

基于 GPU 的 AES 并行算法实现

19335074 黄玟瑜

19 级计算机科学与技术 (超级计算方向)

【摘要】 密码学中的高级加密标准 (Advanced Encryption Standard, AES), 是美国联邦政府采用的一种区块加密标准, 此标准用来替代原先的 DES (Data Encryption Standard), 已经被多方分析且广为全世界所使用。AES 算法中一般的加密通常都是块加密, 独立的块加密不能隐藏明文的模式, 为解决此问题需要采用链加密模式, 本文对对称加密和分组加密中的 ECB、CBC、CTR 模式进行分析并实现。在大数据时代背景下, 很多应用服务器面临着执行大量计算稠密的加密挑战, 笔者结合专业所学的高性能计算领域的知识, 对 AES 的加解密模式进行可并行性分析, 使用 GPU 异构编程并行化 ECB、CTR 模式的加解密程序, 实现了高性能的 AES 加解密算法。经性能分析, 并行程序相对于普通串行程序有着极高的计算性能提升, 同时利用线程计数器避免了一般并行算法使明文结构暴露的问题。

【关键词】 AES, 高级加密标准, 链加密模式, 并行计算, GPU, CUDA, 异构编程

1 引言

分组密码是目前应用较为广泛的一种密码体制。DES 加密算法开创了公开密码算法的先例, 使了解算法但不具有正确密钥的密文持有者不能通过算法找出明文数据, 其安全性来源于破解密码计算上和时间上的困难性。然而随着计算机及网络技术的发展, 计算能力已经对像 DES 这样的加密算法构成了威胁。1997 年, 美国国家标准和技术研究所 (NIST) 发起了征集新一代数据加密标准即高级数据加密标准 AES 算法的活动, 2000 年 NIST 宣布来自比利时的 Rijndael 算法被选为 AES 的最终算法。

许多网络应用尤其是电子商务和网络银行, 需要提供数据加密以保证通信安全。由于加密是一个计算稠密性问题, 当用户访问量很大时, 给服务器带来了沉重的负担; 而且随着网络应用的不断发展和普及, 需要进行加密以提供安全的应用越来越多。很多应用服务器一方面承受着巨大的加/解密压力, 另一方面通过升级硬件得到性能的提升也很有限。因此能够利用现有的多核硬件, 以低成本的方式提升加/解密性能, 有很大的现实意义。

义。

GPU (Graphics Processing Unit, 图形处理器), 是最近发展起来的具有并行计算性能的单片多核处理器。设计之初, 它是用做图像和图形相关运算工作的微处理器, 采用单指令多线程 (Single Instruction Multiple Threads) 的体系结构, 可实现密集的并行计算, 满足 AES 高强度的加解密需求。NVIDIA 公司提出了统一计算设备架构 CUDA (Compute Unified Device Architecture), 是一种通用并行计算平台和编程模型, 为编程人员提供了使用 GPU 进行异构编程的通用编程接口, 降低了 GPU 的使用门槛, 极大地方便了计算机从业者使用 GPU 来实现高性能的计算。

2 AES 算法实现

AES 是一个对称分组密码, 以 128 位为一个分组, 可以使用 128 位、192 位或 256 位密钥 (对应 AES 标准分别为 AES-128、AES-192、AES-256) 进行分组加密。AES 是一个完全对称的结构, 加密过程使用异或、查找表和移位操作, 能够通过编程在软件层面上实现。

2.1 算法流程

每一个 128 位的明文分组被设计为执行多轮迭代转换得到同样长的密文分组。轮数由密钥位数决定：128 位密钥迭代 10 轮、192 位密钥迭代 12 轮、256 位密钥迭代 14 轮。除了首轮有一次轮密钥加和末轮少一次列混淆外，其余每轮都是一样的处理方式。

每轮加密的对象为 16 字节（即 128 位）的分组，可以看做是一个 4×4 的二维数组，又称作状态矩阵 (state)。每轮的加密可分为 4 个阶段：轮密钥加 (AddRoundKeys, 对轮密钥和状态矩阵进行异或运算，达到盲化效果)、字节替换 (SubBytes, 使用 S-Box 进行非线性替换)、行移位 (ShiftRows, 以行号为偏移量对每行分别进行循环移位，达到字内混淆)、列混淆 (MixColumns, 线性变换每列以混淆数据)^[1]。

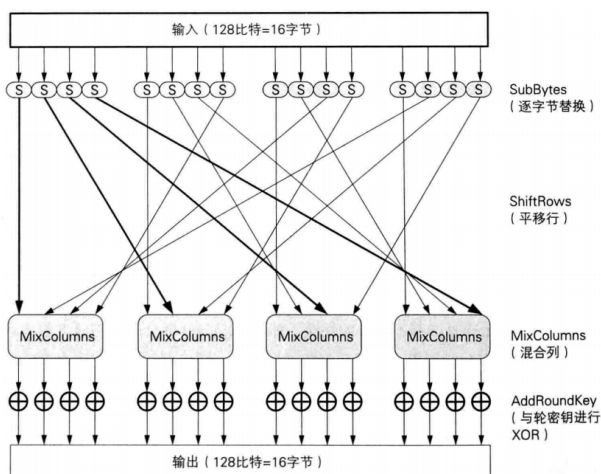


图 1 AES 轮加密流程

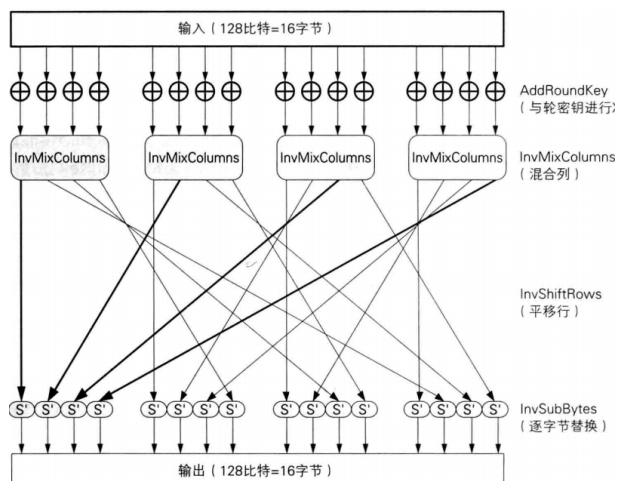


图 2 AES 轮解密流程

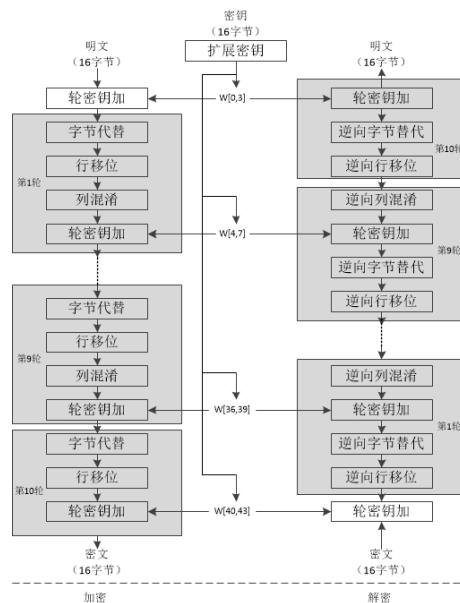


图 3 AES-128 加解密流程图

2.1.1 电码本模式 (ECB)

ECB (Electronic Code Book Mode, 电子密码本) 模式是将整个明文分成若干长度相同的明文分组，然后对每一个明文分组进行加密。

ECB 模式的加解密过程如下所示：

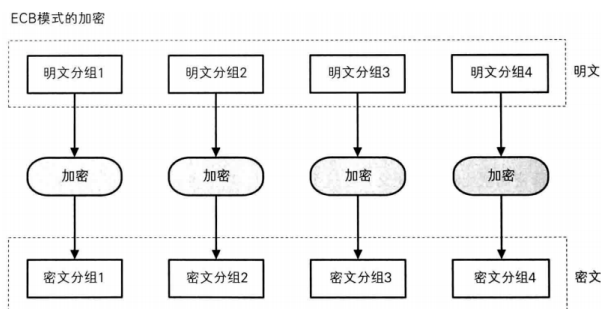


图 4 ECB 模式加密

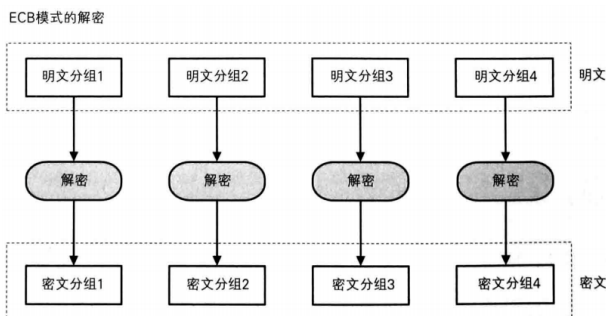


图 5 ECB 模式解密

ECB 的优点在于加密流程简单、误差不会在分组间传递，但它不能隐藏明文模式，尤其是对于

图像数据，加密后易于分辨图像轮廓。此外，它还存在受到主动攻击的风险，若修改明文，后续内容不受影响，只要误差传递该缺点就存在。

2.1.2 密码分组链模式 (CBC)

CBC (Cipher Block Chaining, 密码分组链接模式) 将整个明文分成若干长度相同的明文分组，在每一个明文分组加密时先将其与初始化向量或者上一个明文分组加密后的密文分组进行异或运算，再与密钥进行加密。

CBC模式的加密

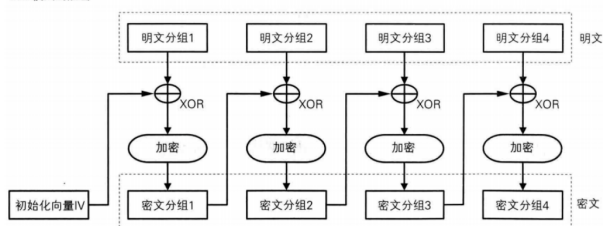


图 6 CBC 模式加密

CBC模式的解密

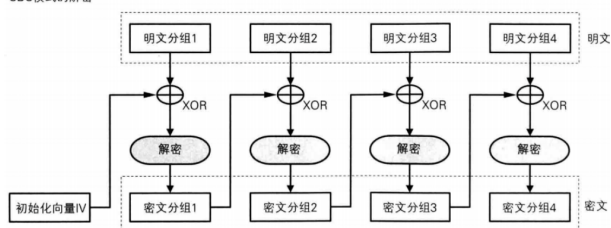


图 7 CBC 模式解密

CBC 模式的优点在于能掩盖明文结构信息，使得保证相同密文可得不同明文，从而不容易受到主动攻击。从安全性上说，CBC 模式好于 ECB，更适合传输长度长的报文，也是 SSL 和 IPsec 的标准。

但 CBC 模式也存在缺点，链式加密会导致误差传递，若一个分组出错会导致后续全部出错，而且第一个明文块需要与一个初始化向量进行异或，初始化向量的选取也比较复杂。

初始化向量 IV 的选取方式有：固定 IV、计数器 IV 和随机 IV，其中随机 IV (用的最多) 只能得到伪随机数，瞬时 IV 难以得到瞬时值。

2.1.3 计数器模式 (CTR)

CTR (Counter, 计数器模式) 是完全的流模式，将瞬时值与计数器的值相加，然后对此进行加密产生密钥流的一个密钥分组，将该密钥分组和

明文分组进行异或操作，得到密文分组。

CTR模式的加密

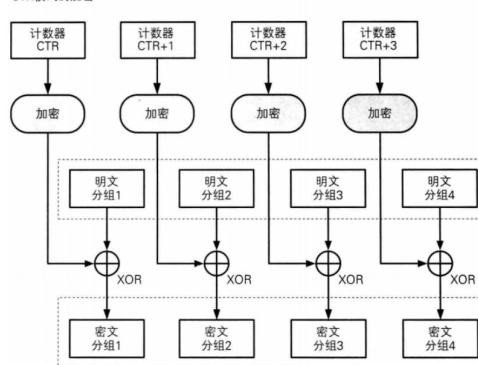


图 8 CTR 模式加密

CTR模式的解密

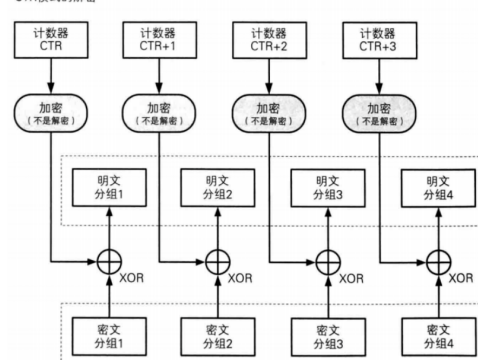


图 9 CTR 模式解密

CTR 的优点是不泄露明文结构，并且利用了异或运算的逻辑原理，即一个位对另外一个位异或两次的结果是它本身，使得加解密过程是相同的，仅需实现加密函数。

CTR 模式需要瞬时值的初始化向量 IV，难以保证初始化向量 IV 的唯一性。

2.2 编程实现

根据前面所述算法流程，可以把代码实现分成 2 个部分，第一部分是单个分组的加、解密函数的实现，第二部分实现不同模式的加、解密，即对一个给定长度的明文，将其拆分成多个明文分组，调用第一部分实现的加、解密函数实现不同模式的加密。

2.2.1 分组加解密函数

首先实现对单个分组的加、解密函数。

对加、解密的 4 个状态变换操作进行封装，以便于在加、解密的主函数 Cipher、InvCipher 中调用，以下是加、解密过程所涉及的函数声明：

```
// 为状态矩阵加上轮密钥，使用异或操作
void AddRoundKey(uint8_t round, state_t* state, const uint8_t* RoundKey);
// 对状态矩阵进行S盒替换
void SubBytes(state_t* state, const uint8_t *sbox);
// 行移位
void ShiftRows(state_t* state);
// 列混合
void MixColumns(state_t* state);
// 加密过程的主函数
void Cipher(const uint8_t *sbox, state_t* state, const uint8_t* RoundKey);
// 对状态矩阵进行逆S盒替换
void InvSubBytes(state_t* state, const uint8_t *rsbox);
// 逆行移位
void InvShiftRows(state_t* state);
// 逆列混合
void InvMixColumns(state_t* state);
// 解密过程的主函数
void InvCipher(const uint8_t *rsbox, state_t* state, const uint8_t* RoundKey);
```

由于篇幅问题，不展示状态矩阵变换操作函数具体的实现代码，可以在源代码目录下找到对应的源文件。

加密过程如下所示，其中 Nr 为加密轮数，根据所使用的 AES 标准而定：

```
void Cipher(const uint8_t *sbox, state_t* state, const uint8_t* RoundKey)
{
    uint8_t round = 0;

    // 第一轮加密之前将初始密钥添加到状态矩阵
    AddRoundKey(0, state, RoundKey);
    // 进行Nr轮加密，前Nr-1轮的加密过程是相同的，最后一轮加密不需要进行列混合
    for (round = 1; ; ++round)
    {
        SubBytes(state, sbox);
        ShiftRows(state);
        if (round == Nr) {
            break;
        }
        MixColumns(state);
        AddRoundKey(round, state, RoundKey);
    }
    // 最后一轮的轮密钥加
    AddRoundKey(Nr, state, RoundKey);
}
```

解密过程如下所示：

```
void InvCipher(const uint8_t *rsbox, state_t* state, const uint8_t* RoundKey)
{
    uint8_t round = 0;

    // 第一轮解密之前将初始密钥添加到状态矩阵
    AddRoundKey(Nr, state, RoundKey);

    // 进行Nr轮解密，前Nr-1轮的解密过程是相同的，最后一轮解密不需要进行逆列混合
    for (round = (Nr - 1); ; --round)
    {
        InvShiftRows(state);
        InvSubBytes(state, rsbox);
        AddRoundKey(round, state, RoundKey);
        if (round == 0) {
            break;
        }
        InvMixColumns(state);
    }
}
```

此外，还需定义密钥结构体 AES_ctx，用于保存轮密钥和初始化向量，并定义实现密钥拓展的函数 KeyExpansion，如下所示：

```
struct AES_ctx
{
    uint8_t RoundKey[AES_keyExpSize];
```

```
// 在ECB模式下不会使用到初始向量
#if (defined(CBC) && (CBC == 1)) || (defined(CTR) && (CTR == 1))
    uint8_t Iv[AES_BLOCKLEN];
#endif
};

void KeyExpansion(const uint8_t* sbox, uint8_t* RoundKey, const uint8_t* Key);
```

2.2.2 不同模式的加解密函数

下面实现 ECB、CBC、CTR 模式的加、解密函数。

同算法流程中所述的 ECB 模式，它对每一个明文分组分别进行加密，如下所示：

```
void AES_ECB_encrypt_buffer(const uint8_t* sbox, const struct AES_ctx* ctx, uint8_t* buf, size_t length)
{
    // 使用AES算法对buf进行加密
    size_t i;
    for (i = 0; i < length; i += AES_BLOCKLEN)
    {
        Cipher(sbox, (state_t*)buf, ctx->RoundKey);
        buf += AES_BLOCKLEN;
    }
}
```

输入参数 buf 为所需要加密的明文，length 指明明文长度。

以下是 CBC、CTR 模式的加密过程：

```
void AES_CBC_encrypt_buffer(const uint8_t* sbox, struct AES_ctx* ctx, uint8_t* buf, size_t length)
{
    size_t i;
    uint8_t *Iv = ctx->Iv;
    for (i = 0; i < length; i += AES_BLOCKLEN)
    {
        XorWithIv(buf, Iv);
        Cipher(sbox, (state_t*)buf, ctx->RoundKey);
        Iv = buf;
        buf += AES_BLOCKLEN;
    }
    // 存储当前加密后的块作为下一个块加密的Iv
    memcpy(ctx->Iv, Iv, AES_BLOCKLEN);
}

void AES_CTR_xcrypt_buffer(const uint8_t* sbox, struct AES_ctx* ctx, uint8_t* buf, size_t length)
{
    uint8_t buffer[AES_BLOCKLEN];

    size_t i;
    int bi;
    for (i = 0, bi = AES_BLOCKLEN; i < length; ++i, ++bi)
    {
        if (bi == AES_BLOCKLEN) // 生成新的组
        {
            // 对计数器进行加密
            memcpy(buffer, ctx->Iv, AES_BLOCKLEN);
            Cipher(sbox, (state_t*)buffer, ctx->RoundKey);

            // 增加 Iv 并处理溢出
            for (bi = (AES_BLOCKLEN - 1); bi >= 0; --bi)
            {
                if (ctx->Iv[bi] == 255)
                {
                    ctx->Iv[bi] = 0;
                    continue;
                }
                ctx->Iv[bi] += 1;
                break;
            }
            bi = 0;
        }
        buf[i] = (buf[i] ^ buffer[bi]); // 将计数器加密结果与明文组进行异或
    }
}
```

在解密时，ECB 模式对每个密文分组分别进行解密，CBC 模式从最后一个密文分组链式向前

解密，对于 CTR 模式，解密过程和加密过程是相同的。

3 AES 算法并行化

在并行化一个程序时，首先需要对程序进行可并行性分析。在这个过程中，通过分析程序确定其中必须串行执行的任务以及可以并行执行的部分，从而保证并行化过程中逻辑的正确性。

对于其中可并行执行的部分，根据设计并行计算程序的 Foster 方法，设计方法中的四步为划分、通信、聚集和映射，实现并行 AES 算法程序的设计^[2]。

3.1 可并行性分析

2001 年 12 月 NIST 公布了 AES 的 5 种工作模式：电子密码本 (Electronic Code Book, ECB)、密码分组链接 (Cipher Block Chaining, CBC)、密码反馈 (Cipher Feedback, CFB) 和输出反馈 (Output Feedback, OFB) 和 CTR (计数器模式, Counter Mode)。

ECB 为最简单的加密模式，对每个分组一一加密有利于并行计算，但存在包括暴露明文轮廓的风险。其他工作模式可以较好的隐藏明文轮廓，其中 CBC、CFB、OFB 模式都采用了链加密的方式来避免此问题，它们依赖于相邻分组的加密结果来进行加密，因此难以并行化。CTR 模式结合了初始化向量 IV 和瞬时值，线程/进程在计算时可以根据其标识计算瞬时值，实现并行加、解密，同时不泄露明文轮廓。

各种工作模式的可并行性如下表所示^[3]：

表 1 AES 五种加密模式的可并行性	
Block Cipher Mode	Parallel Capacity of Encryption
ECB	Suitable
CBC	Not suitable
CFB	Not suitable
OFB	Not suitable
CTR	Suitable

通过分析可知 ECB、CTR 工作模式可以并行执行，下面考虑将这 2 种工作模式并行化。

3.2 并行程序设计

根据 Foster 方法，首先对任务进行划分，每个线程完成对一个块的加解密。考虑到可能出现分组数远大于线程数的情况（实际上，GPU 上的线程数量已经十分庞大了，一般情况达到 10^5 以上），使每个线程加密 stride 个分组，stride 可理解为并行加密的步长，每一步调用一个线程。

接下来是通信部分，每个线程进行独立的加密，故不需要和其他线程进行通信，但需要共享的数据 S-Box 和轮密钥，它们是只读的，此外线程间还共享整个明文/密文段，每个线程只对明文/密文段中自己所负责的部分进行读写，因此不会导致竞争等问题。

再考虑可聚集的部分，由于生成轮密钥的任务量很小，生成了轮密钥后可供所有线程共同使用，故此部分串行执行即可。

最后是指派和部署。CUDA 线程架构是一个两层的线程层次结构，由线程块和线程块网格组成，如下图所示：

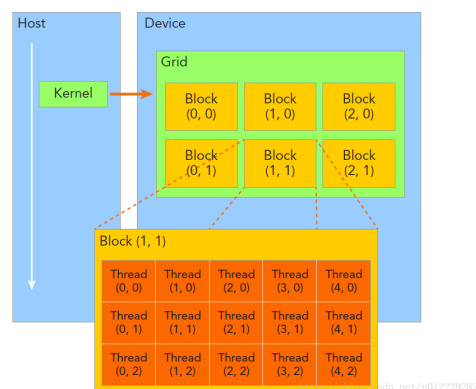


图 10 CUDA 线程层次架构

在 CUDA 架构里，一个线程内所有线程都执行一次的函数成为 Kernel，或者说，由一个 Kernel 启动所产生的所有线程统称为一个线程网格 (Grid)。同一线程网格中的所有线程共享同全局内存空间。

一个网格有多个线程块 (Block) 构成，一个线程块包含一组线程，同一线程块内的线程协同可以通过“同步”和“共享内存”的方式来实现。

Grid 和 Block 都可以使用多维的线程组织形式，比如，在矩阵计算中往往使用二维的线程组织形式。由于明文/密文段是一维的，在这里采用一维的线程组织形式，每个 Grid 负责明文/密文段

中的一大段，每个 Block 负责所在 Grid 对应的子明文/子密文段中的一小段，每个 Thread 负责所在 Block 对应小段中的 Stride 个明文/密文分组。

综上，并程序的加密流程可以描述如下：主机端接收明文，根据初始密钥生成轮密钥，随后将轮密钥、明文以及 S-Box 传送到 GPU 上的共享内存中（CTR 模式还需要传送初始化向量 IV），在 GPU 上，大量并发的线程对所负责明文段进行加密，加密完毕后 GPU 通知主机端，主机端将加密后的密文拷贝回主机内存中。

ECB 模式并行加密算法如下：

```
void AES_ECB_encrypt_buffer_parallel(const uint8_t* sbox, const
    struct AES_ctx* ctx, uint8_t* buf, size_t length)
{
    cudaDeviceProp prop; //cudaDeviceProp 的一个对象
    cudaGetDeviceProperties(&prop, 0); //第二参数为那个gpu

    uint8_t *d_buf, *d_sbox;
    struct AES_ctx* d_ctx;
    // allocate device memory
    cudaMalloc((void**)&d_buf, sizeof(uint8_t) * length);
    cudaMalloc((void**)&d_ctx, sizeof(struct AES_ctx));
    cudaMalloc((void**)&d_sbox, sizeof(uint8_t) * 256);
    // host send data: host => device
    cudaMemcpy(d_buf, buf, sizeof(uint8_t) * length,
        cudaMemcpyHostToDevice);
    cudaMemcpy(d_ctx, ctx, sizeof(struct AES_ctx),
        cudaMemcpyHostToDevice);
    cudaMemcpy(d_sbox, sbox, sizeof(uint8_t) * 256,
        cudaMemcpyHostToDevice);
    // compute
    size_t stride = 4;
    size_t AES_num_block = length / AES_BLOCKLEN;
    size_t threadPerBlock = min((AES_num_block + stride - 1) /
        stride, (size_t)prop.maxThreadsPerBlock);
    size_t blockNumber = (((AES_num_block + stride - 1) / stride)
        + threadPerBlock - 1) / threadPerBlock;
    Cipher_Kernel_ECB<<<blockNumber, threadPerBlock>>>> (d_sbox,
        d_ctx, d_buf, AES_num_block, stride);
    // host receive data: device => host
    cudaMemcpy(buf, d_buf, sizeof(uint8_t) * length,
        cudaMemcpyDeviceToHost);
    // release device memory
    cudaFree(d_buf);
    cudaFree(d_ctx);
    cudaFree(d_sbox);
}
```

ECB 加密模式的 Kernel 函数如下：

```
__global__ void Cipher_Kernel_ECB(const uint8_t* sbox, const
    struct AES_ctx* ctx, uint8_t* buf, size_t AES_num_block,
    size_t stride)
{
    int x = threadIdx.x + (blockDim.x * blockIdx.x);
    for(size_t i = x * stride; i < (x + 1) * stride && i <
        AES_num_block; i++){
        Cipher(sbox, (state_t*)buf + i, ctx->RoundKey);
    }
}
```

在 CUDA 编程中，由主机端调用，GPU 执行的函数需加上 `__global__` 关键字。此外，前面实现的单个分组的加解密函数默认只能由主机调用，若要使 GPU 也能调用，需加上 `__host__ __device__`，其中 `__host__` 声明为主机端可调用函数，`__device__` 声明为设备端可调用函数，这两个关键字一起使用表明该函数在主机端和设备端都可以使用^[4]。

相应的，CTR 工作模式的 Kernel 函数如下所

示：

```
__global__ void Cipher_Kernel_CTR(const uint8_t* sbox, const
    struct AES_ctx* ctx, uint8_t* buf, size_t AES_num_block,
    size_t stride)
{
    size_t x = threadIdx.x + (blockDim.x * blockIdx.x);
    for(size_t index = x * stride; index < (x + 1) * stride &&
        index < AES_num_block; index++){
        size_t i, remain;
        uint8_t buffer[AES_BLOCKLEN];
        memcpy(buffer, ctx->Iv, AES_BLOCKLEN);
        // counter = Iv + index
        for(remain = index, i = (AES_BLOCKLEN - 1); remain > 0 && i
            >= 0; remain /= 256, i--){
            if((short)buffer[i] + (short)(remain % 256) > 255 && i >
                0) // 处理进位
                buffer[i - 1]++;
            buffer[i] += remain % 256;
        }
        Cipher(sbox, (state_t*)buffer, ctx->RoundKey);

        for(i = 0; i < AES_BLOCKLEN; i++)
            buf[(index * AES_BLOCKLEN) + i] = (buf[(index *
                AES_BLOCKLEN) + i] ^ buffer[i]);
    }
}
```

在主机端 CTR 模式加密过程中，每个明文分组使用 IV 加密完毕后使 IV 递增 1，以供下一个明文分组的加密使用，在 CTR 模式的 Kernel 函数中，CUDA 线程根据当前分组在所有分组中的序号，加上加密过程初始时刻的 IV，构成当前分组加密所需的 IV，从而进行分组加密。

最后完成了并行化的 ECB、CTR 工作模式的加解密函数，如下：

```
// ECB模式并行加解密
void AES_ECB_encrypt_buffer_parallel(const uint8_t* sbox, const
    struct AES_ctx* ctx, uint8_t* buf, size_t length):
void AES_ECB_decrypt_buffer_parallel(const uint8_t* rsbox, const
    struct AES_ctx* ctx, uint8_t* buf, size_t length)
// CTR模式并行加解密
void AES_CTR_encrypt_buffer_parallel(const uint8_t* sbox, struct
    AES_ctx* ctx, uint8_t* buf, size_t length):
```

4 结果和性能分析

首先对串行版本 AES 算法的 ECB、CBC、CTR 模式做常规正确性检验，调用上述实现函数对 64 字节（4 个分组）的明文/密文进行加解密，对比程序运行结果和标准结果，若完全一致则说明实现的加/解密程序通过正确性检验。

随后通过加/解密不同大小的文件，记录串行程序和并行程序所需的计算时间，分析运行时间和加速比来比较串行程序和并行程序的性能差异。

4.1 实验环境

1. 硬件设备

- Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz
- NVIDIA Corporation GP108M [GeForce MX250]

2. 软件平台

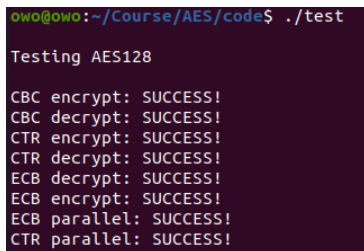
- Ubuntu 20.04 LTS
- gcc version 9.3.0
- Cuda compilation tools, release 10.1, V10.1.243
- CUDA Version 11.4

4.2 正确性检验

编写以下函数实现对 AES 工作模式的测试：

```
int test_encrypt_ecb(void); // 测试ECB加密函数的正确性
int test_decrypt_ecb(void); // 测试ECB解密函数的正确性
int test_encrypt_cbc(void); // 测试CBC加密函数的正确性
int test_decrypt_cbc(void); // 测试CBC解密函数的正确性
int test_encrypt_ctr(void); // 测试CTR加密函数的正确性
int test_decrypt_ctr(void); // 测试CTR解密函数的正确性
// 测试并行ECB加密解密函数的正确性（只有加解密都正确才通过）
int test_parallel_ecb(void);
// 测试并行CTR加密解密函数的正确性（只有加解密都正确才通过）
int test_parallel_ctr(void);
```

以上函数的具体实现见源代码，对 64 字节（4 个分组）的明文/密文进行加解密，逐字节的比较程序运行结果和标准结果，以此来检验结果的正确性。



```
owo@owo:~/Course/AES/code$ ./test
Testing AES128
CBC encrypt: SUCCESS!
CBC decrypt: SUCCESS!
CTR encrypt: SUCCESS!
CTR decrypt: SUCCESS!
ECB decrypt: SUCCESS!
ECB encrypt: SUCCESS!
ECB parallel: SUCCESS!
CTR parallel: SUCCESS!
```

图 11 正确性检验结果

由运行结果可知，所实现的 AES 加解密函数通过了正确性检验。

4.3 性能分析

实验结果取同样明文规模的 AES 并行加密算法和串行加密算法的实际运行时间。对较小的明文运行 3 次加解密算法，取运行时间的中值，以减少数据的误差。并行算法执行时间包括了初始生成轮密钥、主机端和 GPU 通信的时间。具体实验数据如表 2、3 所列。

其中加速比的计算公式为

$$Speedup = \frac{T_{serial}}{T_{parallel}}$$

由实验运行结果数据可知，当明文规模较小时，并程序的计算时间比串程序要长，几乎没

表 2 AES 加密算法并行和串行程序性能：ECB 模式（时间：ms）

明文大小	串行执行时间	并行执行总时间	加速比
256B	0.129	0.607	0.213
1KB	0.477	0.599	0.796
4KB	1.969	0.904	2.178
16KB	7.511	1.205	6.233
64KB	30.605	2.138	14.315
256KB	119.541	5.170	23.122
1MB	473.321	17.555	26.962
4MB	1886.881	65.702	28.719
16MB	7550.928	235.820	32.020
64MB	31022.021	922.387	33.632
256MB	125147.127	3672.751	34.074

表 3 AES 加密算法并行和串行程序性能：CTR 模式（时间：ms）

明文大小	串行执行时间	并行执行总时间	加速比
256B	0.048	0.596	0.081
1KB	0.192	0.606	0.317
4KB	0.806	0.715	1.127
16KB	7.854	0.942	8.338
64KB	30.423	0.946	32.160
256KB	119.176	1.893	62.956
1MB	472.840	5.380	87.888
4MB	1891.972	18.559	101.944
16MB	7931.621	72.981	108.681
64MB	31341.198	262.672	119.317
256MB	125495.955	1003.418	125.068

有加速效果，因为主机端和 GPU 之间通信、线程调度等会消耗格外的时间，当这部分开销大于多线程执行带来的收益时，会导致拖慢时间的情况出现。但这部分任务一般是低拓展性的，随着明文规模增大，开销不会明显增加，此时并程序的优势展现出来，多线程计算显著降低了计算时间上的开销，使得计算收益远大于线程调度上浪费，体现了远大于串程序的计算性能。这一特性从图 12、13 中 CPU 和 GPU 运行时间的对比中直观地体现出来。

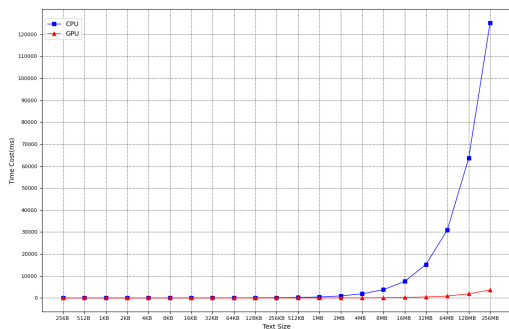


图 12 CPU 和 GPU 运行时间对比图：ECB 模式

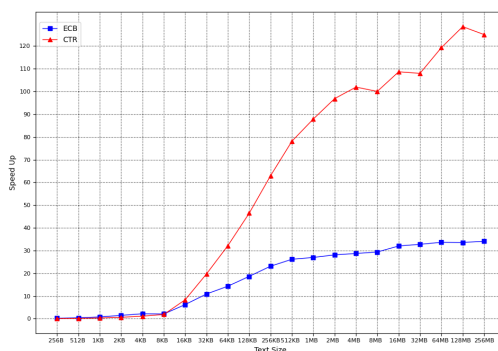


图 14 ECB 模式和 CTR 模式并行算法加速比

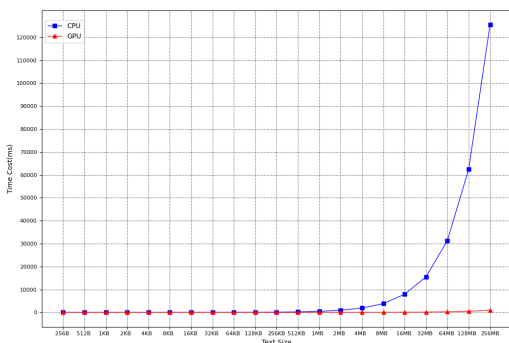


图 13 CPU 和 GPU 运行时间对比图：CTR 模式

加速比的增长趋势如图 14 所示。明文大小从 256B ~ 256MB，综合来看，整个并行 AES 算法性能模型包含 3 个阶段，分别是指数增长、线性增长和稳定阶段。刚到达稳定阶段时的明文大小是一个重要的参考指标，这是充分利用所有计算核心时所需要的最小明文长度。

此外，对比 ECB 模式和 CTR 模式的运行时加速比，CTR 模式的并行程序的加速比高于 CBC 模式的并行程序。据分析，在测试时两种工作模式都进行了一次加密和一次解密，由于 CTR 模式的加密函数和解密函数是相同的，它只有一个 Kernel 函数，也就是说，GPU 执行了 2 次相同的 Kernel 函数，而 ECB 模式需要执行不同的 Kernel 函数，基于局部性原理，CBC 模式比 ECB 模式有更高的加解密速度。

5 结 语

在本次课程项目实践中，笔者首先实现了 ECB 工作模式下 AES 加密与解密算法，考虑到单分组加密暴露明文结构的问题，随后实现了链式加密的 CBC 加解密算法以及基于瞬时值的 CTR 模式加解密算法。随着网络应用的不断发展和普及，需要进行加密以提供安全的应用越来越多，需要加密的数据文件越来越大，需要高性能的加解密程序来满足计算需求，笔者结合并行计算领域知识，对 AES 加解密算法进行可并行性分析，实现了基于 GPU 的 AES 并行算法。经分析，并行算法有着极高的运算效率，在实际应用中部署到集群上运行会得到更高的加速比和计算性能，其中 CTR 模式在并行计算下有更高的加速比，并且仍能隐藏明文轮廓，和普通并行算法相比有更高的安全性。

通过此次实践，笔者对 AES 加解密算法有了进一步的理解，并了解了许多提高加密安全性的算法。同时在思考如何提升算法性能的过程中，锻炼了自己思考和解决问题的能力，受益匪浅。

参考文献

- [1] Mingxing He, Hao Lin. Implementation of the advanced encryption standard[J]. Application Research of Computers, 2002(03): 61-63.
- [2] 费雄伟, 李肯立, 阳王东, 等. 基于 CUDA 的并行 AES 算法的实现和加速效探索[J]. Computer Science, 2015, 42(1): 59-74.
- [3] Faculty of Computer Science and Information Systems. Parallelization of the aes algorithm[J]. Communications and Computers, 2005 (11): 224-228.
- [4] Jason sanders, Edward Kandrot. An introduction to genera-purpose gpu programming[J]. 2011.