1.串行通用矩阵乘法

串行通用矩阵乘法没什么需要注意的(可能注意一下越界和优化问题就行了)，代码如下：

```
for(int i = 0; i < M; ++i){
    for(int k = 0; k < K; ++k){
        for(int j = 0; j < N; ++j){
            C[i][k] += A[i][j] * B[j][k];
        }
    }
}
```

优化代码：

- 调整循环顺序：

```
for(int i = 0; i < M; ++i){
    for(int j = 0; j < N; ++j){
        for(int k = 0; k < K; ++k){
            C[i][k] += A[i][j] * B[j][k];
        }
    }
}
```

- 循环展开：

```
for(int i = 0; i < M; ++i){
    for(int k = 0; k < K; ++k){
        for(int j = 0; j < N; j+=stride){
            C[i][k] += A[i][j] * B[j][k];
            C[i][k] += A[i][j+1] * B[j+1][k];
            ......
            C[i][k] += A[i][j+(stride-1)] * B[j+(stride-1)][k];
        }
    }
}
```

2.基于分布式内存的并行编程框架的矩阵乘法(MPI)

首先要导入头文件`mpi.h`。

在使用MPI之前，同样需要导入头文件`mpi.h`，然后在使用MPI执行并行任务之前必须要先进行MPI初始化，在最后任务结束时必须结束MPI：

```
MPI_Init(&argc,&argv);  // MPI初始化
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);  // 获取当前进程的进程号
MPI_Comm_size(MPI_COMM_WORLD,&process_num); // 获取进程数量
......// 调用MPI的API以及其他函数完成矩阵乘法任务
MPI_Finalize(); // MPI结束
```

MPI有点对点通信和集合通信两种方式：

- 点对点通信：

  点对点通信需要注意的是两个进程进行通信时如何确保信息传输无误，而这主要是通过源进程号和`tag`来保证的：

```
int rows_per_process = M / process_num; //每个进程参与计算的矩阵A的行数
vector<double> local_C(M*K);    //// 用来存每个非 0 号进程的计算结果
if(myrank == 0){    // 0号进程
    for(int i = 0; i < process_num; ++i){
        MPI(B.data(), N*K, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);   // 将矩阵B发送给其他进程
        MPI(A.data()+rows_per_process*(i-1)*N, M/process_num*N, MPI_DOUBLE, i, 1, MPI_COMM_WORLD);  //将矩阵A的分块发送给对应的进程
    }

    // 完成剩余部分的计算
    for(int i = (process_num-1)*rows_per_process; i < M; ++i){
        for(int k = 0; k < K; ++k){
            double local_sum = 0;
            for(int j = 0; j < N; ++j){
                local_sum += A[i*N+j] * B[j*K+k];
            }
            C[i*K+k] = local_sum;
        }
    }


    for(int i = 1; i < process_num; ++i){
        MPI_Recv(local_C.data(), rows_per_process*K, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);  //接收来自其他进程的计算结果
        for(int j = 0; j < rows_per_process; ++j){
            for(int k = 0; k < K; ++k){
                C[((i-1)*rows_per_process+j)*K+k] = local_C[j*K+k];
            }
        }
    }
}
else{   // 其他进程
    MPI_Recv(B.data(), N*K, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);   //接收矩阵B
    vector<double> local_A(rows_per_process*N); //使用一个本地数组A来存储矩阵A的分块
    MPI_Recv(local_A.data(), rows_per_process*N, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);  // 接收0号进程发送过来的矩阵A的分块

    // 进行矩阵乘法
    for(int i = 0; i < rows_per_process; ++i){
```

```
        for(int k = 0; k < K; ++k){
            double local_sum = 0;
            for(int j = 0; j < N; ++j){
                local_sum += local_A[i*N+j] * B[j*K+k];
            }
            local_C[i*K+k] = local_sum;
        }
    }

    MPI_Send(local_C.data(), rows_per_process*K, MPI_DOUBLE, 0, 0,
MPI_COMM_WORLD); //将计算结果发送给0号进程
}
```

- 集合通信：

```
int rows_per_process = M / process_num;
vector<double> local_A(rows_per_process*N); // 接收A的分块
vector<double> local_C(M*K);    // 存储每个进程的计算结果
MPI_Bcast(B.data(), N*K, MPI_DOUBLE, 0, MPI_COMM_WORLD);    // 将矩阵B广播到各个进程
MPI_Scatter(A.data(), rows_per_process*N, MPI_DOUBLE, local_A.data(),
rows_per_process*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);   // 0号进程将矩阵A分发出去，所有
进程都会接收到对应的A的分块

// 每个进程计算矩阵A的分块和矩阵B的乘法结果
for(int i = 0; i < row_process_num; ++i){
    for(int k = 0; k < K; ++k){
        double local_sum = 0;
        for(int j = 0; j < N; ++j){
            local_sum += local_A[i*N+j] * B[j*K+k];
        }
        local_C[i*K+k] = local_sum;
    }
}
MPI_Gather(local_C.data(), rows_per_process*K, MPI_DOUBLE, C.data(),
rows_per_process*K, MPI_DOUBLE, 0, MPI_COMM_WORLD);    // 汇总计算结果
if(myrank == 0){
    // 计算矩阵A剩余的(M%process_num)行与矩阵B的乘积
    for(int i = (process_num-1)*rows_per_process; i < M; ++i){
        for(int k = 0; k < K; ++k){
            double local_sum = 0;
            for(int j = 0; j < N; ++j){
                local_sum += A[i*N+j] * B[j*K+k];
            }
            C[i*K+k] = local_sum;
        }
    }
}
```

3.基于共享内存的CPU多线程编程(Pthread)

定义一个多线程并行矩阵乘法函数，需要注意的是函数的参数类型和返回类型均为`void*`：

```
void* parallel_gemm(void* id){
    int thread_id = *((int*)id);    // 获取线程号
    int beginRow = thread_id*rows_per_thread, endRow =
(thread_id+1)*rows_per_thread;   // 计算出每个线程参与计算的矩阵A的行(左闭右开),
rows_per_thread为每个线程参与计算的矩阵A的行数，是一个全局变量
    for(int i = beginRow; i < endRow; ++i){
        for(int k = 0; k < K; ++k){
            double local_sum = 0;
            for(int j = 0; j < N; ++j){
                local_sum += A[i*N+j] * B[j*K+k];
            }
            C[i*K+k] = local_sum;
        }
    }
    pthread_exit(NULL); //结束线程
}
```

在`main`函数(进程)中要为各个线程分配空间，然后定义线程号数组(在执行矩阵乘法操作时需要使用线程号来找到指定的行数，线程号实际上是作为并行矩阵乘法函数的形参):

```
pthread_t handles[THREAD_NUM];  // 为各个线程分配空间，实际上使用动态分配可能会更好一点
int thread_ids[THREAD_NUM]; // 线程号数组
```

接下来要使用`pthread_create`创建并启动线程，指明其执行内容(`parallel_gemm`函数)，在线程执行完毕之后，使用汇聚操作(`pthread_join`)合并执行结果:

```
// 使用pthread_create创建并启动线程
for(int i = 0; i < THREAD_NUM; ++i){
    thread_ids[i] = i;  // 指明线程号
    pthread_create(&handles[i], NULL, parallel_gemm, (void*)&thread_ids[i]);
}

// 使用pthread_join合并执行结果
for(int i = 0; i < THREAD_NUM; ++i){
    pthread_join(handles[i], NULL);
}
// 如果前面是动态分配线程内存，则需要手动释放(malloc-free/new-delete)
```

由于我在使用**Pthreads**实现多线程矩阵乘法时并未出现竞争，但是在使用**Pthreads**时要注意竞争条件的影响，具体解决方案可以参考理论分析报告，给出了多种解决方案。

4.GPU多线程编程CUDA

CUDA编程的访存方式分为全局内存访存和共享内存访存。

- 使用全局内存访存:

  需要注意计算网格大小时要覆盖到整个矩阵以及当前线程参与计算的矩阵A的行号和B列号的计算。

  具体实现如下:

  定义核函数:

  ```
  ```

```
__global__ void matrix_multiply_global(double* A, double* B, double* C, int
M, int N, int K){
    int row = blockIdx.y * blockDim.y + threadIdx.y;//计算行号
    int col = blockIdx.x * blockDim.x + threadIdx.x;//计算列号

    if(row < M && col < K){
        double value = 0;

        //计算A的第row行和B的第col列的乘积，将其存储在C的第row行第col列
        for(int i = 0; i < N; ++i){
            value += A[row * N + i] * B[i * K + col];
        }
        C[row * K + col] = value;
    }
}
```

确定线程块大小和维度、网格大小和维度：

```
dim3 dimBlock(blockDim_x, blockDim_y, 1);
dim3 dimGrid((K + blockDim_x - 1) / blockDim_x, (M + blockDim_y - 1) /
blockDim_y, 1);
```

调用核函数：

```
matrix_multiply_global<<<dimGrid, dimBlock>>>(A, B, C, M, N, K);
```

- 使用共享内存访存：

使用共享内存访存需要注意的问题比较多，除了全局内存访存需要注意的问题之外，还需要注意：

①使用静态共享内存和动态共享内存时的区别，使用动态共享内存时，在调用核函数时除了网格信息、线程块信息之外再增加一个动　态共享内存的大小，声明共享内存时要再家一个`extern`关键字，而使用静态共享内存时则不用。

②将矩阵存储到共享内存时，要使用`__syncthreads()`进行同步。

③使用共享内存还要注意避免`bank`冲突，即多个线程块同时访问同一个`bank`的内容是会出现冲突，可以通过扩大共享内存即多设一　列来避免这个问题。

- **具体实现如下：**

定义核函数：

```
__global__ void matrix_multiply_shared(double* A, double* B, double* C, int
M, int N, int K, int tile_size){
    extern __shared__ double smem[];//一个大小为2*(tile_size)*(tile_size+1)的共
享内存，前半部分存储矩阵A的块，后半部分存储矩阵B的块
    double *smemA = smem, *smemB = smem + (tile_size+1) * tile_size;
    int row = blockIdx.y * blockDim.y + threadIdx.y, col = blockIdx.x *
blockDim.x + threadIdx.x;//获取行号和列号
    double value = 0;

    // 接下来将当前块内的矩阵A和B的相应的元素存储到共享内存中并进行计算
    for (int i = 0; i < N / tile_size; ++i) {
        if (row < M && (i * tile_size + threadIdx.x) < N) {
```

```
            smemA[threadIdx.y * (tile_size + 1) + threadIdx.x] = A[row * N +
i * tile_size + threadIdx.x];
        } else {
            smemA[threadIdx.y * (tile_size + 1) + threadIdx.x] = 0.0;
        }

        if (col < K && (i * tile_size + threadIdx.y) < N) {
            smemB[threadIdx.y * (tile_size + 1) + threadIdx.x] = B[(i *
tile_size + threadIdx.y) * K + col];
        } else {
            smemB[threadIdx.y * (tile_size + 1) + threadIdx.x] = 0.0;
        }

        __syncthreads();
        for (int j = 0; j < tile_size; ++j) {
            value += smemA[threadIdx.y * (tile_size + 1) + j] * smemB[j *
(tile_size + 1) + threadIdx.x];
        }

        __syncthreads();
    }

    if (row < M && col < K) {
        C[row * K + col] = value;
    }

}
```

确定线程块大小和维度、网格大小和维度：

```
dim3 dimBlock(blockDim_x, blockDim_y, 1);
dim3 dimGrid((K + blockDim_x - 1) / blockDim_x, (M + blockDim_y - 1) /
blockDim_y, 1);
int tile_size = blockDim_x;//这里使用共享内存时仅考虑所取线程块大小均为D*D即线程数开
二次方后仍为整数
```

调用核函数：

```
matrix_multiply_shared<<<dimGrid, dimBlock, 2 * tile_size * (tile_size+1) *
sizeof(double)>>>(A, B, C, M, N, K, tile_size);
```