 KB
2	xdd.xlsx	20.4 KB
3	kalman.doc	255 KB
4	File1.phy	2.21 MB
5	vc9.lib	12.5 MB
6	AES256.sdf	34.8 MB
7	Cryptography.pdf	52.9 MB
8	Hotel.ppt	103 MB
9	NVIDIA_Nsight.msi	191 MB
10	blobs.bin	278 MB
11	MCRInstaller.exe	352 MB
12	celebration.avi	470 MB
13	2011travel.rar	500 MB
14	acrobat.zip	598 MB

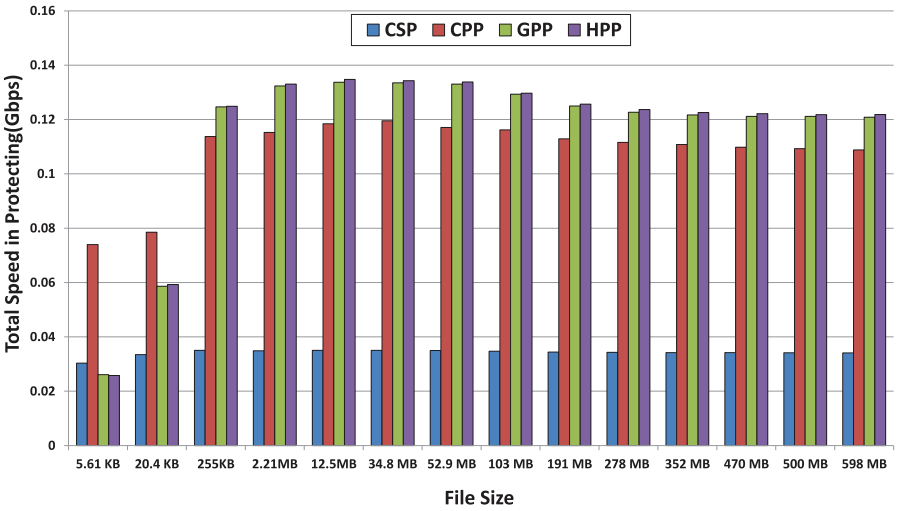

Fig. 3. Total protecting speed on Platform 1. This speed is calculated by *FileSize*/*TotalTime* in unit Gbps, where *FileSize* is the size of file to be protected and *TotalTime* represents the total time spent in protecting, including the time of reading file, writing file, encrypting file, and hashing file, etc.

Firstly, we choose 14 files for experiments, then perform protecting on them by CSP, CPP, GPP, and HPP, respectively, and record the time (including protecting total time, encrypting time, and hash time). For fully exhibiting the performance of SEFPS, we compute the protecting speed, encrypting speed, hash speed, etc.

Secondly, for those protected files produced in the previous step, we perform unprotecting on them by CSUP, CPUP, GPUP, and HPUP to recover and verify the original files, respectively, and record the time (including unprotecting time, decrypting time, and hash time). We also compute the unprotecting speed, decrypting speed, hash speed, etc. The experiments are executed 20 times for getting average results. The experimental results on Platforms 1 and 2 are given in Sections 6 and 7, respectively.

6. Experimental results and performance evaluation on Platform 1

The experimental results of executing SEFPS on Platform 1 will be described and discussed here. Firstly, we give and discuss the results of performing CSP, CPP, GPP, and HPP in Section 6.1, followed by CSUP, CPUP, GPUP, and HPUP in Section 6.2.

6.1. CSP, CPP, GPP, and HPP

For showing the performance of all CSP, CPP, GPP, and HPP clearly, we will give the results of protecting speed, encryption speed, hash speed, speedup, and parallel efficiency in Sections 6.1.1–6.1.4, respectively, and discuss these results.

6.1.1. Protecting speed

The protecting speed of CSP, CPP, GPP, and HPP can be seen in Fig. 3. The method of calculating protecting speed can be seen in the caption of Fig. 3.

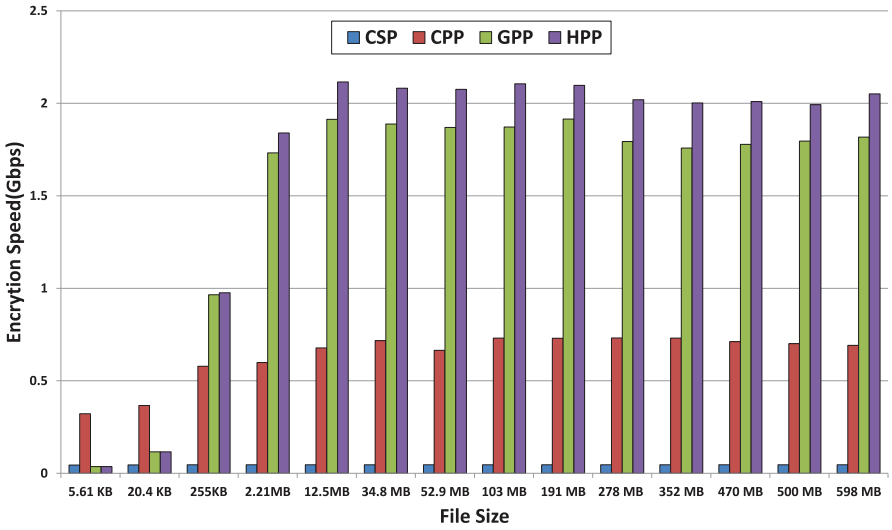

Fig. 4. Encryption speed on Platform 1. This speed is calculated by *FileSize*/*EncTime* in unit Gbps, where *FileSize* is the size of protected file and *EncTime* represents the time just spent in encrypting file. But for GPP, *EncTime* includes transferring data time and encrypting time on the GPU. And for HPP, *EncTime* includes all the time of threads on the CPUs finishing their work.

As shown in Fig. 3, HPP is superior to all GPP, CPP, and CSP, except the files with size of 5.61 and 20.4 KB. Specifically, for the file with size of 5.61 KB, the speeds of CSP, CPP, GPP, and HPP are 0.030343, 0.073959,0.026113, and 0.025791 Gbps, respectively. This means that CSP is faster than both GPP and HPP but slower than CPP for this file. This is because the relatively high extra overhead cost (such as transferring data, etc.) offsets the benefit from GPU parallelism for small files, but CPP has received the benefit from CPU parallelism because of less extra cost (such as deriving threads and scheduling them, etc.).

For the file with size of 20.4 KB, this situation is changed. The speeds of CSP, CPP, GPP, and HPP are 0.033454, 0.078497, 0.058616, and 0.059279 Gbps, respectively. Clearly, they all accelerate the speed of protecting. GPP and HPP surpass CSP, but are still slower than CPP. This is because this larger file can provide more data, and as a result, GPP and HPP get more benefit from GPU parallelism. In addition, HPP is slightly faster than GPP, because it has enough data to be split and executed simultaneously to GPUs and CPUs.

For the file with size of 255 KB, HPP outperforms all CSP, CPP, and GPP. Their speeds are 0.035017, 0.113735, 0.124685, and 0.124881 Gbps in the previous same order, respectively. This means that they accelerate their speeds, but GPP and HPP can get more benefit from GPU parallelism due to more data parallelism.

For the file with size of 2.21 MB, CSP, CPP, GPP, and HPP protect it at 0.034903, 0.115238, 0.132348, and 0.133041 Gbps, respectively, and slight increase in speed can be seen in protecting this file.

For our case files with size between 12.5 and 598 MB, the speeds of the four algorithms in protecting keep stable and the same trend, i.e., their speeds descend by HPP, GPP, CPP, and CSP. In details, the average speeds of HPP, GPP, CPP and CSP are 0.0345194, 0.1134414, 0.1262129, and 0.1270204 Gbps, respectively.

6.1.2. Encryption speed

The encryption speed of CSP, CPP, GPP, and HPP can be seen in Fig. 4.

As shown in Fig. 4, HPP is superior to CSP, CPP, and GPP, except the files with size of 5.61 and 20.4 KB. Specifically, for the file with size of 5.61 KB, the speeds of CSP, CPP, GPP, and HPP are 0.04437, 0.322115, 0.036276, and 0.035726 Gbps, respectively. This means that CSP is faster than both GPP and HPP but slower than CPP for this file. This is because the relatively high extra cost (such as transferring data, etc.) of GPP and HPP offsets the benefit from GPU parallelism for this small file, but CPP has received the benefit from CPU parallelism because of less extra cost (such as deriving threads and scheduling them, etc.).

For the file with size of 20.4 KB, the speeds of CSP, CPP, GPP, and HPP are 0.045587, 0.366286, 0.115808, and 0.116252 Gbps, respectively. GPP and HPP surpass CSP, but are still slower than CPP. This is because this larger file can provide more data, and as a result, GPP and HPP get more benefit from GPU parallelism. However, HPP is slightly faster than GPP because of its simultaneous execution on the GPU and the CPUs. Moreover, all CSP, CPP, GPP and HPP become faster in protecting this file than the previous file.

For the file with size of 255 KB, HPP outperforms all CSP, CPP, and GPP. Their speeds are 0.04629, 0.578567, 0.964797, and 0.975771 Gbps in the previous same order. This means that they all continuously accelerate, but HPP and GPP obtain more benefit from GPU parallelism or/and CPU parallelism due to more data parallelism.

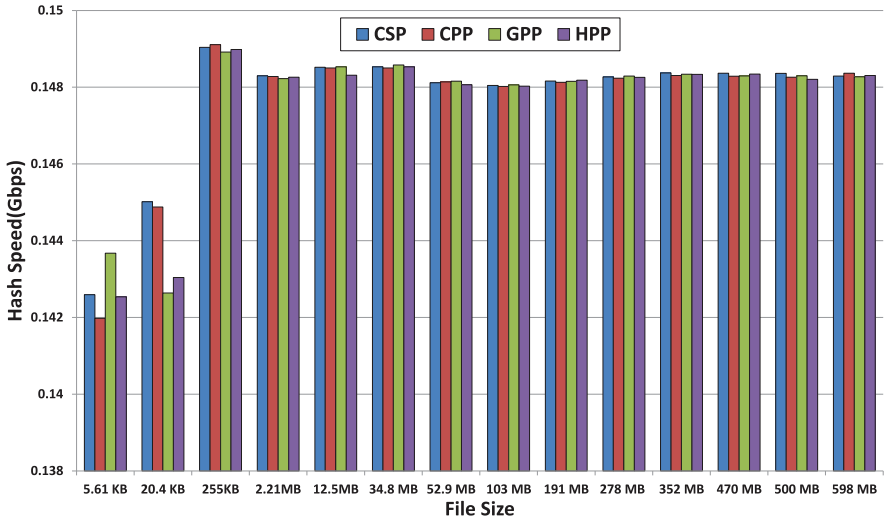

Fig. 5. Hash speed on Platform 1. This speed is calculated by *FileSize*/*HashTime* in unit Gbps, where *FileSize* is the size of protected file and *HashTime* represents the time just spent in hashing the file.

For the file with size of 2.21 MB, CSP, CPP, GPP, and HPP protect it at 0.046325, 0.598744, 1.73168, and 1.839178 Gbps, respectively. HPP impolitely enlarges its superiority than the others.

For our case files with size between 12.5 and 598 MB, the speeds of the four algorithms in encryption keep stable and have the same trend, i.e., that their speeds descend by HPP, GPP, CPP, and CSP. In details, the average speeds of HPP, GPP, CPP, and CSP are 0.046357, 0.708608, 1.840009, and 2.054892 Gbps, respectively.

All in all, the main difference of CSP, CPP, GPP, and HPP is encrypting data in protecting files. GPP encrypts the data by many threads of the GPU in parallel, and CPP by multiple threads of the CPUs in parallel, and HPP encrypts the data by the former two types of many threads simultaneously, and CSP encrypts the data by only one thread of the CPUs serially. Thus, if the data are enough, the performance of HPP will be superior to all CSP, CPP and GPP. On Platform 1, HPP is superior to all CSP, CPP, and GPP in protecting files which are equal to or larger than 255 KB.

6.1.3. Hash speed

The hash speeds of CSP, CPP, GPP, and HPP can be seen in Fig. 5.

As shown in Fig. 5, regardless of CSP, CPP, GPP or HPP, the hash speeds are almost same. For files with size of 5.61 and 12.5 KB, the hash speeds are relatively slower but are less than 4.2% relative difference to the others files. This is because all HPP, GPP, CPP and CSP use one thread of the CPUs in hashing files and the process of hashing has some constant overhead of function calls, such as initialization and finalization functions. And once the file has enough data, SHA3-256 produces steady performance. In addition, SHA3-256 has higher speed than the encrypting speed of CSP, their speeds are about 0.147621 and 0.046341 Gbps, respectively. This shows that SHA3-256 executed on Platform 1 is about 3.19 times as fast as AES-128 executed by one thread.

6.1.4. Protecting speedup

We will evaluate the performance of CSP, CPP, GPP, and HPP on Platform 1 from the other aspects, i.e., total protecting speedup and encrypting speedup. These two speedups are demonstrated in Figs. 6 and 7, respectively.

In Fig. 6, we can observe that HPP, GPP, and CPP have a speedup of about 3.66, 3.64, and 3.27 over CSP on total protecting, respectively. This demonstrates that HPP is superior to all GPP, CPP, and CSP in performance of protecting files, especially to CSP.

As shown in Fig. 7, HPP has a speedup of about 44.18 over CSP on encrypting, GPP has a speedup of about 39.52 over CSP, and CPP just has a speedup of about 15.35 over CSP. By Amdahl's law [42],

$$SP = \frac{1}{1 - r_p + \frac{r_p}{n}}, \tag{5}$$

where n represents the number of processors, SP represents the theoretical speedup of a program when it runs on the n processors, and r_p denotes the ratio of parallel portion in the program. That is to say, HPP has a larger speedup of encrypting than the speedup of protecting over CSP, because its serial part (including hashing, IO, and so on) offsets some part of the speedup. This conclusion is suitable for GPP and CPP too. Therefore, this shows that Amdahl's law works on our SEFPS. Moreover, this law enlightens us that improving the parallel portion of the file protecting system, such as by parallelizing hash, optimizing IO, and so on, can improve the speedup further.

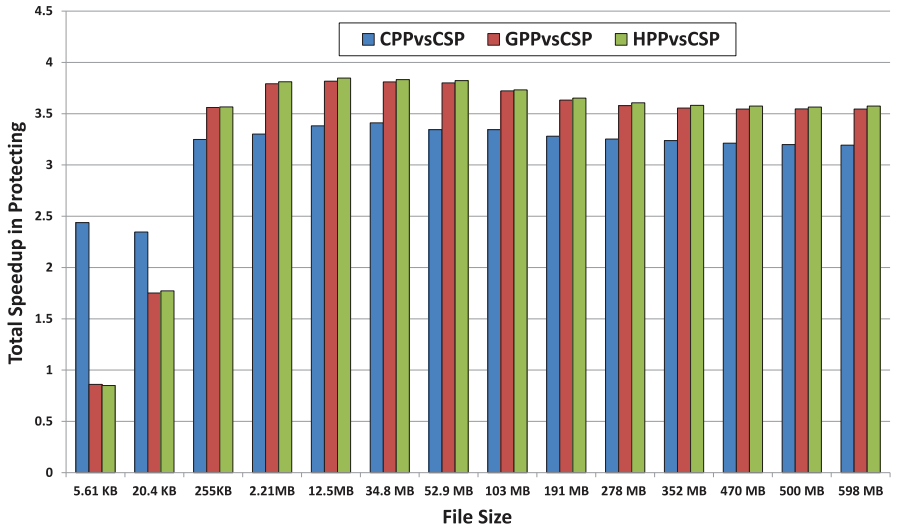

Fig. 6. Total protecting speedup on Platform 1. The speedup of HPP over CSP (HPPvsCSP) is calculated by *HPP_ProcSpeed*/*CSP_ProcSpeed*, where *HPP_ProcSpeed* and *CSP_ProcSpeed* represent the total protecting speed of HPP and CSP, respectively. Similarly, GPPvsCSP and CPPvsCSP are calculated by *GPP_ProcSpeed*/*CSP_ProcSpeed* and *CPP_ProcSpeed*/*CSP_ProcSpeed*, respectively.

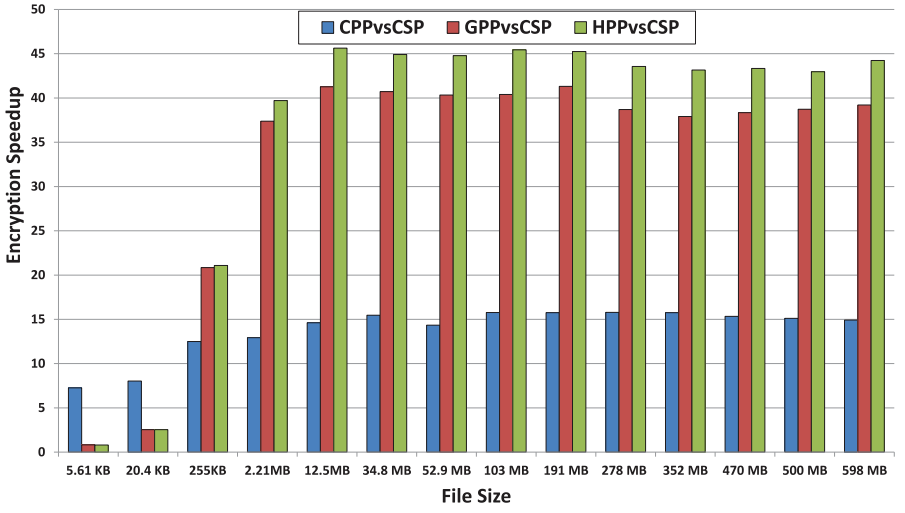

Fig. 7. Encryption speedup on Platform 1. The speedup of HPP over CSP (HPPvsCSP) is calculated by *HPP_EncSpeed*/*CSP_EncSpeed*, where *HPP_EncSpeed* and *CSP_EncSpeed* represent the encrypting speed of HPP and CSP, respectively. Similarly, GPPvsCSP and CPPvsCSP are calculated by *GPP_EncSpeed*/*CSP_EncSpeed* and *CPP_EncSpeed*/*CSP_EncSpeed*, respectively.

6.1.5. Parallel efficiency

Parallel efficiency is an important metric for evaluating how effectively a parallel program uses multiple processors. We will focus on the parallel part of SEFPS, i.e., the AES encryption/decryption, to evaluate the parallel efficiency.

For CPP, its parallel efficiency *EC* can be calculated as:

$$EC = \frac{SPC}{Cc} \times 100\%, \tag{6}$$

where *SPC* represents the speedup of CPP over CSP and *Cc* represents the number of cores of CPUs.

For fairness, we employ the relative computing capacity of GPUs to CPUs in calculating the parallel efficiency of GPP *EG*. In a given platform, the GPUs can be equivalent to CPUs with $ratio \times Cc$ cores, where *ratio* is defined in Eq. (1) and *Cc* represents the number of cores of the CPUs. Therefore, *EG* can be calculated as:

$$EG = \frac{SPG}{ratio \times Cc} \times 100\%, \tag{7}$$

where *SPG* represents the speedup of GPP over CSP.

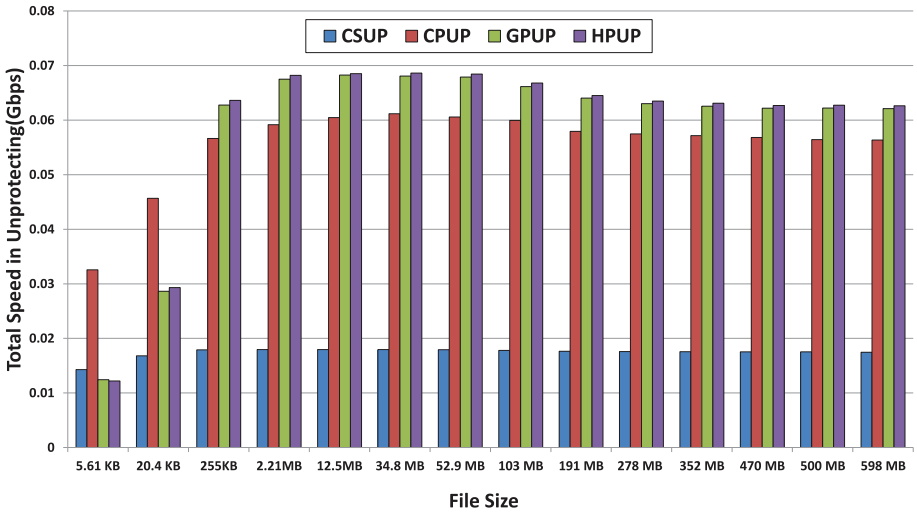

Fig. 8. Total unprotecting speed on Platform 1. This speed is calculated by *FileSize*/*TotalTime* in unit Gbps, where *FileSize* is the size of the file to be unprotected and *TotalTime* represents all the time spent in unprotecting, including the time of reading file, writing file, decrypting file, hashing file, checking hash value, etc.

Based on Eqs. (6) and (7), the parallel efficiency of HPP *EH* can be calculated as:

$$EH = \frac{SPH}{ratio \times Cc + Cc - 1} \times 100\%, \tag{8}$$

where *SPH* represents the speedup of HPP over CSP, $Cc - 1$ represents the number of working cores of CPUs except one service core. Similarly, the above methods are also adopted in calculating the parallel efficiencies of CPUP, GPUP, and HPUP.

After the calculations in our experiments, the average parallel efficiencies of CPP, GPP, and HPP are 92.0141%, 14.0002%, and 14.0756%. GPP and HPP on Platform 1 have lower parallel efficiencies than CPP, because one GPU core has relatively weak computing capacity than one CPU core, and GPP and HPP have the extra data transfer overhead. HPP has a little higher parallel efficiency than GPP, because HPP simultaneously uses the relatively higher parallel efficiency of the CPUs.

6.2. CSUP, CPUP, GPUP, and HPUP

For the legal users who possess the correct key, if they have enjoyed the services of protecting, they have the right to unprotect the protected files for using. This can be implemented by CPUP, GPUP or HPUP of SEFPS. For the sake of comparing, our experiments also include CSUP of SEFPS. Similarly, we will evaluate them on unprotecting speed in Section 6.2.1, on decryption speed in Section 6.2.2, on hash speed in Section 6.2.3, on speedup in Section 6.2.4, and on parallel efficiency in Section 6.2.5.

6.2.1. Unprotecting speed

The unprotecting speeds of CSUP, CPUP, GPUP, and HPUP can be seen in Fig. 8.

As shown in Fig. 8, HPUP is superior to all GPUP, CPUP, and CSUP, except the files with size of 5.61 and 20.4 KB. Concretely, for the file with size of 5.61 KB, the speeds of CSUP, CPUP, GPUP, and HPUP are 0.014267, 0.032573, 0.012423, and 0.012200 Gbps, respectively. This means that CSUP is faster than both GPUP and HPUP but slower than CPUP for this file. This is because the relatively high extra cost (such as transferring data, etc.) of GPUP and HPUP offsets the benefit from GPU parallelism for this small file, but CPUP has received the benefit from CPU parallelism because of less extra cost (such as deriving threads and scheduling them, etc.).

For the file with size of 20.4 KB, the speeds of CSUP, CPUP, GPUP, and HPUP are 0.016817, 0.045666, 0.028629, and 0.029316 Gbps, respectively. Both GPUP and HPUP have surpassed CSUP, but are still slower than CPUP. This is because a larger file can provide more data, and as a result, GPUP and HPUP get benefit from GPU parallelism but still have relatively higher cost than CPUP. Moreover, all HPUP, GPUP, CPUP, and CSUP become faster in unprotecting this file than the previous file because more data can improve the efficiency of IO transferring.

For the file with size of 255 KB, HPUP outperforms all the others. Their speeds are 0.017911, 0.056635, 0.062767, and 0.063626 Gbps ordered by CSUP, CPUP, GPUP, and HPUP. This means that they accelerate their speeds, but HPUP can get more benefit from GPU parallelism and CPU parallelism due to more cores.

For the file with size of 2.21 MB, CSUP, CPUP, GPUP, and HPUP unprotect it at 0.017943, 0.059170, 0.067514, and 0.068207 Gbps, respectively, and continue to increase their speeds. And HPUP greedily enlarges its superiority than all the others due to fully using computing resources.

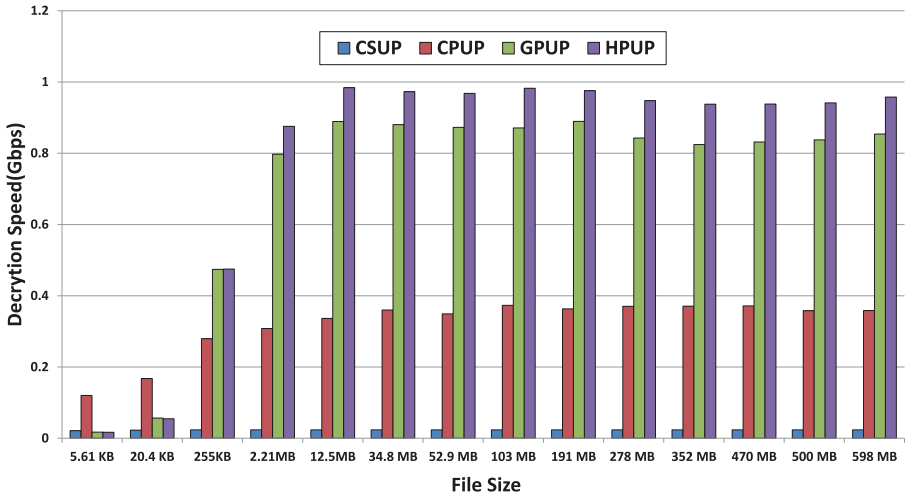

Fig. 9. Decryption speed on Platform 1. This speed is calculated by *FileSize*/*DecTime* in unit Gbps, where *FileSize* is the size of files to be unprotected and *DecTime* represents the time just spent in decrypting. But for GPUP, *DecTime* includes transferring data time and decrypting time on the GPU. And for HPUP, *DecTime* includes all the time spent in decrypting by the threads of both the CPUs and the GPU.

For those files with size between 12.5 and 598 MB, the speeds keep stable and have the same trend, i.e., that the speeds are ordered by HPUP, GPUP, CPUP, and CSUP descendingly. In details, their average speeds are 0.065152, 0.064648, 0.058435, and 0.017695 Gbps, respectively.

6.2.2. Decryption speed

Fig. 9 portrays the encryption speeds of CSUP, CPUP, GPUP, and HPUP.

As shown in Fig. 9, HPUP is superior to all CSUP, CPUP and GPUP, except the files with size of 5.61 and 20.4 KB. Specifically, for the file with size of 5.61 KB, the speeds of CSUP, CPUP, GPUP, and HPUP are 0.021158, 0.120299, 0.017518, and 0.016950 Gbps, respectively. This means that CSUP is faster than both GPUP and HPUP but slower than CPUP for this file. This is because the relatively high extra cost of GPUP and HPUP (such as transferring data, etc.) offsets the benefit from GPU parallelism for this small file, but CPUP has received the benefit from CPU parallelism because of less extra cost (such as deriving threads and scheduling them, etc.).

For the file with size of 20.4 KB, the speeds of CSUP, CPUP, GPUP, and HPUP are 0.022983, 0.167821, 0.054985, and 0.056727 Gbps, respectively. Both HPUP and GPUP surpass CSUP, but are still slower than CPUP. This is because a larger file can provide more data, and as a result, HPUP and GPUP get more benefit from GPU parallelism. Moreover, all CSUP, CPUP, GPUP, and HPUP become faster in unprotecting this file than the previous file.

For the file with size of 255 KB, HPUP outperforms all the others. Their speeds are 0.023667, 0.279617, 0.474197, and 0.475066 Gbps, respectively. This means that they all accelerate their speeds.

For the file with size of 2.21 MB, CSUP, CPUP, GPUP, and HPUP unprotect it at 0.023698, 0.308252, 0.797410, and 0.875649 Gbps, respectively, and continue to improve their performance. And HPUP quickly enlarges its superiority than all the others.

For our case files with size between 12.5 and 598 MB, the speeds of the four algorithms in decryption keep stable and have the same trend, i.e., that their speeds descend by HPUP, GPUP, CPUP, and CSUP. Specially, the average speeds of HPUP, GPUP, CPUP, and CSUP are 0.960619, 0.859300, 0.361211, and 0.023715 Gbps, respectively.

All in all, the main difference of CSUP, CPUP, GPUP, and HPUP is decrypting data in unprotecting files. HPUP decrypts the data using hybrid parallelism, CPUP using multiple threads of CPUs in parallel, GPUP using many threads of the GPU in parallel, and CSUP by only one thread of CPUs serially. Thus, if the data are enough, the performance of HPUP will be superior to all the others. On Platform 1, HPUP is superior to all the others in unprotecting files which are equal to or larger than 255 KB.

6.2.3. Hash speed

Fig. 10 exhibits the hash speeds of CSUP, CPUP, GPUP, and HPUP using the method described in the caption of Fig. 10.

As shown in Fig. 10, no matter CSUP, CPUP, GPUP or HPUP, the hash speed is almost same. For files with size of 5.61 and 12.5 KB, the hash speed are relatively slower but are less than 6.5% relative difference to the others files. This is because all HPUP, GPUP, CPUP, and CSUP use one thread of the CPUs in hashing files and the process of hashing has some constant overhead of function calls, such as initialization and finalization functions. And once the file has enough data, SHA3-256 produces its steady performance. In addition, SHA3-256 has higher speed than the decrypting speed of CSUP, their speeds are about 0.075415 and 0.023715 Gbps, respectively. This shows that SHA3-256 executed on Platform 1 is about 3.18 times as fast as AES-128 by one thread.

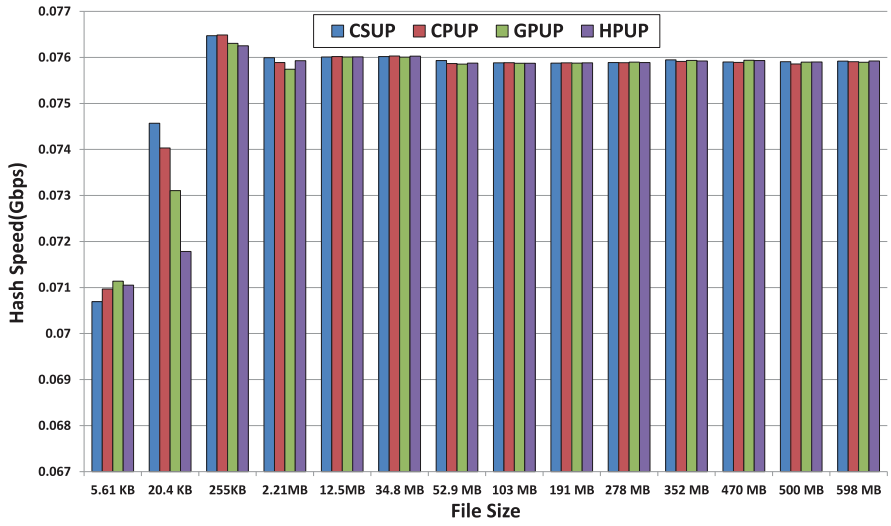

Fig. 10. Hash speed on Platform 1. This speed is calculated by *FileSize*/*HashTime* in unit Gbps, where *FileSize* is the size of the file to be hashed and *HashTime* represents the time just spent in hashing the file.

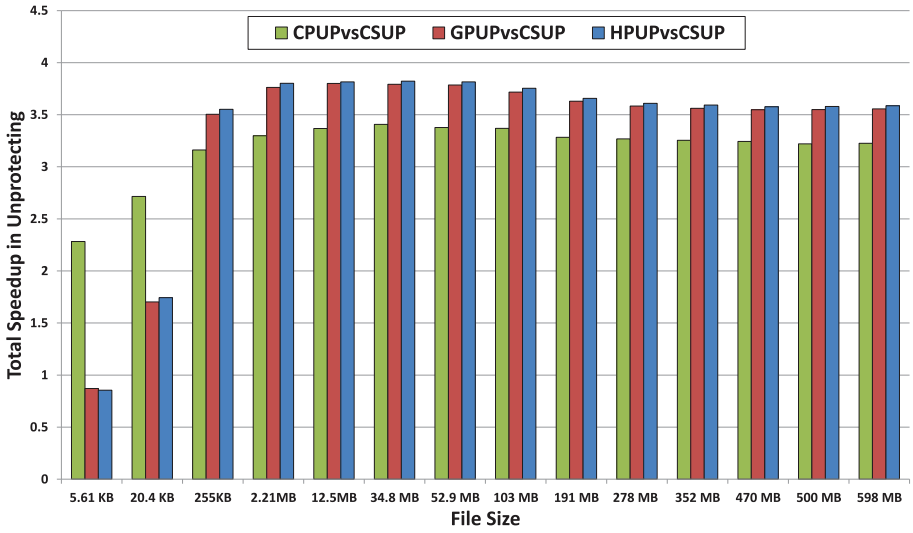

Fig. 11. Total unprotecting speedup on Platform 1. The speedup of HPUP over CSUP (HPUPvsCSUP) is calculated by *HPUP_UnprocSpeed*/*CSUP_UnprocSpeed*, where *HPUP_UnprocSpeed* and *CSUP_UnprocSpeed* represent the total unprotecting speed of HPUP and CSUP, respectively. Similarly, GPUPvsCSUP and CPUPvsCSUP are calculated by *GPUP_UnprocSpeed*/*CSUP_UnprocSpeed* and *CPUP_UnprocSpeed*/*CSUP_UnprocSpeed*, respectively, where *GPUP_UnprocSpeed* and *CPUP_UnprocSpeed* are the total unprotecting speed of GPUP and CPUP, respectively.

6.2.4. Unprotecting speedup

Now, we evaluate the performance of CSUP, CPUP, GPUP, and HPUP on Platform 1 from the other aspects, i.e., total unprotecting speedup and decrypting speedup. Figs. 11 and 12 demonstrate them, respectively.

In Fig. 11, HPUP has a speedup of about 3.67 over CSP on total unprotecting, GPUP has a speedup of about 3.65, and CPUP just has a speedup of about 3.30. This demonstrates that HPUP is superior to both GPUP and CPUP in unprotecting files.

As shown in Fig. 12, HPUP has a speedup of about 40.40 over CSUP on decrypting, GPUP has a speedup of about 36.10, and CPUP just has a speedup of about 15.37 over CSUP. That is to say, HPUP is superior to all CSUP, CPUP, and GPUP in decrypting files. Moreover, HPUP has a higher speedup of decrypting than its speedup of unprotecting, because its serial part (including hashing, IO, and so on) offsets some parts of speedup. This conclusion is suitable for both GPUP and CPUP too. Further, this demonstrates the effect of Amdahl's law on the SEFPS.

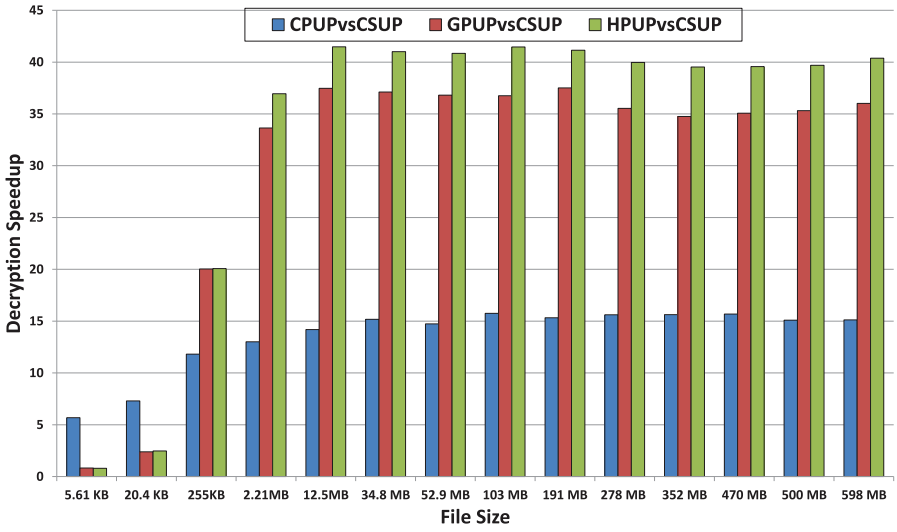

Fig. 12. Decryption speedup on Platform 1. The speedup of HPUP over CSUP (HPUPvsCSUP) is calculated by *HPUP_DecSpeed*/*CSUP_DecSpeed*, where *HPUP_DecSpeed* and *CSUP_DecSpeed* represents the decrypting speed of HPUP and CSUP, respectively. Similarly, GPUPvsCSUP and CPUPvsCSUP are calculated by *GPUP_DecSpeed*/*CSUP_DecSpeed* and *CPUP_DecSpeed*/*CSUP_DecSpeed*, respectively, where *GPUP_DecSpeed* and *CPUP_DecSpeed* are the decrypting speed of GPUP and CPUP, respectively.

6.2.5. Parallel efficiency

The average parallel efficiencies of CPUP, GPUP, and HPUP are calculated similarly by Eqs. (6), (7), and (8), respectively, and are 90.7761%, 13.2703%, and 13.3294%, respectively. These parallel efficiencies have similar relationship with those of CPP, GPP, and HPP. The reasons are similar too.

7. Experimental results and performance evaluation on Platform 2

This section describes and discusses the experimental results of executing SEFPS on Platform 2.

7.1. CSP, CPP, GPP, and HPP

In this subsection, we will give the results of protecting speed, encryption speed, hash speed, speedup, and parallel efficiency in Sections 7.1.1–7.1.5, respectively, and discuss these results.

7.1.1. Protecting speed

Fig. 13 demonstrates the protecting speeds of CSP, CPP, GPP, and HPP.

As shown in Fig. 13, HPP is superior to all the others when *FileSize* is equal to or larger than 2.21 MB. For the file with size of 5.61 KB, the speeds of CSP, CPP, GPP, and HPP are 0.006871, 0.007994, 0.006181, and 0.005889 Gbps, respectively. This means that CSP is faster than both GPP and HPP but slower than CPP for this file, because the relatively high extra cost (such as transferring data, etc.) of GPP and HPP offsets the benefit from GPU parallelism for this small file. But, CPP has received the benefit from CPU parallelism because of less extra cost (such as deriving threads and scheduling them, etc.).

For the file with size of 20.4 KB, the speeds of CSP, CPP, GPP, and HPP are 0.017030, 0.027959, 0.022369, and 0.021636 Gbps, respectively. Both GPP and HPP surpass CSP, but are still slower than CPP. This is because this larger file can provide more data, and as a result, both GPP and HPP get benefit from GPU parallelism. Moreover, all CSP, CPP, GPP, and HPP become faster in protecting this file than the previous file, because more data can improve the efficiency of IO transferring, etc.

For the file with size of 255 KB, GPP outperforms all the others. Their speeds are 0.028557, 0.079618, 0.093053, and 0.091778 Gbps at the previous same order. This means that they accelerate their speeds, but GPP can get more benefit for relatively lower overhead than HPP for this file.

For the file with size of 2.21 MB, CSP, CPP, GPP, and HPP protect it at 0.029761, 0.093186, 0.115534, and 0.119340, respectively, and continue to increase their speeds in protecting this file. HPP outperforms all the others in protecting this file for having enough data spit to and encrypted on CPUs and GPUs.

For our case files with size between 12.5 and 598 MB, the speeds of the four algorithms in protecting keep stable but have the same trend, i.e., that their speeds descend by HPP, GPP, CPP, and CSP. In details, the average speeds of HPP, GPP, CPP and CSP are 0.114912, 0.114199, 0.092277, and 0.029227 Gbps, respectively.

7.1.2. Encryption speed

The encryption speeds of CSP, CPP, GPP and HPP can be seen in Fig. 14.

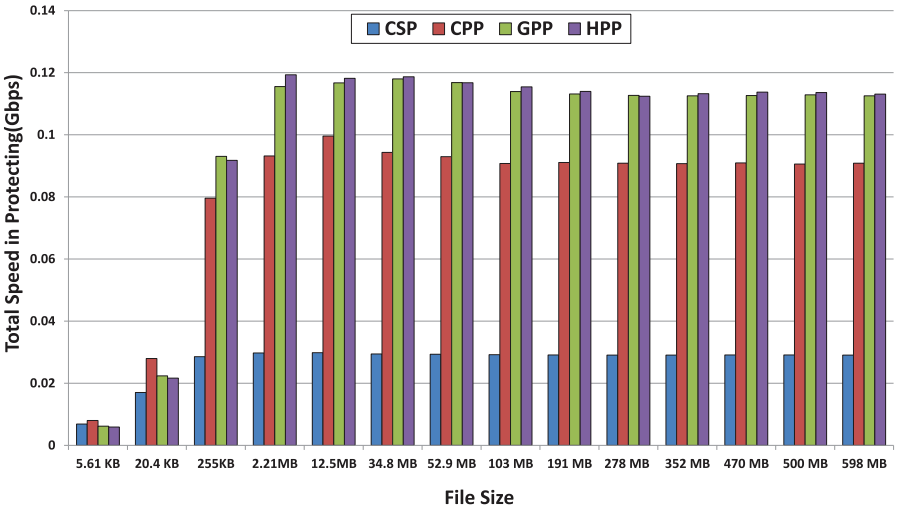

Fig. 13. Total protecting speed on Platform 2. This speed is calculated by *FileSize*/*TotalTime* in unit Gbps, where *FileSize* is the size of the file to be protected and *TotalTime* represents the total time spent in protecting, including the time of reading file, writing file, encrypting file, hashing file, etc.

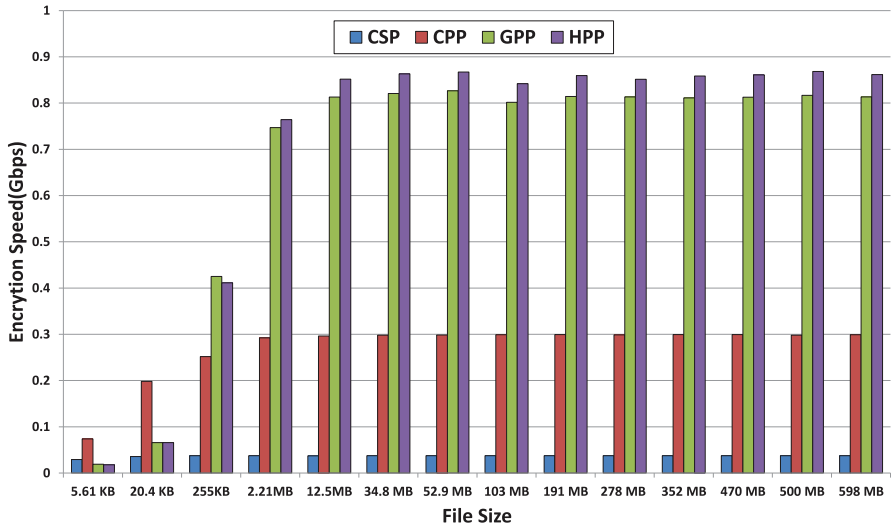

Fig. 14. Encryption speed on Platform 2. This speed is calculated by *FileSize*/*EncTime* in unit Gbps, where *FileSize* is the size of the protected file and *EncTime* represents the time just spent in encrypting the file. But for GPP and HPP, *EncTime* includes transferring data time and encrypting time.

As shown in Fig. 14, both GPP and HPP are superior to CPP and CSP, when the file is larger than or equal to 2.21 MB. This is because more GPU threads accelerate the encryption due to enough workload. CPP is uncertainly faster than CSP because of more CPU threads.

For the file with size of 2.21 MB, CSP, CPP, GPP, and HPP encrypt it at 0.037535, 0.292387, 0.746905, and 0.763977 Gbps, respectively. HPP surpasses all the others in encrypting the file with size of 2.21 MB.

For our case files with size between 12.5 and 598 MB, the speeds of the four algorithms in encryption keep stable and have the same trend, i.e., that their speeds descend by HPP, GPP, CPP, and CSP. In other words, the average speeds of HPP, GPP, CPP, and CSP are 0.858311, 0.814322, 0.298615, and 0.037397 Gbps, respectively.

All in all, the main difference of CSP, CPP, GPP, and HPP is encrypting data in protecting files. HPP encrypts the data by GPU parallelism and CPU parallelism, GPP encrypts the data by many threads of GPUs in parallel, CPP by multiple threads of CPUs in parallel, and CSP by only one thread of CPUs serially. Thus, if the data are enough, the performance of HPP will be superior to all the others. In Platform 2, HPP is superior to all the others when files to be encrypted are equal to or larger than 2.21 MB.

7.1.3. Hash speed

The hash speeds of CSP, CPP, GPP, and HPP are portrayed in Fig. 15.

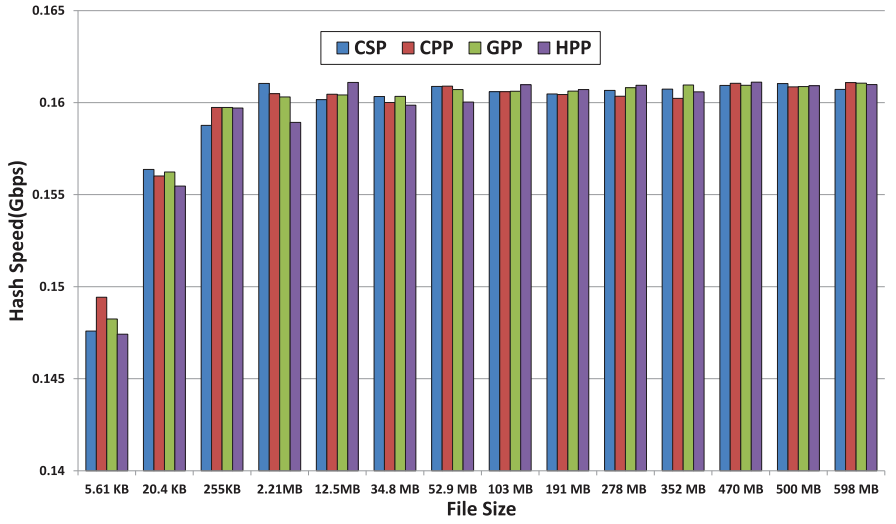

Fig. 15. Hash speed on Platform 2. This speed is calculated by *FileSize*/*HashTime* in unit Gbps, where *FileSize* is the size of file to be protected and *HashTime* represents the time just spent in hashing the file.

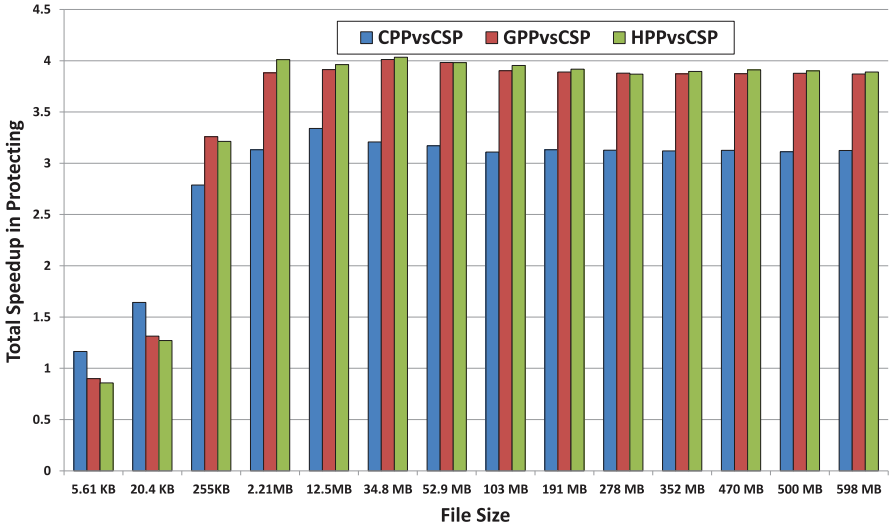

Fig. 16. Total protecting speedup on Platform 2. This speedup of HPP over CSP (HPPvsCSP) is calculated by *HPP_ProcSpeed*/*CSP_ProcSpeed*, where *HPP_ProcSpeed* and *CSP_ProcSpeed* represent the total protecting speed of HPP and CSP, respectively. Similarly, GPPvsCSP and CPPvsCSP are calculated by *GPP_ProcSpeed*/*CSP_ProcSpeed* and *CPP_ProcSpeed*/*CSP_ProcSpeed*, respectively, where *GPP_ProcSpeed* and *CPP_ProcSpeed* are the total protecting speed of GPP and CPP, respectively.

As shown in Fig. 15, no matter CSP, CPP, GPP or HPP, the hash speed is almost same. The speed is about 0.159331 Gbps. This is because all the algorithms use one CPU thread in hashing files.

7.1.4. Speedup

Now, we evaluate the performance of CSP, CPP, GPP, and HPP on Platform 2 from other aspects, i.e., total protecting speedup and encrypting speedup. These two speedups are demonstrated in Figs. 16 and 17, respectively.

In Fig. 16, we can observe that HPP has a speedup of about 3.93 over CSP on total protecting, GPP has a speedup of about 3.91 over CSP on total protecting, and CPP has a speedup of about 3.16 over CSP on total protecting. This demonstrates that HPP is superior to CPP in protecting files.

As shown in Fig. 17, HPP has a speedup of about 22.96 over CSP on encrypting, GPP has a speedup of about 21.77 over CSP on encrypting, and CPP just has a speedup of about 7.99 over CSP on encrypting. That is to say, HPP has a larger speedup of encrypting than the speedup of protecting over CSP, because its serial part (including hashing, IO, etc.) offsets some part of speedup. This conclusion is suitable for both GPP and CPP too. This shows that Amdahl's law works on our SEFPS.

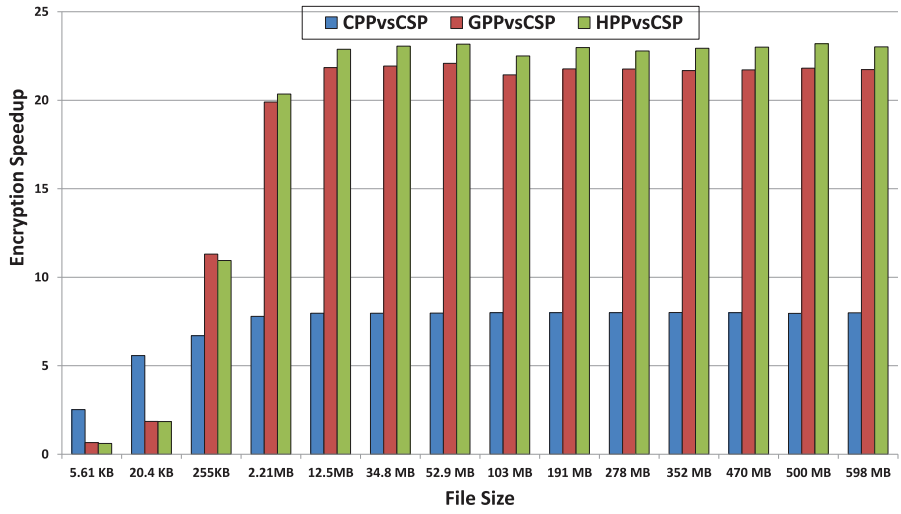

Fig. 17. Encryption speedup on Platform 2. This speedup of HPP over CSP (HPPvsCSP) is calculated by *HPP_EncSpeed*/*CSP_EncSpeed*, where *HPP_EncSpeed* and *CSP_EncSpeed* represent the encrypting speed of HPP and CSP, respectively. Similarly, GPPvsCSP and CPPvsCSP are calculated by *GPP_EncSpeed*/*CSP_EncSpeed* and *CPP_EncSpeed*/*CSP_EncSpeed*, respectively.

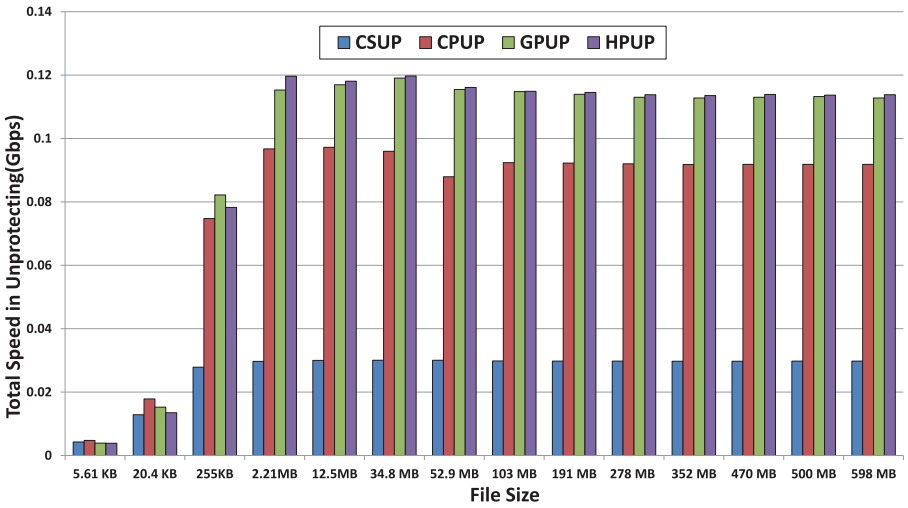

Fig. 18. Total unprotecting speed on Platform 2. This speed is calculated by *FileSize*/*TotalTime* in unit Gbps, where *FileSize* is the size of the file to be unprotected and *TotalTime* represents the total time spent in unprotecting, including the time of reading file, writing file, decrypting file, hashing file, checking hash value, etc.

7.1.5. Parallel efficiency

The average parallel efficiencies of CPP, GPP, and HPP on Platform 2 are calculated by Eqs. (6), (7), and (8), respectively, and are 87.0522%, 28.1362%, and 28.1938%, respectively. These parallel efficiencies have similar relationship with those on Platform 1. But GPP and HPP on Platform 2 have higher parallel efficiencies than on Platform 1. This is because the GPU on Platform 2 transfers data faster. The data transfer bandwidth of the GPU on Platform 1 is about 4.711 Gbps while on Platform 2 is about 6.283 Gbps.

7.2. CSUP, CPUP, GPUP, and HPUP

In this subsection, we will give the results of CSUP, CPUP, GPUP, and HPUP of SEFPS, and then discuss them. Similarly, we will evaluate them on unprotecting speed in Section 7.2.1, on decryption speed in Section 7.2.2, on hash speed in Section 7.2.3, on speedup in Section 7.2.4, and on parallel efficiency in Section 7.2.5.

7.2.1. Unprotecting speed

The unprotecting speeds of CSUP, CPUP, GPUP, and HPUP can be seen in Fig. 18.

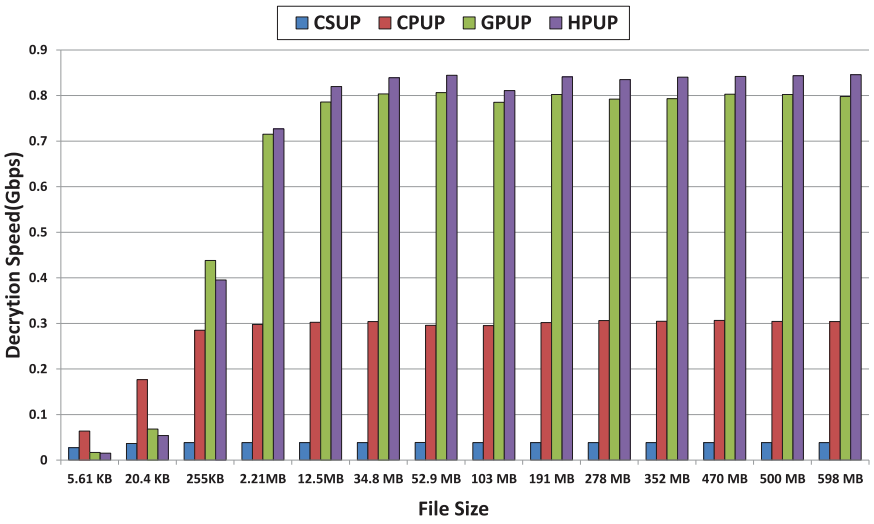

Fig. 19. Decryption speed on Platform 2. This speed is calculated by *FileSize*/*DecTime* in unit Gbps, where *FileSize* is the size of files to be unprotected and *DecTime* represents the time just spent in decrypting. But for GPUP and HPUP, *DecTime* includes data transferring time and decrypting time.

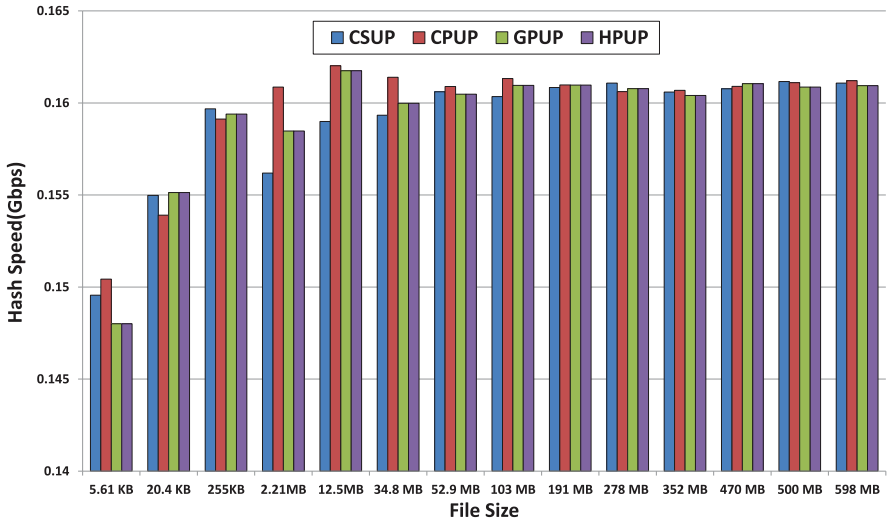

Fig. 20. Hash speed on Platform 2. This speed is calculated by *FileSize*/*HashTime* in unit Gbps, where *FileSize* is the size of files to be hashed and *HashTime* represents the time just spent in hashing the file.

As shown in Fig. 18, the basic trend of unprotecting speed is the same as protecting speed. For conciseness, we just discuss the average speeds of unprotecting the case files with size between 12.5 and 598 MB, because their speeds keep stable. The average speeds of HPUP, GPUP, CPUP, and CSUP are 0.115228, 0.114520, 0.092527, and 0.029857 Gbps, respectively.

7.2.2. Decryption speed

Fig. 19 portrays the encryption speeds of CSUP, CPUP, GPUP, and HPUP on Platform 2, and exhibits the same trend as decryption speeds. Thus, we briefly discuss the average encryption speeds.

On Platform 2, HPUP is superior to all the others in unprotecting files which are equal to or larger than 2.21 MB. In our case files with size between 12.5 and 598 MB, the encryption average speeds of CSUP, CPUP, GPUP, and HPUP are 0.038372, 0.295220, 0.785203, and 0.810829 Gbps, respectively. In a word, the main difference of CSUP, CPUP, GPUP, and GPUP is decrypting data in unprotecting files. HPUP decrypts the data by GPU parallelism and CPU parallelism simultaneously, GPUP decrypts the data by many threads of GPUs in parallel, CPUP by multiple threads of CPUs in parallel, and CSUP by only one thread of CPUs serially. Thus, if the data are enough, the performance of HPUP will be superior to all the others.

7.2.3. Hash speed

Fig. 20 exhibits the hash speeds of HPUP, GPUP, CPUP, and CSUP. These speeds have almost the same value about 0.160771 Gbps. This is because all the algorithms use one CPU thread in hashing files.

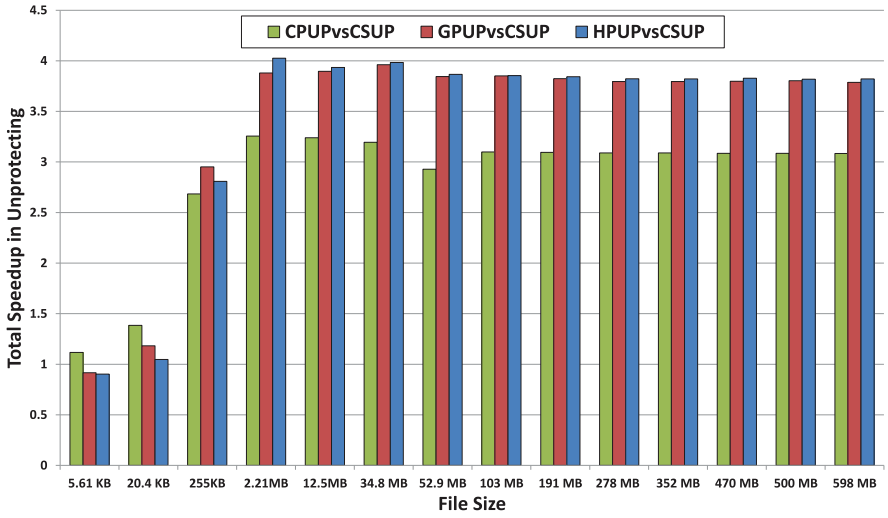

Fig. 21. Total unprotecting speedup on Platform 2. This speedup of HPUP over CSUP (HPUPvsCSUP) is calculated by *HPUP_UnprocSpeed*/*CSUP_UnprocSpeed*, where *HPUP_UnprocSpeed* and *CSUP_UnprocSpeed* represent the total unprotecting speed of HPUP and CSUP, respectively. Similarly, GPUPvsCSUP and CPUPvsCSUP are calculated by *GPUP_UnprocSpeed*/*CSUP_UnprocSpeed* and *CPUP_UnprocSpeed*/*CSUP_UnprocSpeed*, respectively, where *GPUP_UnprocSpeed* and *CPUP_UnprocSpeed* are the total unprotecting speeds of GPUP and CPUP, respectively.

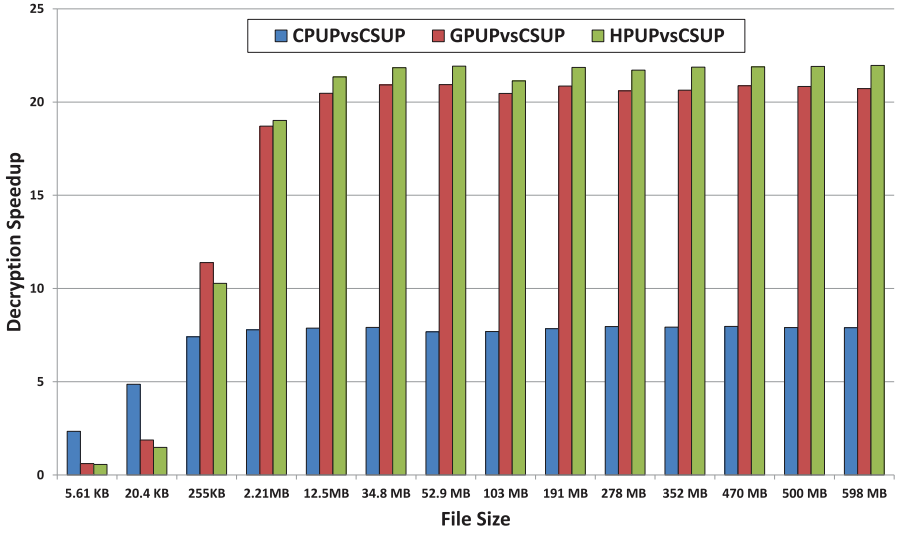

Fig. 22. Decryption speedup on Platform 2. This speedup of HPUP over CSUP (HPUPvsCSUP) is calculated by *HPUP_DecSpeed*/*CSUP_DecSpeed*, where *HPUP_DecSpeed* and *CSUP_DecSpeed* represent the decrypting speed of HPUP and CSUP, respectively. Similarly, GPUPvsCSUP and CPUPvsCSUP are calculated by *GPUP_DecSpeed*/*CSUP_DecSpeed* and *CPUP_DecSpeed*/*CSUP_DecSpeed*, respectively, where *GPUP_DecSpeed* and *CPUP_DecSpeed* are the decrypting speeds of GPUP and CPUP, respectively.

7.2.4. Speedup

Now, we evaluate the performance of CSUP, CPUP, GPUP, and HPUP on Platform 2 from the other aspects, i.e., total unprotecting speedup and decrypting speedup. Figs. 21 and 22 demonstrate these two speedups, respectively.

In Fig. 21, HPUP has a speedup of about 3.86 over CSUP on total unprotecting, GPUP has a speedup of about 3.84 over CSUP, and CPUP just has a speedup of about 3.10 over CSUP. This demonstrates that HPUP is superior to all the others in performance of unprotecting files, especially to CSUP.

As shown in Fig. 22, HPUP has a speedup of about 21.74 over CSUP on decrypting, GPUP has a speedup of about 20.73 over CSUP, and CPUP has a speedup of about 7.87 over CSUP. That is to say, HPUP has a larger speedup of decrypting than the speedup of unprotecting over CSUP, but its serial part (including hashing, IO, and so on) offsets some parts of speedup. This conclusion is suitable for all the others too. This demonstrates that Amdahl's law works on the SEFPS.

7.2.5. Parallel efficiency

The average parallel efficiencies of CPUP, GPUP, and HPUP on Platform 2 are calculated similarly, and are 85.6512%, 25.5484%, and 25.6246%, respectively. These parallel efficiencies have similar relationship with CPP, GPP, and HPP on Platform 2. The reasons are similar too.

8. Conclusions

We present a secure and high efficient file protecting system (SEFPS) based on SHA3-256 and parallel AES. It can be used in protecting users' files for transferring or storing safely.

On one side, SEFPS can achieve the security of confidentiality and integrity. That is to say, it can effectively protect files against the attacks of forging or breaking. On the other side, SEFPS have higher performance than its CPU serial implementations (i.e., including CSP and CSUP). SEFPS' six algorithms (CPP, CPUP, GPP, GPUP, HPP, HPUP) are designed and implemented on two typical platforms. These two platforms represent low-end and high-end computers, respectively. We evaluate their encryption/decryption speed, hashing speed, total protecting/unprotecting speed, etc., through the experiments of protecting or unprotecting 14 files.

In summary, the speedups on Platform 1 are followings. For speedup of total protecting, HPP is about 3.66 times over CSP, GPP is about 3.64 times over CSP, and CPP is about 3.27 times over CSP. For speedup of encrypting, HPP is about 44.18 times over CSP, GPP is about 39.52 times over CSP, and CPP is about 15.35 times over CSP. For speedup of total unprotecting, HPUP is about 3.67 times over CSUP, GPUP is about 3.65 times over CSUP, and CPUP is about 3.30 times over CSUP. For speedup of decrypting, HPUP is about 40.40 times over CSUP, GPUP is about 36.10 times over CSUP, and CPUP is about 15.37 times over CSUP.

In summary, the speedups on Platform 2 are described below. For speedup of total protecting, HPP is about 3.93 times over CSP, GPP is about 3.91 times over CSP, and CPP is about 3.16 times over CSP. For speedup of encrypting, HPP is about 22.96 times over CSP, GPP is about 21.77 times over CSP, and CPP is about 7.99 times over CSP. For speedup of total unprotecting, HPUP is about 3.86 times over CSUP, GPUP is about 3.84 times over CSUP, and CPUP is about 3.10 times over CSUP. For speedup of decrypting, HPUP is about 21.74 times over CSUP, GPUP is about 20.73 times over CSUP, and CPUP is about 7.87 times over CSUP.

Naturally, HPP and HPUP outperform GPP and GPUP, respectively, and outperform CPP and CPUP more. Of course, GPP and GPUP outperform CSP and CSUP, respectively. For those computers not equipped with Nvidia GPUs, CPP and CPUP can be employed, which still outperform CSP and CSUP, respectively. Moreover, because SEFPS can run on CPUs or/and GPUs by choosing CPP, CPUP, GPP, GPUP, HPP, and HPUP, it is also universal for protecting/unprotecting files.

This work represents our initial work to effectively protect files based on SHA3-256 and parallel AES, and we correspondingly design, implement, and evaluate six algorithms which are suitable for GPUs or CPUs. In the future, we plan to further perfect the file protecting system, such as improving IO, optimizing SHA3-256, and so on.

Acknowledgments

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 and Technology Cooperation Program of China(2015DFA11240), the National High Technology Research and Development Program (grant no. 2014AA01A302), and the Scientific Research Project of Hunan Provincial Education Department (grant no. 15C0254).

References

[1] K. Li, W. Yang, K. Li, Performance analysis and optimization for SPMV on GPU using probabilistic modeling, IEEE Trans. Parallel Distrib. Syst. 26 (1) (2014) 196–205.
[2] W. Yang, K. Li, Z. Mo, Performance optimization using partitioned SPMV on GPUs and multicore CPUs, IEEE Trans. Comput. 64 (9) (2015) 2623–2636.
[3] H.-V. Dang, B. Schmidt, Cuda-enabled sparse matrix–vector multiplication on GPUs using atomic operations, Parallel Comput. 39 (11) (2013) 737–750.
[4] M. Krotkiewski, M. Dabrowski, Efficient 3d stencil computations using cuda, Parallel Comput. 39 (10) (2013) 533–548.
[5] J. Daemen, V. Rijmen, AES Proposal, Rijndael, 1999.
[6] P. FIPS, 197: Specification for the Advanced Encryption Standard, 2001, vol. 4 (2009) 17–18. http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf
[7] R.L. Rivest, Rfc 1321: The md5 message-digest algorithm, Internet Activities Board, vol. 143 (1992).
[8] S.H. Standard, The cryptographic hash algorithm family: Revision of the secure hash standard and ongoing competition for new hash algorithms, 2009. Retrieved Jan 26, 2015, from http://csrc.nist.gov/publications/nistbul/March2009_cryptographic-hash-algorithm-family.pdf.
[9] X. Wang, D. Feng, X. Lai, H. Yu, Collisions for hash functions md4, md5, haval-128 and ripemd, in: Rump session of Crypto 2004, E-print (2004). Retrieved Feb 3, 2015, from http://eprint.iacr.org/2004/199.pdf.
[10] X. Wang, Y.L. Yin, H. Yu, Finding collisions in the full sha-1, in: Advances in Cryptology-Crypto 05, Springer, Berlin Heidelberg, 2005, pp. 17–36.
[11] D. Khovratovich, C. Rechberger, A. Savelieva, Bicliques for preimages: Attacks on skein-512 and the sha-2 family, in: Fast Software Encryption, Springer, 2012, pp. 244–263.
[12] C. Boutin, Nist selects winner of secure hash algorithm(sha-3) competition, Retrieved Jan 18, 2015, from http://www.nist.gov/itl/csd/sha-100212.cfm.
[13] S.A. Manavski, CUDA compatible GPU as an efficient hardware accelerator for AES cryptography, in: IEEE International Conference on Signal Processing and Communications, 2007 (ICSPC'07), IEEE, 2007, pp. 65–68.

[14] K. Iwai, T. Kurokawa, N. Nisikawa, AES encryption implementation on CUDA GPU and its analysis, in: 2010 First International Conference on Networking and Computing (ICNC), IEEE, 2010, pp. 209–214.
[15] P. Maistri, F. Masson, R. Leveugle, Implementation of the advanced encryption standard on gpus with the nvidia cuda framework, in: 2011 IEEE Symposium on Industrial Electronics and Applications (ISIEA), IEEE, 2011, pp. 213–217.
[16] C.-L. Duta, G. Michiu, S. Stoica, L. Gheorghe, Accelerating encryption algorithms using parallelism, in: 2013 19th International Conference on Control Systems and Computer Science (CSCS), IEEE, 2013, pp. 549–554.
[17] A. Pousa, V. Sanz, A. de Giusti, Performance analysis of a symmetric cryptographic algorithm on multicore architectures , in: Computer Science & Technology Series-XVII Argentine Congress of Computer Science-Selected Papers, Edulp, 2012, pp. 57–66.
[18] H. Chen, P. Lee, Enabling data integrity protection in regenerating-coding-based cloud storage: Theory and implementation, IEEE Trans. Parallel Distrib. Syst. 25 (2) (2014) 407–416.
[19] G.-A. Yandji, L.L. Hao, A.-E. Youssouf, J. Ehoussou, Research on a normal file encryption and decryption, in: 2011 International Conference on Computer and Management (CAMAN), IEEE, 2011, pp. 1–4.
[20] C. JunLi, Q. Dinghu, Y. Haifeng, Z. Hao, M. Nie, Email encryption system based on hybrid aes and ecc, in: IET International Communication Conference on Wireless Mobile and Computing (CCWMC'11), IET, 2011, pp. 347–350.
[21] J. Zhang, X. Jin, Encryption system design based on des and sha-1, in: 2012 11th International Symposium on Distributed Computing and Applications to Business, Engineering Science (DCABES), IEEE, 2012, pp. 317–320.
[22] L. Tan, S.L. Song, P. Wu, Z. Chen, R. Ge, D.J. Kerbyson, Investigating the interplay between energy efficiency and resilience in high performance computing, in: Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium, IEEE, 2015, pp. 786–796.
[23] K. Li, J. Liu, L. Wan, S. Yin, K. Li, A cost-optimal parallel algorithm for the 0–1 knapsack problem and its performance on multicore cpu and gpu implementations, Parallel Comput. 43 (2015) 27–42.
[24] X. Shi, F. Park, L. Wang, J. Xin, Y. Qi, Parallelization of a color-entropy preprocessed chan-vese model for face contour detection on multi-core cpu and gpu, Parallel Comput. 49 (2015) 28–49.
[25] L. Tan, Z. Chen, S.L. Song, Scalable energy efficiency with resilience for high performance computing systems: A quantitative methodology, ACM Trans. Archit. Code Optim. 12 (4) (2015) Article35.
[26] J.W. Bos, D.A. Osvik, D. Stefan, Fast implementations of AES on various platforms, IACR Cryptol. 2009 (2009) Retrieved Jan 19, 2015, from http://eprint.iacr.org/2009/501.pdf.
[27] C. Mei, H. Jiang, J. Jenness, 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, 2010, pp. 1–7.
[28] K. Iwai, N. Nishikawa, T. Kurokawa, Acceleration of AES encryption on CUDA GPU, Int. J. Netw. Comput. 2 (1) (2012) 131–145.
[29] T. Nhat-Phuong, L. Myungho, H. Sugwon, L. Seung-Jae, High throughput parallelization of AES-CTR algorithm, IEICE Trans. Inform. Syst. 96 (8) (2013) 1685–1695.
[30] S. Navalgund, A. Desai, K. Ankalgi, H. Yamanur, Parallelization of AES algorithm using OpenMP, Lect. Notes Inform. Theor. 1 (4) (2013) 144–147.
[31] M. Nagendra, M.C. Sekhar, Performance improvement of Advanced Encryption Algorithm using parallel computation, Int. J. Softw. Eng. Appl. 8 (2) (2014) 287–296.
[32] F. Hossain, M. Ali, M. Al Abedin Syed, A very low power and high throughput aes processor, in: 2011 14th International Conference on Computer and Information Technology (ICCIT), IEEE, 2011, pp. 339–343.
[33] A. Moh'd, Y. Jararweh, L.A. Tawalbeh, Aes-512: 512-bit advanced encryption standard algorithm design and evaluation, in: IAS, IEEE, 2011, pp. 292–297.
[34] J.S. Banu, M. Vanitha, J. Vaideeswaran, S. Subha, 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, 2013, pp. 481–485.
[35] B. Liu, B.M. Baas, Parallel aes encryption engines for many-core processor arrays, IEEE Trans. Comput. 62 (3) (2013) 536–547.
[36] A. Malik, A. Aziz, D.-e.-S. Kundi, M. Akhter, Software implementation of standard hash algorithm (sha-3) keccak on intel core-i5 and cavium networks octeon plus embedded platform, in: 2013 2nd Mediterranean Conference on Embedded Computing (MECO), IEEE, 2013, pp. 79–83.
[37] S. Bayat-Sarmadi, M. Mozaffari-Kermani, A. Reyhani-Masoleh, Efficient and concurrent reliable realization of the secure cryptographic sha-3 algorithm, IEEE Trans. Comput.-Aid. Design Integ. Circuits Syst. 33 (7) (2014) 1105–1109.
[38] N. Moreira, A. Astarloa, U. Kretzschmar, Sha-3 based message authentication codes to secure IEEE 1588 synchronization systems, in: 39th Annual Conference of the IEEE on Industrial Electronics Society (IECON'13), IEEE, 2013, pp. 2323–2328.
[39] Q. Dong, J. Zhang, L. Wei, A sha-3 based rfid mutual authentication protocol and its implementation, in: 2013 IEEE International Conference on Signal Processing, Communication and Computing (ICSPCC), IEEE, 2013, pp. 1–5.
[40] M. Biglari, E. Qasemi, B. Pourmohseni, Maestro: A high performance AES encryption/decryption system, in: 2013 17th CSI International Symposium on Computer Architecture and Digital Systems (CADS), IEEE, 2013, pp. 145–148.
[41] J. Diaz, C. Munoz-Caro, A. Nino, A survey of parallel programming models and tools in the multi and many-core era, IEEE Trans. Parallel Distrib. Syst. 23 (8) (2012) 1369–1386.
[42] G.M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, in: Proceedings of the Spring Joint Computer Conference, April 18–20, 1967, ACM, 1967, pp. 483–485.