Parallel Implementations of Compression and Cryptographic Algorithms

Yuqing Guan, Jihan Li
{yg2392, jl4346}@columbia.edu

I. Introduction

Compression and cryptographic algorithms are widely used in every area of computer science. Since all of these algorithms are aiming to transform data to some other format and then convert it back losslessly, which requires a great amount of computation, we can employ GPU devices' rich computing resources, and parallelize these algorithm to improve the efficiency of compression and cryptography.

In our project, we study on several cryptographic algorithms, like DES, AES and RSA, and compression algorithms like RLE, Huffman, LZ77 and LZW, and try to port them to the CUDA platform. We find that cryptographic algorithms are better fits for parallel execution on GPU devices, and succeed in increasing their speed by about 10 ~ 50 times, while compression algorithms are relatively harder to be accelerated by CUDA. We also create a command-line and GUI application for the cryptographic algorithms.

II. Original Algorithms

There are many mature CPU implementations of these aforementioned algorithms. We read some of them to understand these algorithms and verify the correctness of our own programs. Then we write our own implementations for both CPU and GPU platforms.

1. Cryptographic algorithms

a) DES

DES (Data Encryption Standard) is a symmetric-key algorithm. It is a block cipher which divides plaintext to 64-bit blocks (we will call them 'chunks' in the latter part of this report, to avoid confusion with the 'blocks' in CUDA).

i. Set key

DES uses a 64-bit key to encrypt plaintext and decrypt ciphertext. When setting a new key, DES firstly selects 56 bits from it by Permuted Choice 1 (PC1), then divides these bits to two halves.

After selecting the 56 bits, DES performs 16 rounds of sub key generations. When creating one sub key, DES rotates both halves of the 56-bit key, and selects 48 bits as a sub key by Permuted Choice 2 (PC2). [1][2]

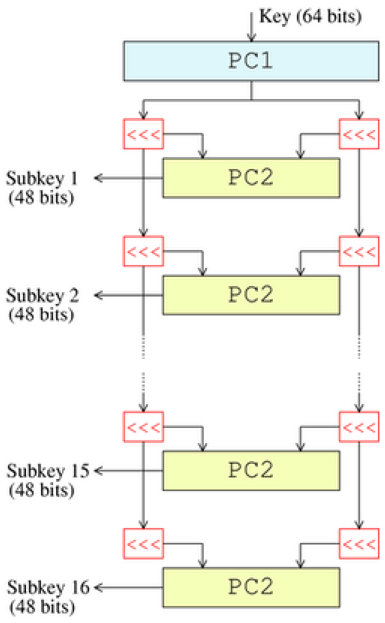


Figure 1. The key-schedule of DES [1]

ii. Encryption

When encrypting plaintext, DES firstly permutes it by an IP lookup table, then the text will be processed in 16 rounds of operations: Firstly, the 64-bit chunk is divided to 2 halves. The right half will be copied to the left half in the next round, while it will also be passed to a Feistel function including the following four manipulations:

1. Key expansion: expand the 32-bit half chunk to 48 bits
2. Key mixing: combine the expanded half chunk with the sub key of the current round
3. Substitution: apply an S-box transformation
4. Permutation: rearrange the result of the S-box transformation by a fixed permutation

The Feistel result will be combined with the left half in this round by XOR, and the result will be passed to the right half in the next round. After all 16 rounds and an extra FP permutation, the plaintext can be transformed into ciphertext. [1][2]

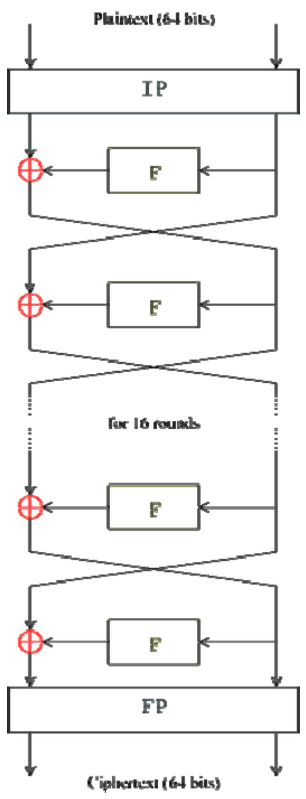


Figure 2. The overall Feistel structure of DES [1]

iii. Decryption

DES decryption is very similar to the encryption. The key difference between them is that DES decryption will apply the Feistel function on left half chunks. The IP and FP permutations are exactly the same. [1][2]

b) AES

AES (Advanced Encryption Standard) is conceived as the replacement of DES, as DES is now considered unsafe for many applications. It is also a symmetric-key algorithm and a block cipher.

i. Key expansion

Similar to DES, AES uses 'key expansion' to generate round keys for each round in encryption or decryption. The key process is Rijndael's key schedule, including rotating 32-bit words, Rcon and S-box transformations. [3][4][5]

ii. Encryption

According to the generated round keys and given plaintext, AES performs 10 rounds for 128-bit keys, 12 rounds for 192-bit keys and 14 rounds for 256-bit ones. The encryption can be divided to 3 stages:

1. InitialRound

   a) AddRoundKey: each byte of the state is combined with a block of the round key using bitwise XOR.

2. Rounds

   a) SubBytes: a non-linear substitution step where each byte is replaced with another according to a lookup table.
   b) ShiftRows: a transposition step where the last three rows of the state are shifted cyclically a certain number of steps.
   c) MixColumns: a mixing operation on the columns of the state, combining the four bytes in each column.
   d) AddRoundKey

3. Final Round

   a) SubBytes
   b) ShiftRows
   c) AddRoundKey [3][4]

iii. Decryption

AES decryption is quite similar to its encryption, as it also consists of three stages as below:

1. InitialRound

   a) AddRoundKey

2. Rounds

   a) InvShiftRows
   b) InvSubBytes
   c) AddRoundKey
   d) InvMixColumns

3. Final Round

   a) InvShiftRows
   b) InvSubBytes
   c) AddRoundKey [3][4]

c) RSA

RSA is an asymmetric-key algorithm which is based on the difficulty of factoring the product of two very large prime numbers. It also divides data to chunks, but treats each chunk as a big integer and uses modular exponentiation to perform encryption and decryption. [6]

i. Generating key pairs

RSA needs two big prime numbers *p*, *q* to create key pairs. For the generation of each big prime number, RSA firstly generates a big odd number and then checks it by all prime numbers smaller than 1000. [7] If this big number is indivisible by all these small prime numbers, RSA will use about 32 rounds of Miller-Rabin primality test to check this big number. If one number can pass one Miller-Rabin test, the probability that it is composite is only 1/4.Therefore, if a big number can pass all 32 independent tests, the probability that it is still composite is smaller than 4^{-32}, which can be considered impossible. [8][9]

Once two prime numbers are generated, RSA computes $n = pq$, and $\varphi(n) = \varphi(p)\varphi(q) = (p - 1)(q - 1)$. Then it chooses an e as the public key exponent. Typically, e is the fifth Fermat number, 65537. Then RSA gets the modular multiplicative inverse of e (modulo $\varphi(n)$), d, which is the private key exponent, by the extended Euclidean algorithm. [10] A public key includes n and e, while a private key includes n and d. [6]

ii. Encryption and decryption

During the RSA encryption and decryption, each chunk will be processed as a very big integer. Denote the plaintext by m and the ciphertext by c, we can get these equations [6]:

$$c \equiv m^e \pmod n$$
$$m \equiv c^d \pmod n$$

Since the modular exponentiation is the key part of RSA encryption and decryption, we use Montgomery modular multiplication to convert most divisions in modular multiplications and modular exponentiations to simple binary shifts.

In our RSA implementation, we use 1024-bit key pairs and chunks, and use 2048 bits to store a big integer, so in most cases we do not need to consider overflow problems. However, when computing a modular exponentiation m^a, if the exponent a has 1024 bit, we have to perform multiplications for at least $\log(a) = 1024$ times, so the intermediate result must exceed the maximum number that can be represented by 2048 bits. Therefore, we have to reduce the intermediate result modulo n after every multiplication, which results in more than 1024 times of division. Since a division includes many loops of binary searches, additions, subtractions and multiplications, it is very expensive to perform so many divisions in a modular exponentiation.

When we try to compute $c \equiv ab \pmod n$, we can choose an R which is relatively prime to n, then transform a, b to the Montgomery domain:

$$\bar{a} \equiv aR \pmod n$$
$$\bar{b} \equiv bR \pmod n$$

Then we find a modular inverse of R (modulo n) by the extended Euclidean algorithm:

$$RR^{-1} \equiv 1 \pmod n$$

We can get:

$$\bar{c} \equiv cR \equiv (ab)R \equiv (aR \cdot bR)R^{-1} \equiv (\bar{a}\bar{b})R^{-1} \pmod n$$
$$c \equiv \bar{c}R^{-1} \pmod n$$

In our program, because both p and q are big prime numbers and $n = pq$, p, q and n are all odd numbers. Therefore, we can easily let $R = 2^{1024}$, which is obviously prime to n and larger than n.

If $ab \geq 2^{1024}$, we need to convert it to $c \equiv ab \pmod n$ and let $c < 2^{1024}$. We firstly convert them to Montgomery domain and get a coefficient k:

$$k \equiv -(\bar{a}\bar{b} \bmod R) \cdot n^{-1} \pmod R$$

Then let $\bar{c} = \bar{a}\bar{b} + kn$:

$$\begin{aligned}
\bar{c} &= \bar{a}\bar{b} + kn \equiv \bar{a}\bar{b} \pmod n \\
\bar{c} &= \bar{a}\bar{b} + kn \\
&\equiv \bar{a}\bar{b} - (\bar{a}\bar{b} \bmod R) \cdot nn^{-1} \pmod R \\
&\equiv \bar{a}\bar{b} - (\bar{a}\bar{b} \bmod R) \pmod R \\
&\equiv 0 \pmod R
\end{aligned}$$

$\bar{c} \equiv 0 \pmod R$, so we can just divide it by R to get c. Because $R = 2^{1024}$, the division can be replaced by a shift:

$$\begin{aligned}
c &\equiv \bar{c}R^{-1} \pmod n \\
&\equiv \bar{c} / R \pmod n \\
&\equiv (\bar{c} >> 1024) \pmod n
\end{aligned}$$

In our modular exponentiation function, we firstly convert the base to Montgomery domain and then replace all divisions by shifts. After the exponentiation is completed, we convert the result back from Montgomery domain. [11][12]

We verify the correctness of our Montgomery modular multiplication and modular exponentiation by Python because it supports computing very large integers. We find that all the results of our algorithm are correct and it is far faster than Python so that we can hardly compare their speeds due to their significantly different orders of magnitude.

2. Compression algorithms

a) Context-based Compression: RLE & LZ77

Context-based compression takes advantage of context, in other words, encoding incoming strings using the information obtained from the completed parts.

Run-length encoding (RLE) classifies the data as two types: repeated strings and non-repeated strings. Each character is assigned a group based on whether it is the same character as its previous one or not. Only the repeated strings can be compressed into one-byte length code and one-byte character. Therefore, this algorithm is better suited for compression of data with large repetition ratio. [13]

LZ77, on the other hand, encodes a string by searching the identical string which has been already encoded within a sliding window. It outputs a triple which records the offset and length of the encoded string in the window followed by one character to be encoded in the next round. The compression takes place in the encoding of the triple, which uses Golomb code or gamma code. [14]

i. Compression

During the compression of RLE, we simply count the number of repeated and non-repeated characters in a serial way. We consider a character repeated only if it appears more than twice consistently. And we firstly output the indicator bit to determine whether the next character is repeated or not, followed by the length of the string. Then we output the exact repeated character or non-repeated string. [13]

The compression of LZ77 can be divided into four steps:

1. Starting from the current position of unencoded string, try to find its longest matched string within the sliding window, if found, go to step 2, otherwise proceed to step 3. If we get to the end of file, go to step 4.
2. Output the triple (off, len, c) where “off” is the offset towards the beginning of the sliding window, “len” is the length of matched string, “c” is the following character. Then the window slides across len + 1 characters, and proceed to step 1.
3. Output the triple (0, 0, c) where “c” is the following character. Then the window slides across one character, and proceed to step 1.
4. Encode the output triple of which the second element is encoded by Golomb or gamma code. Write the code into the compression file.

In order to find the longest matched string, we use a table list to record all positions of two-byte prefix appeared and an array to record the head position of this prefix. In this way, we only need to search for the possible matched strings at these recorded points.

In this project, we use gamma code to encode the triple. Suppose we encode the character x. Then we have q = int (log 2x). The first part of gamma code is q 1s plus one 0, and the second part is a q-bit binary number, the value of which is equal to x - 2q. [14]

ii. Decompression

The decompression of RLE is the entire opposite process of compression. It reads an indicator bit followed by 7-bit length, and then outputs either one character repeatedly or a non-repeated string. [13]

For LZ77 decompression, it is actually a process to reconstruct the window. As we decode the triple, the string will slide into the window, then following triples will be decoded based on the new data in the window. [14]

b) Dictionary-based Compression: Huffman & LZW

Dictionary-based compression keeps a dictionary which is used to transform each character into another representation leading to an overall compression. The dictionary here is global. Encoding process is more like to look up a dictionary in this kind of compression algorithms.

Huffman algorithm sets up a dictionary based on the frequency of each character in a file. The code is longer for less frequent characters and shorter for more frequent characters so that the size of the file after compression will become smaller in overall. [15]

LZW is not a very typical dictionary-based compression. It actually sets up a dictionary for prefixes. Strings are encoded using the longest matched prefix index, and new string will be added into the dictionary as a new prefix, which will be used for future encoding. This means that the dictionary will be dynamically updated in the encoding process. [16][17]

i. Compression

During the compression of the Huffman algorithm, we firstly count the frequency of each character. Then we set up a Huffman tree. Two least frequent characters will be selected and assigned one-bit codes, and then their frequency will be added to form a new effective one. Repeat this process until there is only one frequency left as the root. Then we go from the root to each leaf, and output the code along the path to the corresponding character. Finally we get the codebook of each character. When we compress a file, we simply transform each character to the code stored in the codebook. Aside from the compression file, the codebook will be stored as a file too. In our experiments, Huffman compression is mainly tested on plain text files, which usually has no Huffman code longer than 8-bit, so we use unsigned chars to store each code, which might be not suitable for general binary files. [15]

For LZW compression, we firstly initialize a dictionary of certain size, and set the first 256 prefixes as the basic 256 characters. When we encode a string, we search the longest matched prefix of it in the dictionary. If found, we output the prefix index, and add the current prefix plus one new character into the dictionary as a new prefix. Then we go on encoding the next character. [16][17]

ii. Decompression

Decompression of Huffman is to simply transform the bit stream into the original character based on the codebook. Since each prefix is unique, we do not need to know the exact length of each character in the compression file in advance. [15]

LZW, on the other hand, initializes a dictionary of certain size, and sets the first 256 prefixes as the basic 256 characters just like the compression process. Then it reads out an index from the compression file, and puts the corresponding prefix from the dictionary into the decompression file, and at the same time, if the former prefix is not null, it adds this former prefix plus the first character of the current prefix into the dictionary as a new prefix. LZW decompression repeats this process until all the indices in the compression file are decoded. In our implementation, we keep a 65536-byte dictionary to store all the prefixes. [16][17]

III. Parallel Implementations

1. Cryptographic algorithms

a) Generalities

DES, AES and RSA will all divide data to many chunks and process them separately. Hence, it is very natural for us to parallelize these algorithms by the chunks they create. Furthermore, we can use CUDA's stream mechanism to transmit chunks between the host memory and the GPU device, in order to increase the data throughput. From the graph below we can easily find that the times of host-device memory transmissions can be overlapped by the kernel executions.

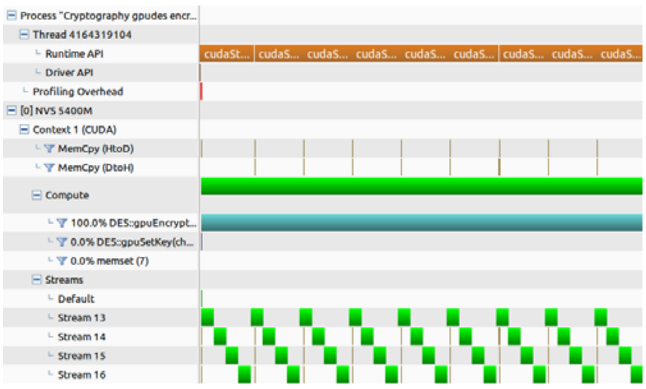


Figure 3. Visual profile of GPU-based DES

Besides these two levels of parallelization, we also research these algorithms themselves, and implement the parallelization of their operations in single chunks.

In order to improve the efficiency of the GPU-accelerated cryptography, we put all lookup tables into the constant memory on the GPU device, and process the plaintext or ciphertext in the shared memory during the entire processes of encryption and decryption.

b) Symmetric-key algorithms: DES and AES

DES and AES are very similar to each other, as they are both based the manipulation of bits or bytes in each chunk. Hence, we can put each bit or byte in one separated thread, so each step of bit / byte manipulation can be processed concurrently.

We can use AES as an example. In one round of AES encryption, AES performs SubBytes, ShiftRows, MixColumns and AddRoundKey. All these algorithms are applied on the 16 bytes in one chunk, so we can combine them to one function and then put them into 16 threads.

```
void CpuAES::encryptBlock(char *in, char *out) {
    memcpy(out, in, 16);
    addRoundKey(out, 0);
    for(int i = 1; i < Nr; ++i) {
        subBytes(out);
        shiftRows(out);
        mixColumns(out);
        addRoundKey(out, i);
    }
    subBytes(out);
    shiftRows(out);
    addRoundKey(out, Nr);
}
```

Figure 4. AES encryption on CPU

```
__global__ void gpuEncrypt(char *in, char *out, char
*w, int Nr) {
    char tmp;
    int times = (blockIdx.y * gridDim.x + blockIdx.x) *
blockDim.y + threadIdx.y;
    int ty = threadIdx.y;
    int tx = threadIdx.x;
    extern __shared__ char allBytes[];
    volatile char *bytes = allBytes + ty * 16;
    volatile char *buffer = bytes + (tx & 12);
    // Add round key
    bytes[tx] = in[tx + times * 16] ^ w[tx];
    int row, col;
    for (int i = 1; i < Nr; ++i) {
        // Sub bytes
        row = (bytes[tx] >> 4) & 15;
        col = bytes[tx] & 15;
        bytes[tx] = d_SBOX[row][col];
        // Shift rows
        col = tx & 3;
        bytes[tx] = bytes[((col << 2) + tx) & 15];
        // Mix columns
        tmp = d_MUL02[(unsigned char) buffer[col]] ^
d_MUL03[(unsigned char) buffer[(col + 1) & 3]] ^
buffer[(col + 2) & 3] ^ buffer[(col + 3) & 3];
        // Add round key
        bytes[tx] = tmp ^ w[(i << 4) + tx];
    }
    // Sub bytes
    row = (bytes[tx] >> 4) & 15;
    col = bytes[tx] & 15;
    bytes[tx] = d_SBOX[row][col];
    // Shift rows
    col = tx & 3;
    tmp = bytes[((col << 2) + tx) & 15];
    // Add round key
    out[tx + times * 16] = tmp ^ w[(Nr << 4) + tx];
}
```

Figure 5. AES encryption on GPU

From the code above, we can find that even if there are data dependency between bytes in each chunk, AES will synchronously perform each step of operation. Furthermore, as the size of one AES chunk is 16 bytes, we can use threads in one single warp to process them, which enables us to eliminate all barriers (syncthreads) in this encryption function.

c) Public-key algorithm: RSA

RSA is based on the calculation of big integers. In our implementation, we use 2048-bit big integers, which are arrays of unsigned long longs, so each digit in a big integer is 64 bits and there are 32 digits in one integer. We succeed in parallelizing all basic arithmetic operations on these digits.

i. Arithmetic operations of big integers

Addition / Subtraction: Consider the addition of two 32-digit number, the ordinary method is to add them from the least significant digit, get the result and carry of the addition on each digit and bring the carry to the higher digit. This operation is serial. But we can parallelize them by completely ignoring the carries and computing the addictive results of all digits concurrently at first. For each digit, we use a Boolean variable to record whether there is a carry form this digit.

Partial Result	1010	0111	1111	1111	0011
Carry	0	1	0	0	1
Final Result	1011	1000	0000	0000	0011

Figure 6. Example of parallelizable addition

We can use digits of 4 bits as an example. After the in-place addition, we can get the partial results without considering the carries. Then, we can let the thread of each digit which produces a carry scan forward.

For each loop of a scan, the next high-order digit will be increased by 1. Then the thread will check whether this digit becomes 0, which indicates it is previously $(1111)_2$. If this digit is 0 now, the scan will be continued.

Obviously, if a digit in the partial result is $(1111)_2$, its carry cannot be 1 as the sum of two 4-bit number cannot be $(11111)_2$. Hence, no scan can pass this digit and no digit will be increased by the scans of more than one threads, which ensures the concurrency of the addition. The subtraction operation is very similar to the addition. When scanning for the carries, each thread will check whether the next high-order digit becomes $(1111)_2$.

Multiplication: There are 32 rounds in the multiplication of two 32-digit big integers. In each round, one digit in the longer number will be multiplied by all digits in the other number. These 32 multiplications can be naturally parallelized.

Because we use one unsigned long long variable to store each digit, when multiply two digits from both operands, there is no space to store the higher 64-bit in the product. Therefore, we firstly divide these two digit into two 32-bit halves respectively, denote the four halves as $high1$, $low1$, $high2$, $low2$, and then do the following computation:

$$
\begin{aligned}
& product = ((high1 << 32) + low1)((high2 << 32) + low2) \\
& = (high1 \cdot high2 << 64) + ((high1 \cdot low2 + high2 \cdot low1) << 32) \\
& + low1 \cdot low2 \\
& = (carry1 << 64) + ((middle1 + middle2) << 32) + cur1 \\
& = ((carry1 + (middle1 >> 32) + (middle2 >> 32)) << 64) \\
& + ((middle1 \,\&\, (2^{32} - 1)) + (middle2 \,\&\, (2^{32} - 1)) + cur1) \\
& = ((carry1 + (middle1 >> 32) + (middle2 >> 32) \\
& + (((middle1 \,\&\, (2^{32} - 1)) + (middle2 \,\&\, (2^{32} - 1))) >> 32)) << 64) \\
& + (((middle1 + middle2) \,\&\, (2^{32} - 1)) + cur1) \\
& = ((carry1 + carry2) << 64) + (cur1 + cur2)
\end{aligned}
$$

Based the equations above, we can guarantee that there are no overflow problems with these generated $carry1$, $carry2$, $cur1$ and $cur2$ variables. In each round of the multiplication, we generate 32 tuples of these four results in 32 separated threads. Then we use our parallelized addition to add them to the final result.

We once considered using CUDA's fast Fourier transformation library to reduce the temporal complexity of multiplication from $O(n^2)$ to $O(n\log n)$. However, the parameter n in our program is a fixed value, 32, so n / log n is only 6.4. Considering the potential increase of the complexity coefficient and the inaccuracy of floating points, we still use the traditional method of multiplication.

High-leveled arithmetic operations: As the basic arithmetic operations are parallelized in our GPU implementation, the high-leveled operations, like division, modular multiplication (Montgomery modular multiplication) and modular exponentiation are also accelerated by parallelization.

We can use division as an example. For the high-precision division, we firstly compute the length of the dividend and the divisor, then use binary searches to find each digits of the quotient. In each step in one binary search, we parallel multiply a tentative digit by the divisor, and compare it to the remaining part of the dividend. After find one digit of the quotient, we parallel subtract the product of this digit and the divisor and then use binary search to find the next digit.

In the process of division, there are parallelized stages, like multiplication and subtraction, as well as serial stages, like computing the lengths of big integers, and comparing two big integers. In order to avoid bank conflicts, we always use the first thread in one chunk to process the serial parts, and store the results in the shared memory. After each serial stage, a barrier will be called to keep all threads in one block synchronized.

ii. Miller-Rabin primality tests

As we have mentioned before, RSA employs more than 30 rounds of Miller-Rabin test to guarantee the primality of generated big numbers. It is possible to perform all tests concurrently on GPU. In our GPU-based RSA, we divide the Miller-Rabin function to two kernel function, one is the initialization and the other is the operation in one loop of Miller-Rabin. This is because that, if we use a single kernel to perform all Miller-Rabin loops, it is very possible that this single kernel will be executed for a long time, as it will generate random big integers and perform many times of modular exponentiations. Therefore, it may be terminated by operating systems as they are in the timeout status, especially when the GPU device our program uses is also the display device.

2. Compression algorithms

a) Context-based Compression: RLE & LZ77

For RLE algorithm, we separate data into 256-byte chunks. Each chunk runs on an independent thread, so this thread will perform the counting of repeated and non-repeated strings. In order to find the starting point for each thread in the decompression process, we also record the length of code for each chunk after compression: outbuf[index*length + (lenbuf[index]++)] = *(src++); Otherwise, the threads cannot not know where to start decompression after reading the compressed byte streams into the shared memory.

For LZ77 algorithm, we also use these techniques. However, in order to record every position that a two-byte prefix appears in the window, we maintain a list table of all positions and an array of head positions for all threads. Therefore, the memory used by one thread will be 524,288 bytes, which definitely exceeds the maximum size of memory application in kernel functions and limits the sizes of our input files to at most 2,048 bytes. In order to solve this problem, we proposed a solution which is to search for matched strings using the KMP algorithm, but have not applied it to our implementation due to the limited time.

b) Dictionary-based Compression: Huffman & LZW

For Huffman algorithm, shared memory and atomic addition are used to count the frequency of each character. Building Huffman tree can be hardly parallelized, since each round of the search for two smallest frequencies is highly dependent on previous calculations. For the parallelization of Huffman, we divide our data into chunks, and let each thread perform compression to one chunk. We do not assign each character to a single thread, because we have to record the length of compressed code for each thread, otherwise we will not be able to get the starting point of each thread when decoding the compressed file. Hence, the parallel unit of Huffman is still a chunk as all the previous algorithms do.

For LZW algorithm, the dictionary is stored in the global memory since it has 65,536 elements, which exceeds the size limit of the shared memory used for each block. We assign a certain space in the dictionary for each thread, and parallel perform prefix encoding and decoding within that space.

IV. EXPERIMENTS

1. Cryptographic algorithms

We use six different files to evaluate the performance improvement on our GPU-based cryptographic algorithms. These files are of various types and their sizes are increased one by one. The description of these files are listed below:

Name	Size	Type	Description
README.txt	2.71 KB	Plain Text	Readme for the cryptographic part
resume.pdf	96.1 KB	PDF Document	Yuqing Guan's resume
AES	592 KB	Executable	Independent executable of CPU- and GPU-based AES
stillife.exr	3.85 MB	Image	Images from HW3
GPU_Proj_Crypto_Part.zip	18.1 MB	Compressed Archive	Compressed file of the cryptographic part
primeV7.in	78.9 MB	Plain Text	A list of all integers ≤ 1e7

Figure 7. File descriptions for cryptographic experiments

We use a CLIC machine, Damascus, to test all algorithms. In the presentation, we used test results on Pretoria, but then mistakenly deleted the result file. Therefore, we re-test our algorithms on Damascus.

Name	CPU Encry-ption	GPU Encry-ption	Impro-vement	CPU Decry-ption	GPU Decry-ption	Impro-vement
README.txt	10.0ms	6.15ms	1.63x	10.0ms	5.95ms	1.68x
resume.pdf	220ms	12.5ms	17.6x	220ms	18.4ms	12.0x
AES	1.37s	33.8ms	40.6x	1.36s	33.6ms	40.5x
stillife.exr	8.94s	195ms	45.7x	9.02s	202ms	44.7x
GPU_Proj_Crypto_Part.zip	41.4s	958ms	43.2x	41.8s	799ms	52.3x
primeV7.in	3.73min	4.35s	51.5x	3.77min	4.65s	48.6x

Figure 8. Performance improvement of GPU-based DES

Name	CPU Encry-ption	GPU Encry-ption	Impro-vement	CPU Decry-ption	GPU Decry-ption	Impro-vement
README.txt	0.00ms	78.4ms	N/A	0.00ms	77.7ms	N/A
resume.pdf	30.0ms	81.1ms	0.37x	30.0ms	78.9ms	0.38x
AES	170ms	86.5ms	1.96x	170ms	86.2ms	1.97x
stillife.exr	1.07s	145ms	7.36x	1.06s	143ms	7.43x
GPU_Proj_Crypto_Part.zip	5.06s	390ms	13.0x	5.03s	374ms	13.4x
primeV7.in	27.3s	2.47s	11.1x	28.2s	2.57s	11.0x

Figure 9. Performance improvement of GPU-based AES

Name	CPU Encry-ption	GPU Encry-ption	Impro-vement	CPU Decry-ption	GPU Decry-ption	Impro-vement
README.txt	780ms	761ms	1.02x	3.83s	1.49s	2.57x
resume.pdf	26.5s	5.15s	5.15x	2.17min	11.2s	11.6x
AES	2.70min	31.3s	5.17x	13.2min	1.13min	11.7x
stillife.exr	17.5min	3.32min	5.26x	1.46h	7.41min	11.8x
GPU_Proj_Crypto_Part.zip	1.34h	15.5min	5.18x	6.85h	34.3min	12.0x
primeV7.in	6.05h	1.12h	5.38x	1.48day	2.49h	14.3x

Figure 10. Performance improvement of GPU-based AES

From the test result, we can find that GPU-acceleration can significantly improve the performances of DES, AES and RSA algorithms. When the test file becomes larger, the performance improvement will become very obvious. If the test file is larger than 1 MB, GPU-based cryptography can be 10~50 times faster than its CPU-based counterpart.

We can also find that, public-key cryptography is typically much slower than symmetric-key cryptography. As a result, the RSA algorithm is usually used to encrypt transmitted symmetric keys, instead of entire files.

Method	CPU Time	GPU Time	"Improvement"
Generating Key Pairs	40.26s	19.51min	0.0344x

Figure 11. Performance comparison of RSA key generation

Although CUDA platform can significantly improve the performance of cryptography, when we use a GPU device to generate key pairs for RSA, we can find that its performance is even worse than CPU-based RSA. It is probable that generating and verifying very big prime numbers involves too complex computation, such as dividing the generated big integers by all prime numbers smaller than 1000, as well as the 32 rounds of Miller-Rabin tests. When parallel computing on GPU involves too much complex computation, its performance can be worse than CPU since the latter one has many special optimizations for these computations.

2. Compression algorithms

We test our compression algorithms using text files of different sizes on a CLIC machine, Baghdad.

Name	Size	Description
test1.txt	0.23 KB	Random Strings
test2.txt	9.07 KB	Jihan Li's cover letter
test3.txt	182 KB	LZ77 Wikipedia
test4.txt	318 KB	Bibliography
test5.txt	732 KB	Markov Chains textbook

Figure 12. File descriptions for compression experiments

All test results are listed below:

Name	CPU Comp-ression	GPU Comp-ression	Ratio	CPU Decom-pression	GPU Decom-pression	Ratio
test1.txt	0.00ms	5.00ms	N/A	0.00ms	4.00ms	N/A
test2.txt	1.00ms	8.00ms	0.125x	0.00ms	1.00ms	N/A
test3.txt	3.00ms	21.0ms	0.143x	1.00ms	9.00ms	0.11x
test4.txt	7.00ms	26.0ms	0.27x	1.00ms	1.00ms	1.00x
test5.txt	17.0ms	55.0ms	0.31x	2.00ms	2.00ms	1.00x

Figure 13. Performance comparison of RLE

Name	CPU Comp-ression	GPU Comp-ression	Ratio	CPU Decom-pression	GPU Decom-pression	Ratio
test1.txt	0.00ms	62.0ms	N/A	0.00ms	5.00ms	N/A

Figure 14. Performance comparison of LZ77

Name	CPU Comp-ression	GPU Comp-ression	Ratio	CPU Decom-pression	GPU Decom-pression	Ratio
test1.txt	1.00ms	10.0ms	0.1x	0.00ms	0.00ms	N/A
test1.txt * 2	1.00ms	12.0ms	0.08x	0.00ms	1.00ms	N/A
test1.txt * 5	1.00ms	15.0ms	0.07x	1.00ms	1.00ms	1.00x
test1.txt * 10	1.00ms	22.0ms	0.045x	1.00ms	1.00ms	1.00x
test1.txt * 20	2.00ms	41.0ms	0.05x	2.00ms	2.00ms	1.00x

Figure 15. Performance comparison of Huffman

Name	CPU Comp-ression	GPU Comp-ression	Ratio	CPU Decom-pression	GPU Decom-pression	Ratio
test1.txt	0.00ms	4.00ms	N/A	0.00ms	1.00ms	N/A
test2.txt	6.00ms	1.13s	0.005x	1.00ms	0.25s	0.004x

Figure 16. Performance comparison of LZW

From the test results we can find that GPU-based compression algorithms are normally even slower than CPU-based versions.

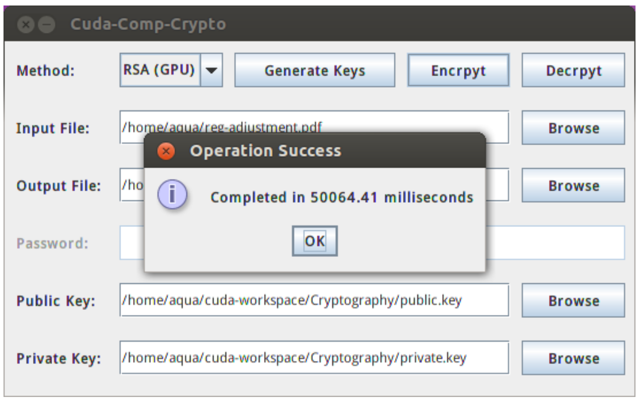
Obviously, in both compression and decompression process of context-based algorithms, each thread will run a serial algorithm by for loops in terms of counting issue. Since these threads start at different positions in the input array, we cannot make the memory access coalesced. This constrains the use of threads. Also, if we use too many threads while keeping the same chunk size, the shared memory will not be able to hold so much input data, so the data have to be stored in the global memory, which extremely slows down the frequent memory access. From this point, the advantages of CUDA parallelization cannot be fully utilized.

For LZW, we store the dictionary in the global memory. The frequent access of the dictionary extensively slows down the speed of process. Hence, GPU-base LZW is far slower than CPU-based LZW.

We can see that LZ77 and LZW have very large gaps between the time costs of their CPU versions and GPU versions. This is because that they require much more computation uncoalescedly accessing the global memory than other two algorithms.

V. Applications

Because GPU-based parallelization can significantly accelerate cryptographic algorithms, we develop a command-line and a GUI cryptographic application, both of which can support CPU- and GPU-based DES, AES and RSA encryption / decryption, as well as RSA key pair generation.


Cuda-Comp-Crypto

Method: RSA (GPU)

Input File: /home/aqua/reg-adjustment.pdf

Output File: /ho

Password:

Public Key: /home/aqua/cuda-workspace/Cryptography/public.key

Private Key: /home/aqua/cuda-workspace/Cryptography/private.key

Operation Success

Completed in 50064.41 milliseconds

Figure 17. GUI cryptographic application

For our GUI program, we use Java Swing to build a user interface, and connect it to our native CUDA/C++ code by Java Native Interface (JNI). In this mode, we compile our native code to a shared object (.so) and then make the Java virtual machine load it. We find that all constant memory objects declared in separated .cu files cannot be accessed by JNI, even though they can work correctly in the native C++-based command-line tool. Hence, we copied our CUDA code into a new project, relocated the declarations of constant memory objects to the main .cu file that JNI will use, and then let our program pass these objects to kernel functions by their pointers got by cudaGetSymbolAddress.

Both these two applications are included in a compressed archive called 'GPU_Proj_Crypto_Part.zip' in our submitted code. Detailed instructions for our applications are given in a readme file which is in the 'GPU_Proj_Crypto_Part' subdirectory of this zip file.

VI. Conclusion

Based on the study on three cryptographic algorithms, DES, AES and RSA, we completed both CPU and GPU implementations of these algorithms and compared their performances on CLIC machines. Accelerated by three levels of parallelization, our GPU-based implementations are typically 10~50 times faster than the CPU-based implementations for input files larger than 1 MB. When the sizes of input files increase, the performance improvements become even more obvious.

On the other hand, according to our experiments, compression algorithms are less suited for parallel processing. Unlike cryptographic algorithms, there are much more data dependency in the processes of compression and decompression. Also, for context-based and dictionary-based compression, large-sized dictionaries and sliding windows are employed which cannot be put into the shared memory due to its limited capacity, so the memory accesses are significantly slowed down.

Because our GPU-based cryptographic algorithms are explicitly faster than their CPU counterparts, we develop two different cryptographic tools for practical use. One is a simple native C++-based command-line tool which supports 14 different types of instructions, and the other one is a GUI application which uses a Java Swing user interface and JNI to call our native C++ programs.

When testing our applications, please notice that if you use the current display device to execute our CUDA program, it is possible that your operating system will limit the execution time of our kernel functions and terminate them before they complete their work. It is highly suggested to use a computer with two graphics card and choose the non-NVIDIA card as the main display device, or remotely login a machine by SSH to test our applications.

Contributions:

1. Yuqing Guan:
   a) Cryptographic algorithms
   b) Command-line and GUI applications
2. Jihan Li
   a) Compression algorithms

REFERENCES

[1] Wikipedia, 'Data Encryption Standard', 2015. [Online]. Available: http://en.wikipedia.org/wiki/Data_Encryption_Standard.

[2] Cnblogs.com, 2012. [Online]. Available: http://www.cnblogs.com/imapla/archive/2012/09/07/2674788.html.

[3] Wikipedia, 'Advanced Encryption Standard', 2015. [Online]. Available: http://en.wikipedia.org/wiki/Advanced_Encryption_Standard.

[4] GitHub, 'SongLee24/Aes_and_Des', 2015. [Online]. Available: https://github.com/SongLee24/Aes_and_Des/blob/master/Aes/Aes.cpp.

[5] GitHub Gists, 'bonsaiviking/aes.py', 2013. [Online]. Available: https://gist.github.com/bonsaiviking/5571001.

[6] Wikipedia, 'RSA (cryptosystem)', 2015. [Online]. Available: http://en.wikipedia.org/wiki/RSA_%28cryptosystem%29.

[7] Primes.utm.edu, 'Lists of small primes (less than 1000 digits)', 2015. [Online]. Available: https://primes.utm.edu/lists/small/.

[8] Wikipedia, 'Miller–Rabin primality test', 2015. [Online]. Available: http://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test.

[9] Yuqing Guan’s homework for 'Introduction to Information Security', Fall 2013, Peking University

[10] w. David Ireland, 'The Euclidean Algorithm and the Extended Euclidean Algorithm', *Di-mgt.com.au*, 2015. [Online]. Available: http://www.di-mgt.com.au/euclidean.html.

[11] Wikipedia, 'Montgomery modular multiplication', 2015. [Online]. Available: http://en.wikipedia.org/wiki/Montgomery_modular_multiplication.

[12] Alicebob.cryptoland.net, 'Understanding the Montgomery reduction algorithm | Alice and Bob in Cryptoland', 2009. [Online]. Available: http://alicebob.cryptoland.net/understanding-the-montgomery-reduction-algorithm/.

[13] Blog.csdn.net, 2013. [Online]. Available: http://blog.csdn.net/anzelin_ruc/article/details/9180525.

[14] Blog.chinaunix.net, 2012. [Online]. Available: http://blog.chinaunix.net/uid-17240700-id-3347894.html.

[15] Wikipedia, 'Huffman coding', 2015. [Online]. Available: http://en.wikipedia.org/wiki/Huffman_coding.

[16] Cs.cf.ac.uk, 'LZW Data Compression', 1997. [Online]. Available: http://www.cs.cf.ac.uk/Dave/Multimedia/Lecture_Examples/Compression/lzw/lzw_docs/LZW_Data_Compression.htm.

[17] Blog.chinaunix.net, 2012. [Online]. Available: http://blog.chinaunix.net/uid-17240700-id-3347894.html.