AES 算法和 OpenMP 与 CUDA 的并行优化

19335174 施天予

2021 年 12 月 19 日

摘　要　密码学中的高级加密标准 AES（Advanced Encryption Standard），是一种对称加密体制，是美国联邦政府采用的一种区块加密标准。本文分析了 AES 算法加密解密的过程，实现了基于 C++ 的 AES 加密解密程序，并使用 OpenMP 和 CUDA 编程分别实现了 AES 算法的 CPU 与 GPU 并行化版本。实验结果表明，使用 OpenMP 和 CUDA 并行化后的 AES 算法程序，性能有着显著的提升。

关键词　AES 算法；OpenMP；CUDA；并行优化

1　引言

由于 DES 的有效密钥长度为 56 位，密钥大小和块大小短，很容易被穷举搜索，缺乏安全性。因此人们设计了 AES 用来替代原先的 DES，目前已经被多方分析且广为全世界所使用。不同于它的前任标准 DES，AES 使用的是置换-组合架构，而非 Feistel 架构。AES 在软件及硬件上都能快速地加解密，相对来说较易于实现，且只需要很少的存储器。AES 算法从很多方面解决了令人担忧的问题。实际上，攻击数据加密标准的那些手段对于 AES 算法本身并没有效果。如果采用真正的 128 位加密技术甚至 256 位加密技术，蛮力攻击要取得成功需要耗费相当长的时间。

OpenMP 是由 OpenMP Architecture Review Board 牵头提出的，用于共享内存并行系统的多处理器程序设计的一套指导性编译处理方案。OpenMP 提供了对并行算法的高层的抽象描述，程序员通过在源代码中加入专用的 pragma 来指明自己的意图，由此编译器可以自动将程序进行并行化，并在必要之处加入同步互斥以及通信，具有强大的灵活性。

CUDA（Compute Unified Device Architecture），是显卡厂商 NVIDIA 推出的运算平台。CUDA 是一种由 NVIDIA 推出的通用并行计算架构，该架构使 GPU 能够解决复杂的计算问题。通过使用 GPU 高性能并行多线程能力，解决大规模计算的模型已被广泛应用在密码破译、深度学习等各项领域。CUDA 线程数目通常能达到数千个甚至上万个，通过 CUDA 编程实现 GPU 上的并行计算，性能可以得到大幅度的提升。

2　AES 算法原理

2.1　算法综述

AES 为分组密码，每次加密一组数据，直到加密完整个明文。AES 的分组长度是 128 位，每个分组为 16 个字节。密钥的长度可以使用 128 位、192 位或 256 位。密钥的长度不同，加密轮数也不同。

AES	密钥长度	加密轮数
AES-128	128	10
AES-192	192	12
AES-256	256	14

Table 1: AES 密钥长度与加密轮数

以 AES-128 为例，密钥的长度是 128 位，加密轮数是 10 轮。在 AES 的加密过程中，会执行一个轮函数，并且执行 10 次。这个轮函数的前 9 次

执行的操作是一样的，第 10 次不进行列混合操作。

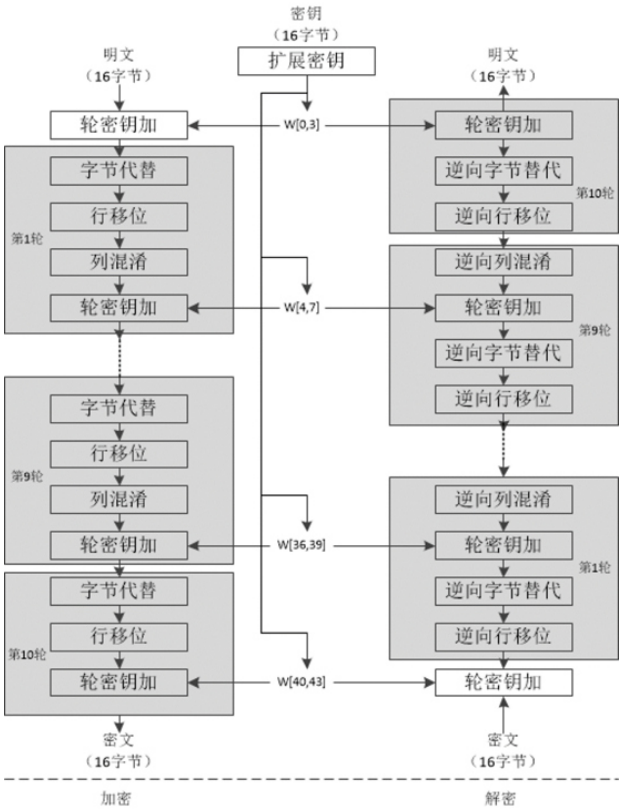

Figure 1: AES 算法流程图

AES 的 128 位输入明文分组 P 和输入密钥 K 都被分成 16 个字节，明文分组用字节为单位的正方形矩阵描述，称为状态矩阵。在算法的每一轮中，状态矩阵的内容不断变化，最后结果作为密文输出。

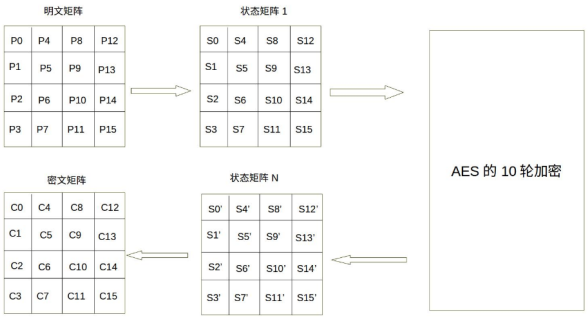

Figure 2: AES 算法的状态矩阵

AES 的核心就是实现一轮中的所有操作。下面分别介绍 AES 中一轮的 4 个操作和密钥扩展。轮函数的 4 个操作可以使输入位得到充分的混淆，而解密轮函数的 4 个操作需要逆向实现。

2.2 字节代换

AES 的字节代换其实就是一个简单的查表操作。AES 定义了一个 S 盒和一个逆 S 盒。

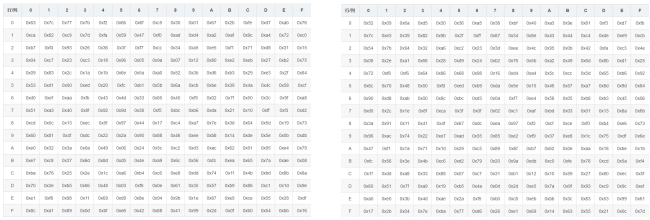

Figure 3: S 盒与逆 S 盒

状态矩阵中的元素按照下面的方式映射为一个新的字节：把该字节的高 4 位作为行值，低 4 位作为列值，加密时取出 S 盒，解密时取出逆 S 盒中对应的元素作为输出。

2.3 行移位

行移位是一个简单的左循环移位操作。当密钥长度为 128 比特时，状态矩阵的第 0 行左移 0 字节，第 1 行左移 1 字节，第 2 行左移 2 字节，第 3 行左移 3 字节。

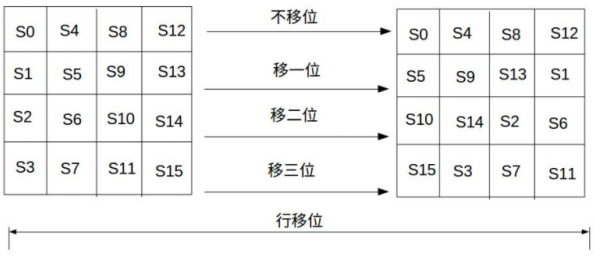

Figure 4: 行移位

解密时行移位的逆变换是将状态矩阵中的每一行执行相反的移位操作，在 AES-128 中，状态矩阵的第 0 行右移 0 字节，第 1 行右移 1 字节，第 2 行右移 2 字节，第 3 行右移 3 字节。

2.4 列混合

列混合变换是通过矩阵相乘来实现的，经行移位后的状态矩阵与固定的矩阵相乘，得到混淆后的状态矩阵。

$$\begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix}$$

Figure 5: 列混合

状态矩阵中的第 j 列的列混合可以表示为：

$$\begin{aligned} s'_{0,j} &= (2 * s_{0,j}) \oplus (3 * s_{1,j}) \oplus s_{2,j} \oplus s_{3,j} \\ s'_{1,j} &= s_{0,j} \oplus (2 * s_{1,j}) \oplus (3 * s_{2,j}) \oplus s_{3,j} \\ s'_{2,j} &= s_{0,j} \oplus s_{1,j} \oplus (2 * s_{2,j}) \oplus (3 * s_{3,j}) \\ s'_{3,j} &= (3 * s_{0,j}) \oplus s_{1,j} \oplus s_{2,j} \oplus (2 * s_{3,j}) \end{aligned}$$

Figure 6: 状态矩阵中第 j 列的列混合

这里矩阵元素的乘法和加法并不是通常意义上的乘法和加法。这种二元运算的加法等价于两个字节的异或，乘法则复杂一点。对于一个 8 位的二进制数来说，使用乘法乘以 (00000010) 等价于左移 1 位 (低位补 0) 后，再根据结果决定是否进行进行异或运算：如果最高位为 1 则进行异或运算，否则不进行。

逆向列混合变换可由下图的矩阵乘法定义，可以确定的是，逆变换矩阵同正变换矩阵的乘积恰好为单位矩阵。

$$\begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix}$$

Figure 7: 逆向列混合

2.5 轮密钥加

轮密钥加是将 128 位轮密钥同状态矩阵中的数据进行逐位异或操作。其中密钥中每个字 W[4i],W[4i+1],W[4i+2],W[4i+3] 为 32 位比特，即 4 个字节。轮密钥加可以看成是字逐位异或的结果，也可以看成字节级别或者位级别的操作。也就是说，可以看成 S0 S1 S2 S3 组成的 32 位字与 W[4i] 的异或运算。轮密钥加的逆运算同正向运算完全一致，因为异或的逆操作是其自身。

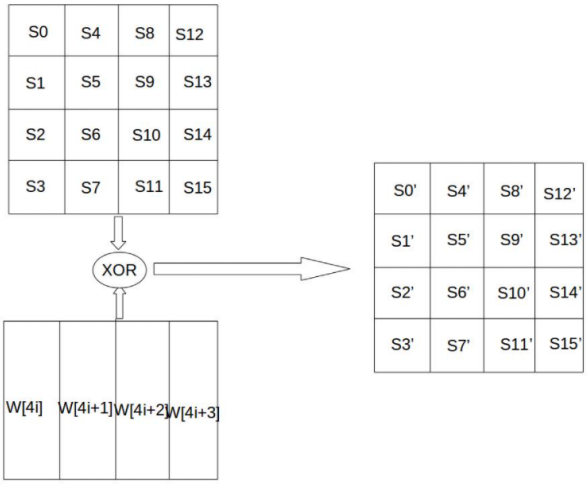

Figure 8: 轮密钥加

2.6 密钥扩展

除了轮函数中的 4 个操作，轮密钥加的生成算法密钥扩展也尤为重要。AES 首先将初始密钥输入到一个 4*4 的状态矩阵中，如下图所示

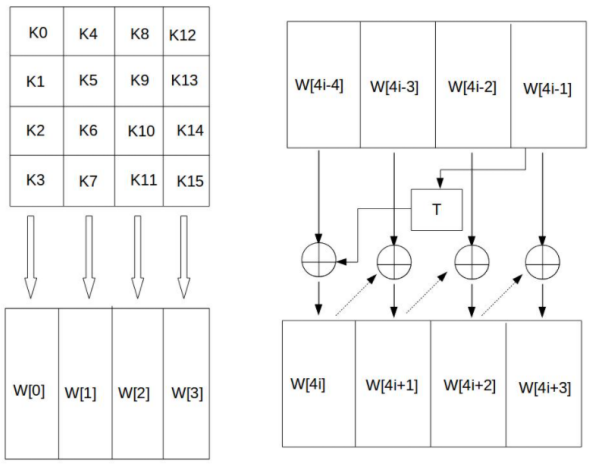

Figure 9: 密钥扩展

这个 4*4 矩阵的每一列的 4 个字节组成一个

字，矩阵 4 列的 4 个字依次命名为 W[0]、W[1]、W[2] 和 W[3]，它们构成一个以字为单位的数组 W。接着对 W 数组扩充 40 个新列，构成总共 44 列的扩展密钥数组。新列以如下的递归方式产生：

$$W[i] = \begin{cases} W[i-4] \oplus W[i-1], & i \ mod \ 4 \neq 0 \\ W[i-4] \oplus T(W[i-1]), & i \ mod \ 4 = 0 \end{cases}$$

函数 T 包含 3 个部分组成：

- 字循环：将 1 个字中的 4 个字节循环左移 1 个字节。
- 字节代换：对字循环的结果使用 S 盒进行字节代换。
- 轮常量异或：将前两步的结果同轮常量 Rcon[j] 进行异或，其中 j 表示轮数。

j	1	2	3	4	5
Rcon[j]	01 00 00 00	02 00 00 00	04 00 00 00	08 00 00 00	10 00 00 00
j	6	7	8	9	10
Rcon[j]	20 00 00 00	40 00 00 00	80 00 00 00	1B 00 00 00	36 00 00 00

Figure 10: 轮常量

3 AES 算法串行实现

首先使用 C++ 实现 AES 的串行加密解密算法，这里主要讲解几个关键的函数，受篇幅限制，解密的逆操作函数因为与加密的几个函数类似就不再展示。

- byteSubstitution 函数用于字节代换。将输入的 4*4 矩阵一一按照 S 盒进行代换。字节代换的逆操作按照逆 S 盒进行代换即可。

```
void byteSubstitution(unsigned char block
    ↪ [4][4]) {
   for (int i = 0; i < totalWords; i++)
       for (int j = 0; j < 4; j++)
       block[i][j] = getSBoxValue(block[i][
           ↪ j]);
}
```

- rowShifting 函数用于行移位。其实现也非常简单，第 0 行不变，第 1 行左移 1 位，第 2 行左移 2 位，第 3 行左移 3 位。同理行移位的逆操作只需往右移位即可。

```
void rowShifting(unsigned char block[4][4])
{
    char temp=block[1][0];
    block[1][0]=block[1][1];
    block[1][1]=block[1][2];
    block[1][2]=block[1][3];
    block[1][3]=temp;
    temp=block[2][0];
    block[2][0]=block[2][2];
    block[2][2]=temp;
    temp=block[2][1];
    block[2][1]=block[2][3];
    block[2][3]=temp;
    temp=block[3][0];
    block[3][0]=block[3][3];
    block[3][3]=block[3][2];
    block[3][2]=block[3][1];
    block[3][1]=temp;
}
```

- columnMixing 函数用于列混合，实现矩阵乘法。其中加法其实是异或操作，乘法 product 通过位操作实现。

```
#define product(x) ((x<<1)^(((x>>7)&1)*0x1b
    ↪ ))
void columnMixing(unsigned char block
    ↪ [4][4]) {
   unsigned char temp, temp2, t;
   for (int i = 0; i < 4; i++) {
       t = block[0][i];
       temp = block[0][i] ^ block[1][i] ^
           ↪ block[2][i] ^ block[3][i];
       temp2 = product(block[0][i]^block
           ↪ [1][i]);
       block[0][i] ^= temp2 ^ temp;
       temp2 = product(block[1][i]^block
           ↪ [2][i]);
       block[1][i] ^= temp2 ^ temp;
       temp2 = product(block[2][i]^block
           ↪ [3][i]);
```

```
        block[2][i] ^= temp2 ^ temp;
        temp2 = product(block[3][i]^t);
        block[3][i] ^= temp2 ^ temp;
    }
}
```

- addRoundKey 函数用于轮密钥加，128 位轮密钥同状态矩阵中的数据进行逐位异或，其逆操作也是如此。

```
void addRoundKey(int roundNo, unsigned char
    ↪  block[4][4]) {
    int val = roundNo * totalColumn * 4;
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            int indexVal = val + i*
                ↪ totalColumn + j;
            block[j][i] = block[j][i] ^
                ↪ roundKey[indexVal] ;
        }
    }
}
```

- keyExpansion 函数实现了密钥扩展，为每一轮的轮密钥加提供密钥。首先需要扩充 40 个新列，然后进行字循环、字节代换、轮常量异或。

```
void keyExpansion() {
    int bytePos=0;
    unsigned char compute[4];
    /*扩充40个新列*/
    for (; bytePos < totalWords*4; bytePos
        ↪ ++)
        roundKey[bytePos] = key[bytePos];
    bytePos = totalWords;
    for(; bytePos < 44; bytePos++) {
        for(int j = 0; j < 4; j++)
            compute[j] = roundKey[(bytePos-1)
                ↪  * 4 + j];
        if (bytePos%totalWords==0) {
            /*字循环*/
            int temp = compute[0];
            compute[0] = compute[1];
            compute[1] = compute[2];
            compute[2] = compute[3];
            compute[3] = temp;
            /*字节代换*/
            compute[0]=getSBoxValue(compute
                ↪ [0]);
            compute[1]=getSBoxValue(compute
                ↪ [1]);
            compute[2]=getSBoxValue(compute
                ↪ [2]);
            compute[3]=getSBoxValue(compute
                ↪ [3]);
            /*轮常量异或*/
            compute[0] = compute[0] ^
                ↪ roundConstant[bytePos/
                ↪ totalWords];
        }
        roundKey[bytePos*4+0] = roundKey[(
            ↪ bytePos-totalWords)*4+0] ^
            ↪ compute[0];
        roundKey[bytePos*4+1] = roundKey[(
            ↪ bytePos-totalWords)*4+1] ^
            ↪ compute[1];
        roundKey[bytePos*4+2] = roundKey[(
            ↪ bytePos-totalWords)*4+2] ^
            ↪ compute[2];
        roundKey[bytePos*4+3] = roundKey[(
            ↪ bytePos-totalWords)*4+3] ^
            ↪ compute[3];
    }
}
```

- 加密函数 encrypt 和解密函数 decrypt 如下。首先进行密钥扩展和轮密钥加，然后对明文/密文进行轮函数的 4 个操作，注意最后一轮不需要列混合。

```
void encrypt(unsigned char block[4][4]) {
    keyExpansion();
    addRoundKey(0, block);
    for (int i = 1; i <= totalRounds; i++) {
        byteSubstitution(block);
        rowShifting(block);
        if (i != totalRounds)
            columnMixing(block);
        addRoundKey(i, blocky);
    }
}
```

```
void decrypt(unsigned char block[4][4]) {
    keyExpansion();
    addRoundKeyInverse(totalRounds, block);
    for (int i = totalRounds-1; i >= 0; i--)
    {
        rowShiftingInverse(block);
        byteSubstitutionInverse(block);
        addRoundKeyInverse(i, block);
        if (i != 0)
            columnMixingInverse(block);
    }
}
```

4 AES 算法 OpenMP 实现

在之前的串行算法中，我们需要将明文/密文 128 位一组进行加密/解密，而每一组的运算过程都是独立的，不存在数据依赖，所以很容易让人想到将每一部分的运算工作并行实现，从而加快加密/解密的运行速度。

OpenMP 是一种用于共享内存的多线程并行编程模型，通过高阶指令，可以轻松地将串行程序改为并行程序。于是我们可以将 128 位一组的明文/密文充分利用多个线程进行加密/解密，最后再整合到一起即可。比如下面这段加密代码：

```
#pragma omp parallel private(block, block_count,
    ↪ thread_count, output) \
    shared(textTemp, filesize, total_blocks,
        ↪ result_final)
{
    block_count = omp_get_thread_num();
    thread_count = omp_get_num_threads();
    for (block_count = omp_get_thread_num();
        ↪ block_count < total_blocks;
        ↪ block_count+=thread_count) {
        createStateArray(block_count, filesize,
            ↪ textTemp, block);
        encrypt(block);
        createOutputArray(block_count, filesize,
            ↪ output, block);
    }
}
```

5 AES 算法 CUDA 实现

然而 CPU 受限于线程数的影响，使用 OpenMP 在 CPU 的并行编程并不能使运算得到充分的并行，因此我们当然也可以使用 GPU 进行 CUDA 编程，进一步提升运行速度。

GPU 有着高吞吐量、高计算能力、高并发线程数等特点，所以 GPU 非常适合用于计算任务。相比之下，CPU 是专为顺序串行处理而优化的几个核心组成。而 GPU 则由数以千计的更小、更高效的核心组成，这些核心专门为同时处理多任务而设计，可高效地处理并行任务。虽然 CPU 虽然每个核心自身能力极强，但其核心少，在并行计算上表现不佳；GPU 虽然其每个核心的计算能力不算强，但 GPU 的核心非常多，可以使用上千个线程同时处理多个计算任务，因此并行计算的效果更好。

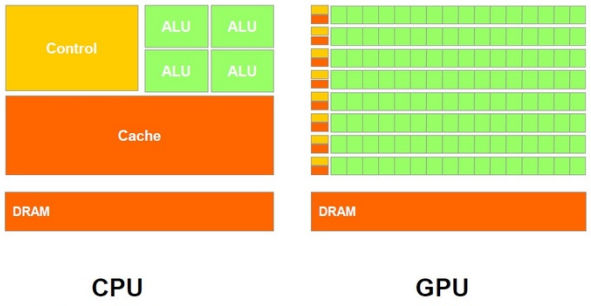

Figure 11: CPU 与 GPU 对比

CUDA 的并行计算需要依赖 CPU-GPU 的异构计算，即在 CPU 的 host 端进行串行处理，在 GPU 的 device 端进行并行计算。在 AES 算法中，我们可以先在 host 端完成文件的 IO 操作，之后将分组的数据块从 host 端拷贝到 device 端，在 GPU 上使用大量线程进行加密/解密的计算。由于 128 位一组的数据块没有数据依赖，我们可以在加密/解密的函数前加上 ___device___ 声明在 device 端运行即可。

CUDA 编程模型在 GPU 上需要为 device 端分配相应空间，再将数据从 host 端拷贝到 device 端，指定 gridsize 和 blocksize，在 device 端创建大量线程启动核函数进行并行计算，最后再将结果返

回到 host 端。部分加密过程的代码如下：

```
__global__ void encrypt_one_block(unsigned char
    ↪ *block, unsigned int numblock) {
    int id = blockIdx.x*blockDim.x + threadIdx.x;
    int offset = id * 16;
    if (id >= numblock) return;
    encrypt(block, offset);
}
cudaMalloc(&ddata,sizeof(unsigned char)*16);
cudaMemcpy(ddata, data, sizeof(unsigned char)
    ↪ *16, cudaMemcpyHostToDevice);
encrypt_one_block<<<blocks, threads>>>(ddata,
    ↪ numblock);
cudaThreadSynchronize();
cudaMemcpy(out_data, ddata,sizeof(unsigned char)
    ↪ *nbytes, cudaMemcpyDeviceToHost);
cudaFree(ddata);
```

6 实验结果与分析

6.1 实验环境

- Ubuntu 20.04.3
- gcc 9.3.0
- Intel® Core™ i7-9750H CPU @ 2.60GHz
- GTX TITAN X

6.2 性能分析

经过实验发现，在明文文件规模较小时，使用 GPU 并行算法并不能加速运算过程，反而会减慢速度，这是因为对 GPU 的数据拷贝以及线程管理会带来额外的开销。当问题规模增大时，计算部分的占比会增大，也就是说并行化部分占比会增加，GPU 并行计算的效果也会越来越出色。然而我使用的 OpenMP 是在 CPU 上进行并行计算，创建线程的开销相对较小，所以当问题规模较小时并行化的效果仍然明显。

为了比较使用 OpenMP 编程在 CPU 并行化以及使用 CUDA 编程在 GPU 并行化的加速效果，我测试了不同文件大小的明文文件，记录时间进行对比分析。首先是 OpenMP 不同线程数对运行速度的影响，可以发现由于我的 CPU 是 6 核 12 线程，所以当 OpenMP 编程使用 12 线程时，加速效果达到最佳。

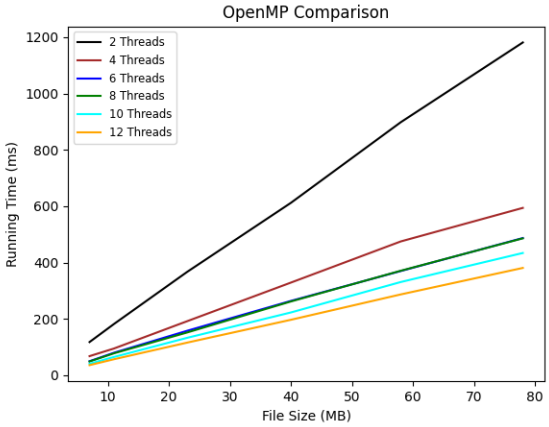

Figure 12: OpenMP 不同线程数的性能对比

接下来比较使用 CUDA 的 GPU 并行计算，设置不同的线程数，观察线程数对运行时间的影响。可以发现，当 GPU 上设置 1024 个线程时，加速效果达到最佳。

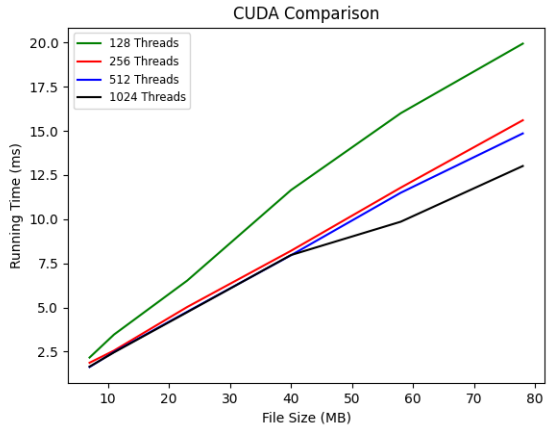

Figure 13: CUDA 不同线程数的性能对比

最后将 OpenMP 和 CUDA 的最佳运行结果与串行程序的运行结果进行对比，结果如下图。

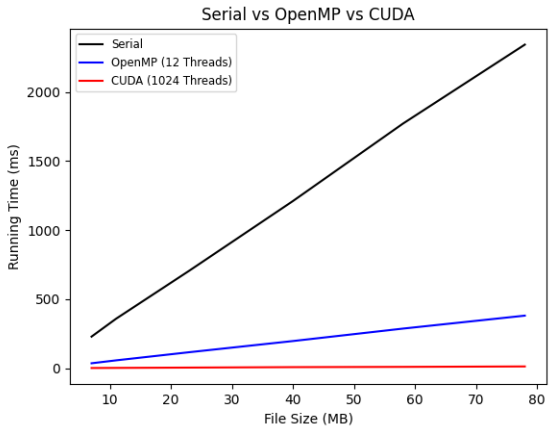

Figure 14: 串行、OpenMP、CUDA 的性能对比

比较三组数据可以发现，当明文文件规模不断增大时，串行算法的的运行时间随着问题规模的增大线性增长；OpenMP 并行化后的运行时间也随问题规模增大而增加，但相比于串行算法加速效果越来越明显；而使用 CUDA 并行化后的运行时间随着问题规模的增大变化不大，这是因为相对于使用 1024 线程的 GPU 并行计算，测试的明文文件还不够大，所以运行时间看不出有什么变化。这也从另一方面看出了 GPU 并行计算的强大，其在当前问题规模的情况下远远未达到饱和，当问题规模继续不断增大时，GPU 并行计算的加速效果会越来越出色。

当明文文件为 80 MB 时，串行程序的运行时间是 2344s，OpenMP 编程的 CPU 并行运行时间是 381 秒，加速比达到 6.15；CUDA 编程的 GPU 并行运行时间是 13s，加速比达到 180。虽然 OpenMP 和 CUDA 编程对 AES 的串行程序都有加速效果，但显然使用 GPU 的并行计算加速效果要出色许多。随着问题规模的扩大，GPU 并行计算可以大幅度减少加密解密的运行时间，并且显著提升 AES 算法的性能。

7 总结

本文首先讲述了 AES 算法的原理，然后分别实现了 AES 算法的 CPU 串行版本、OpenMP 编程的 CPU 并行版本以及 CUDA 编程的 GPU 并行版本。通过实验发现，不论是 OpenMP 还是 CUDA，其对 AES 算法的实现都有不错的加速效果，但 CUDA 编程的 GPU 并行效果更好。随着问题规模的增大，其加速效果可以得到显著的提升。

本学期的《信息安全技术》课程即将告一段落，在这门课中我学到了密码学理论、各种加密解密算法、网络安全等一系列相关的信息安全知识，收获颇丰。希望在未来的日子我能有机会继续探索这一领域丰厚的的知识宝库。

参考文献

[1] AES 加密算法的详细介绍与实现.https://blog.csdn.net/qq_28205153/article/details/55798628, 2017.

[2] 费雄伟, 李肯立, 阳王东. 基于 CUDA 的 AES 并行算法优化 [J]. 计算机工程,2014,40(09):6-12.

[3] 费雄伟, 李肯立, 阳王东, 杜家宜. 基于 CUDA 的并行 AES 算法的实现和加速效率探索 [J]. 计算机科学,2015,42(01):59-62+74.

[4] Hesham Alhumyani. AES Overhead Mitigation Using OpenMP[J]. Journal of Computer and Communications, 2019, 7(7) : 206-218.