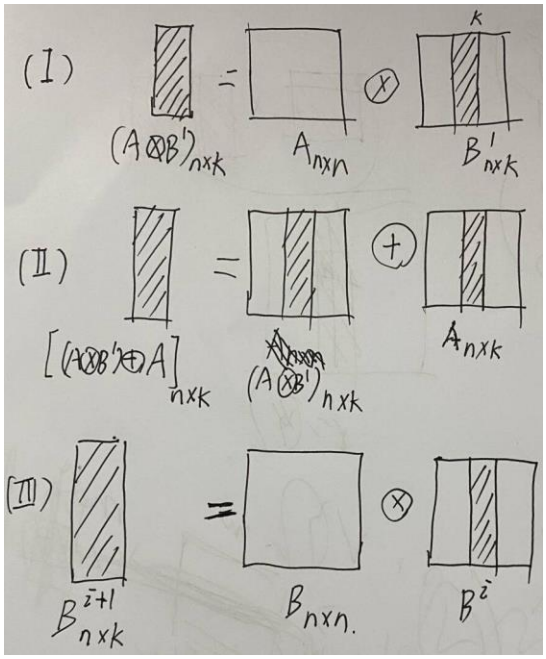
实验思路

并行方式

公式 C 由很多项模 2 矩阵加法组成，在每一项中，B 的计算都依赖于前一项，因此不能根据加法项来划分进程。如果将 A 分块划分，每个进程都需要计算完整的 B。所以，应对 B 进行划分，由于 B 在乘法右侧，因此对 B 按列进行划分。

根据定义，又因为矩阵乘法满足交换律，所以 $B^{i+1} = B^i \times B = B \times B^i$，因此 B 的迭代可以使用列分块来进行。计算 C 的过程中，一共涉及到 3 种运算，如下图所示，每种运算都可以对运算符右侧进行列分块，因此我们可以对列进行划分并使用 MPI 来加速。



每个进程执行的每次运算都是模 2 的矩阵加法或乘法，因此可以使用 GPU 进行并行处理。程序读入的是一维参数，因此可以对运算符左侧矩阵的行进行划分。

MPI 并行

程序参数的读取以及矩阵数据的获取已经由原始代码完成。对于代码实现部分，首先让其他进程等待主进程读取数据，然后广播矩阵大小、迭代次数、每个

grid 的 block 数和每个 block 的 thread 数。vector 无法直接使用 MPI 来发送，因此将其转化为普通数组。另外，主进程需要分配结果数组的空间，并且其他进程也需要等待此部分的数据处理结束。

```
// ============================================================
// ====    Write your implementation below this line    ====
// ============================================================
// ============================================================
MPI_Barrier(MPI_COMM_WORLD);
// 参数在主进程中读取，其他进程需要被广播才有参数值
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&number_of_block_in_a_grid, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&number_of_thread_in_a_block, 1, MPI_INT, 0, MPI_COMM_WORLD);
int *Aarray = (int *)malloc(n * n * sizeof(int));
int *Barray = (int *)malloc(n * n * sizeof(int));
int *Carray;
if (rank == 0)
{
    // vector转array
    int *AarrayTemp = Aarray, *BarrayTemp = Barray;
    for (int i = 0; i < n; i++)
    {
        memcpy(AarrayTemp, &A[i][0], n * sizeof(int));
        memcpy(BarrayTemp, &B[i][0], n * sizeof(int));
        AarrayTemp += n;
        BarrayTemp += n;
    }
    // 分配结果数组的空间
    Carray = (int *)malloc(n * n * sizeof(int));
}
MPI_Barrier(MPI_COMM_WORLD);
```

接下来，每个进程广播在计算的时候都需要用到矩阵 A 和 B，计算在划分之后各个进程的列分块区间。当矩阵的列长度小于进程总数时，设置区间大小为 1。进程数不一定总能整除矩阵大小，因此最后一个进程的终止列需要特殊处理，最后一个进程的处理区间可能比其他进程要大一些。然后，每个进程都调用计算函数，并等待其他进程计算完成。

```
MPI_Bcast(Aarray, n * n, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(Barray, n * n, MPI_INT, 0, MPI_COMM_WORLD);
// 计算各个进程计算的起始终止列
int scatterSize = max(n / number_of_processes, 1);
int startCol = rank * scatterSize;
int endCol = rank == number_of_processes - 1 ? n - 1 : startCol + scatterSize - 1;
int realSize = endCol - startCol + 1; // 只有最后一列可能不一样
int *cPart = (int *)malloc(realSize * n * sizeof(int));
ParallelCalculationWithCuda(Aarray, Barray, cPart, n, m,
                            startCol, endCol,
                            number_of_block_in_a_grid, number_of_thread_in_a_block, rank);
MPI_Barrier(MPI_COMM_WORLD);
```

MPI_Gather 将每个进程的数据进行聚集，但是我们最终得到的数据是按列存储的，直接使用这个函数会出现问题。这里调用 MPI_Gather 函数 n 次，每次聚集同一行不同进程的计算结果。最后一个进程计算的列长度可能比其他进程多，因此这里必须访问 cPart[realSize][*]而不能是 cPart[scatterSize][*]。在聚集的时候，我们使用的是固定大小，因此需要将最后一个进程多出来的数据发送回主进程。这里对最后一个进程和主进程进行特殊判断，使用 MPI_Send 和 MPI_Recv 发送和接收数据。

```
// Gather是按行存，所以这里按列每次取一层
for (int i = 0; i < n; i++)
{
    MPI_Gather(cPart + i * realSize, scatterSize, MPI_INT,
               Carray + i * n, scatterSize, MPI_INT, 0, MPI_COMM_WORLD);
}
// 处理没有整除的最后几列
int lastProcessRank = number_of_processes - 1;
if (rank == lastProcessRank || rank == 0)
{
    int leftCol = n - scatterSize * number_of_processes;
    if (leftCol > 0)
    {
        if (rank == lastProcessRank)
        {
            int *leftMatrix = (int *)malloc(n * leftCol * sizeof(int));
            for (int i = 0; i < n; i++)
            {
                for (int j = scatterSize; j < realSize; j++)
                {
                    leftMatrix[i * leftCol + j - scatterSize] = cPart[i * realSize + j];
                }
            }
            MPI_Send(leftMatrix, n * leftCol, MPI_INT, 0, 0, MPI_COMM_WORLD);
            free(leftMatrix);
        }
        else if (rank == 0)
        {
            int *leftMatrix = (int *)malloc(n * leftCol * sizeof(int));
            MPI_Recv(leftMatrix, n * leftCol, MPI_INT, lastProcessRank, 0,
                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            for (int i = 0; i < n; i++)
            {
                for (int j = 0, k = n - leftCol; j < leftCol; j++)
                {
                    Carray[i * n + k + j] = leftMatrix[i * leftCol + j];
                }
            }
            free(leftMatrix);
        }
    }
}
```

最后，在主进程中将计算后的数组转化为 vector，并释放计算过程中 malloc 的内存，接着执行原始文件中的其他代码。

```
if (rank == 0)
{ // array转vector
    parallel_answer = std::vector<std::vector<int>>(n, std::vector<int>(n));
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            parallel_answer[i][j] = Carray[i * n + j];
        }
    }
}
free(cPart);
free(Aarray);
free(Barray);
if (rank == 0)
{
    free(Carray);
}

// ================================================================
// ====    Write your implementation above this line    ====
// ================================================================
// ================================================================
```

CPU 计算

在编写计算部分的代码时，我首先编写的是 CPU 计算代码，完全测试通过后才编写 GPU 计算代码。

首先，从 A 中初始化 C 的片段，并从 B 中初始化 B 的片段，片段的列长度就是每个进程需要计算的列长度。然后，从 1 迭代到 m，计算$D = A \times B^i$，$C+= D$，迭代 $B^{i+1} = B^i \times B = B \times B^i$。这里使用了 memset 和 memcpy 对矩阵进行快速赋值。

```
void ParallelCalculation(const int *A, const int *B, int *CPart,
                         int n, int m, int startCol, int endCol, int rank)
{
    int colomnLength = endCol - startCol + 1;
    int partMatrixSize = n * colomnLength * sizeof(int);
    int BArrayPart[n][colomnLength];
    // 初始化C的片段为A，初始化B的片段
    for (int i = 0; i < n; i++)···
    for (int t = 1; t <= m; t++)
    {
        int D[n][colomnLength]; // D为 A x B^t
        memset(D, 0, partMatrixSize);
        for (int i = 0; i < n; i++)···
        // 把D加到C
        for (int i = 0; i < n; i++)···
        if (t == m)
        {
            break;
        }
        // 通过左乘B来迭代B
        int nextBPartPower[n][colomnLength];
        memset(nextBPartPower, 0, partMatrixSize);
        for (int i = 0; i < n; i++)···
        memcpy(BArrayPart, nextBPartPower, partMatrixSize);
    }
}
```

循环内的细节如下面四部分代码所示。

```
// 初始化C的片段为A，初始化B的片段
for (int i = 0; i < n; i++)
{
    for (int j = startCol; j <= endCol; j++)
    {
        CPart[i * colomnLength + j - startCol] = A[i * n + j];
        BArrayPart[i][j - startCol] = B[i * n + j];
    }
}
```

```
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < colomnLength; j++)
    {
        for (int k = 0; k < n; k++)
        {
            D[i][j] = (D[i][j] + A[i * n + k] * BArrayPart[k][j]) % 2;
        }
    }
}
```

```
// 把D加到C
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < colomnLength; j++)
    {
        CPart[i * colomnLength + j] = (CPart[i * colomnLength + j] + D[i][j]) % 2;
    }
}
```

```
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < colomnLength; j++)
    {
        for (int k = 0; k < n; k++)
        {
            nextBPartPower[i][j] = (nextBPartPower[i][j] + B[i * n + k] * BArrayPart[k][j]) % 2;
        }
    }
}
```

GPU 计算

在 CPU 版 MPI 并行计算代码的基础上，用 cuda 来进行 GPU 加速。首先，使用 cudaMalloc 在 GPU 中分配内存，对于每个进程来说，将矩阵 A、矩阵 B、列分块 C 和列分块 B 的值拷贝到 device 中。然后，将原来的 for 循环更改为调用 kernal 函数，函数调用后需要进行数据同步。最后，将 GPU 中的计算结果拷

贝回 host 的内存中，并释放在 GPU 中申请的内存。

```
int colomnLength = endCol - startCol + 1;
int partMatrixSize = n * colomnLength * sizeof(int);
int fullMatrixSize = n * n * sizeof(int);
int BArrayPart[n][colomnLength];
// 初始化C的片段为A，初始化B的片段
for (int i = 0; i < n; i++)…
int *d_CPart, *d_BArrayPart, *d_A, *d_B, *d_D, *d_nextBPartPower;
cudaMalloc((void **)&d_CPart, partMatrixSize);
cudaMalloc((void **)&d_BArrayPart, partMatrixSize);
cudaMalloc((void **)&d_nextBPartPower, partMatrixSize);
cudaMalloc((void **)&d_A, fullMatrixSize);
cudaMalloc((void **)&d_B, fullMatrixSize);
cudaMalloc((void **)&d_D, partMatrixSize);
cudaMemcpy(d_CPart, CPart, partMatrixSize, cudaMemcpyHostToDevice);
cudaMemcpy(d_BArrayPart, *BArrayPart, partMatrixSize, cudaMemcpyHostToDevice);
cudaMemcpy(d_A, A, fullMatrixSize, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B, fullMatrixSize, cudaMemcpyHostToDevice);
int step = blockPerGrid * threadPerBlock;
for (int t = 1; t <= m; t++)
{
    cudaMemset(d_D, 0, partMatrixSize);
    matrixMultipleKernal<<<blockPerGrid, threadPerBlock>>>(
        d_D, d_A, d_BArrayPart, n, n, colomnLength, step);
    cudaDeviceSynchronize();
    // 把D加到C
    matrixAddKernal<<<blockPerGrid, threadPerBlock>>>(
        d_CPart, d_D, n, colomnLength, step);
    if (t == m)…
    // 通过左乘B来迭代B
    cudaMemset(d_nextBPartPower, 0, partMatrixSize);
    matrixMultipleKernal<<<blockPerGrid, threadPerBlock>>>(
        d_nextBPartPower, d_B, d_BArrayPart, n, n, colomnLength, step);
    cudaDeviceSynchronize();
    cudaMemcpy(d_BArrayPart, d_nextBPartPower, partMatrixSize, cudaMemcpyDeviceToDevice);
}
cudaMemcpy(CPart, d_CPart, partMatrixSize, cudaMemcpyDeviceToHost);
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_D);
cudaFree(d_BArrayPart);
cudaFree(d_nextBPartPower);
cudaFree(d_CPart);
```

kernel 函数分别是矩阵的模 2 乘法和加法。这里使用的是 1 维的下标计算，根据线程块 id 和线程 id 计算行的下标，然后对这一行的每个元素进行计算。计算后把下标增加 gridDim*blockDim，在循环中继续计算，防止线程数多于或少于矩阵的行数。

```
__global__ void matrixAddKernal(int *d_C, int *d_B,
                                int n, int m, int step)
{ // n*m矩阵的%加法，按行划分线程
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    while (idx < n)
    {
        for (int j = 0; j < m; j++)
        {
            d_C[idx * m + j] = (d_C[idx * m + j] + d_B[idx * m + j]) % 2;
        }
        idx += step;
    }
}

__global__ void matrixMultipleKernal(int *d_C, int *d_A, int *d_B,
                                     int n, int m, int w, int step)
{ // n*m和m*w矩阵的%乘法，按行划分线程
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    while (idx < n)
    {
        for (int j = 0; j < w; j++)
        {
            for (int k = 0; k < m; k++)
            {
                d_C[idx * w + j] =
                    (d_C[idx * w + j] + d_A[idx * m + k] * d_B[k * w + j]) % 2;
            }
        }
        idx += step;
    }
}
```

实验结果

环境配置

在实验教室中使用了两台机器进行互联，设置好 IP，ssh-copy-id 发送公钥，测试 ssh 的连接。

对于两台机器，可以设置 hostname 来区分主机。设置指令如下，重启终端即可看到效果。

```
lenovo@10:~$ sudo hostnamectl set-hostname node19
[sudo] lenovo 的密码:
lenovo@10:~$ hostname
node19
```

makefile 中需要加入设置版本的参数，否则可能执行错误。

makefile

Lab04 > PA4-SampleCodeAndData > makefile

```
1  CC = /usr/bin/gcc
2  NVCC = /usr/local/cuda/bin/nvcc
3  NVCCFLAGS = -Xcompiler -g -w -I.. -gencode arch=compute_35,code=sm_35
```

Hostfile 文件内容如下，每个节点设置为最多 2 个进程。

hostfile

Lab04 > PA4-SampleCodeAndData >

```
1  10.251.137.19:2
2  10.251.137.18:2
```

使用 make 命令编译程序，然后使用 scp 指令将程序发送给另一个节点，具体指令如下图所示。

```
lenovo@node19:~/Lab04/PA4-SampleCodeAndData$ scp multiple lenovo@node18:/home/lenovo/Lab04/PA4-SampleCodeAndData/multiple
multiple                                                                    100%  995KB  76.4MB/s   00:00
```

执行测试用例，可以看到程序正确计算出了结果，下图为 test_small_1 的计算结果。

```
lenovo@node19:~/Lab04/PA4-SampleCodeAndData$ make test_small_1
/usr/local/cuda/bin/nvcc -Xcompiler -g -w -I.. -gencode arch=compute_35,code=sm_35 -I /usr/local/software/openmpi/include/  -L /usr/local/software/op
enmpi/lib -lmpi   --std=c++11 multiple.cu -o multiple
nvcc warning : The 'compute_35', 'compute_37', 'compute_50', 'sm_35', 'sm_37' and 'sm_50' architectures are deprecated, and may be removed in a futur
e release (Use -Wno-deprecated-gpu-targets to suppress warning).
--------------------------------------------------------------------------
PMIx was unable to find a usable compression library
on the system. We will therefore be unable to compress
large data streams. This may result in longer-than-normal
startup times and larger memory footprints. We will
continue, but strongly recommend installing zlib or
a comparable compression library for better user experience.

You can suppress this warning by adding "pcompress_base_silence_warning=1"
to your PMIx MCA default parameter file, or by adding
"PMIX_MCA_pcompress_base_silence_warning=1" to your environment.
--------------------------------------------------------------------------
number_of_block_in_a_grid:4
number_of_thread_in_a_block:512
parallel running time:0.0839029
sequential running time:4.477e-06
speed up:5.33593e-05
sum_sequential_answer = 2
sum_parallel_answer = 2
Correct!!!
```

下图为 test_small_2 的计算结果。

```
number_of_block_in_a_grid:1
number_of_thread_in_a_block:512
parallel running time:0.070973
sequential running time:1.1852e-05
speed up:0.000166993
sum_sequential_answer = 3
sum_parallel_answer = 3
Correct!!!
```

下图为 test_small_3 的计算结果。

```
number_of_block_in_a_grid:1
number_of_thread_in_a_block:512
parallel running time:0.0861168
sequential running time:2.0394e-05
speed up:0.000236818
sum_sequential_answer = 2
sum_parallel_answer = 2
Correct!!!
```

在较大的数据集上执行程序，程序计算的结果也是正确的，下图为 test_large_1 的计算结果。

```
lenovo@node19:~/Lab04/PA4-SampleCodeAndData$ make test_large_1
--------------------------------------------------------------------------
PMIx was unable to find a usable compression library
on the system. We will therefore be unable to compress
large data streams. This may result in longer-than-normal
startup times and larger memory footprints. We will
continue, but strongly recommend installing zlib or
a comparable compression library for better user experience.

You can suppress this warning by adding "pcompress_base_silence_warning=1"
to your PMIx MCA default parameter file, or by adding
"PMIX_MCA_pcompress_base_silence_warning=1" to your environment.
--------------------------------------------------------------------------
number_of_block_in_a_grid:4
number_of_thread_in_a_block:128
parallel running time:1.02141
sequential running time:1.86667
speed up:1.82755
sum_sequential_answer = 20087
sum_parallel_answer = 20087
Correct!!!
```

下图为 test_large_2 的计算结果。

```
number_of_block_in_a_grid:2
number_of_thread_in_a_block:64
1 more process has sent help message help-pcompress.txt / unavailable
1 more process has sent help message help-pcompress.txt / unavailable
parallel running time:13.3593
sequential running time:13.6982
speed up:1.02537
sum_sequential_answer = 44722
sum_parallel_answer = 44722
Correct!!!
```

下图为 test_large_3 的计算结果。

```
number_of_block_in_a_grid:1
number_of_thread_in_a_block:512
1 more process has sent help message help-pcompress.txt / unavailable
parallel running time:6.0118
1 more process has sent help message help-pcompress.txt / unavailable
sequential running time:15.3867
speed up:2.55941
sum_sequential_answer = 179230
sum_parallel_answer = 179230
Correct!!!
```

可以看到在较大的数据集下，程序分别取得了 1.83、1.03 和 2.56 倍的速度提升。