实验五 数据级并行实验

一、实验约定

- 为了避免复杂，本次试验涉及到的矩阵运算的规模 $(2^n, 2^n) \times (2^n, 2^n) = (2^n, 2^n)$，每个程序的矩阵规模需要由参数传入程序，以便考察不同规模的矩阵乘法的性能。
- 为了消除编译器优化的影响，在 CPU 与 GPU 平台上的编译优化参数可以自行选择，你需要提供 `Makefile` 来辅助编译你的程序。
- 在 CPU 与 GPU 平台上的实验的数据类型为 `float32`。
- 在 CPU 平台上计时请包含 `time.h` 文件，使用 `clock()` 函数计时；在 GPU 上请使用 `nvprof` 或 `nsys` 工具对你写的矩阵乘法kernel的时间进行profiling。
- 在 CPU 平台上请使用动态内存分配申请矩阵的空间（为一维数组形式），随机数初始化两个参与计算的矩阵 A 和 B，随机初始化的目的是为了验证计算结果的正确性；在 GPU 上请先在 Host 端使用动态内存分配申请矩阵的空间（为一维数组形式），随机数初始化两个参与计算的矩阵 A 和 B，随机初始化的目的是为了验证计算结果的正确性。
- 本实验无线下检查环节，请各位同学将实验源代码与实验报告打包上传。

二、CPU

任务 1. 基础矩阵乘法

实现一个经典的三重嵌套循环的矩阵乘法。

核心代码：

```
void gemm_baseline(float *A, float *B, float *C) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            float sum = 0.0;
            for (int k = 0; k < N; k++) {
                sum += A[i*N+k] * B[k*N+j];
            }
            C[i*N+j] = sum;
        }
    }
}
```

任务 2. AVX 矩阵乘法

使用 C 语言，通过包含 `immintrin.h` 实现一个简单的 AVX 矩阵乘法。

需要使用 CPU 任务 1 中的基础矩阵乘法验证计算的正确性，验证结束后在性能测量阶段可以不进行正确性的验证。

需要用到以下 AVX 的指令：

- `_mm256_setzero_ps`： Returns a floating-point vector filled with zeros
- `_mm256_broadcast_ss`：复制一个值到一个长度为 8 的向量中
- `_mm256_loadu_ps`：从一个地址开始，加载连续的 8 个浮点数
- `_mm256_fmadd_ps`：Fused Multiply-Add 乘加指令，product to a third (res = a * b + c)

- `_mm256_storeu_ps`：将结果存回原数组，从给定的地址开始连续存 8 个 float

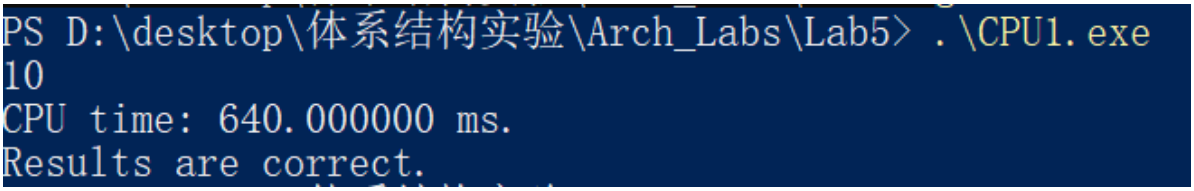
核心代码：

```
void gemm_avx(float *A, float *B, float *C) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j+=8) {
            __m256 c = _mm256_setzero_ps();
            for (int k = 0; k < N; k++) {
                c = _mm256_fmadd_ps(_mm256_broadcast_ss(&A[i*N+k]),
_mm256_loadu_ps(&B[k*N+j]), c);
            }
            _mm256_storeu_ps(&C[i*N+j], c);
        }
    }
}
```

编译指令：

```
gcc -mfma -o CPU1 .\CPU1.c
```

执行结果：



注：在最后的性能测试阶段，正确性验证部分的代码被被注释。

任务 3. AVX 分块矩阵乘法

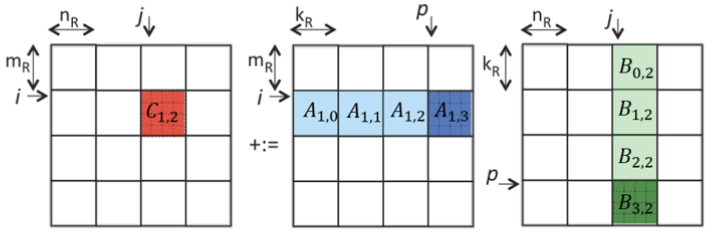
先前的 AVX 实现由于仍然是三重循环为主体，访存跨度较大，并未充分利用 cache 的局部性。

需要使用 CPU 任务 1 中的基础矩阵乘法验证计算的正确性，验证结束后在性能测量阶段可以不进行正确性的验证。

需要对 B 矩阵进行转置

分块优化的目标是通过对**输入数据进行分块**尽可能使**内存访问连续**，提高cache命中率，进而提升程序的整体性能。

$$C = \begin{pmatrix} C_{0,0} & C_{0,1} & \cdots & C_{0,N-1} \\ C_{1,0} & C_{1,1} & \cdots & C_{1,N-1} \\ \vdots & \vdots & & \vdots \\ C_{M-1,0} & C_{M-1,1} & \cdots & C_{M-1,N-1} \end{pmatrix} = \begin{pmatrix} A_{0,0} & A_{0,1} & \cdots & A_{0,K-1} \\ A_{1,0} & A_{1,1} & \cdots & A_{1,K-1} \\ \vdots & \vdots & & \vdots \\ A_{M-1,0} & A_{M-1,1} & \cdots & A_{M-1,K-1} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & \cdots & B_{0,N-1} \\ B_{1,0} & B_{1,1} & \cdots & B_{1,N-1} \\ \vdots & \vdots & & \vdots \\ B_{K-1,0} & B_{K-1,1} & \cdots & B_{K-1,N-1} \end{pmatrix}$$



$$C_{1,2} = A_{1,0}B_{0,2} + A_{1,1}B_{1,2} + A_{1,2}B_{2,2} + A_{1,3}B_{3,2}$$

$$C_{i,j} := \sum_{p=0}^{K-1} A_{i,p}B_{p,j} + C_{i,j}$$

$C_{1,2}$ 矩阵，就由 A 中的4个小矩阵与 B 中的4个小矩阵分别相乘，然后累加得出。

参考：[矩阵乘法的分块优化_矩阵乘法分块]

核心代码：

```
void gemm_avx_block(float* A, float* B, float* C) {
    for (int i = 0; i < N; i += BLOCK_SIZE) {
        for (int j = 0; j < N; j += BLOCK_SIZE) {
            for (int k = 0; k < N; k += BLOCK_SIZE) {
                for (int ii = i; ii < i + BLOCK_SIZE; ++ii) {
                    for (int jj = j; jj < j + BLOCK_SIZE; jj += 8) {
                        __m256 c = _mm256_loadu_ps(&C[ii * N + jj]);
                        for (int kk = k; kk < k + BLOCK_SIZE; ++kk) {
                            c = _mm256_fmadd_ps(_mm256_broadcast_ss(&A[ii * N +
kk]), _mm256_loadu_ps(&B[kk * N + jj]), c);
                        }
                        _mm256_storeu_ps(&C[ii * N + jj], c);
                    }
                }
            }
        }
    }
}
```

编译指令：

```
gcc -mfma -o CPU1 .\CPU2.c
```

执行结果：

```
PS D:\desktop\体系结构实验\Arch_Labs\Lab5> .\CPU1.exe
10
CPU time: 678.000000 ms.
Results are correct.
```

注：在最后的性能测试阶段，正确性验证部分的代码被被注释。

此外，实验文档中提到“可能需要对 B 转置”，也对这种思路进行了实现，保存为 `CPU2_1.c`，在本文档的下一小节中也进行了分析。

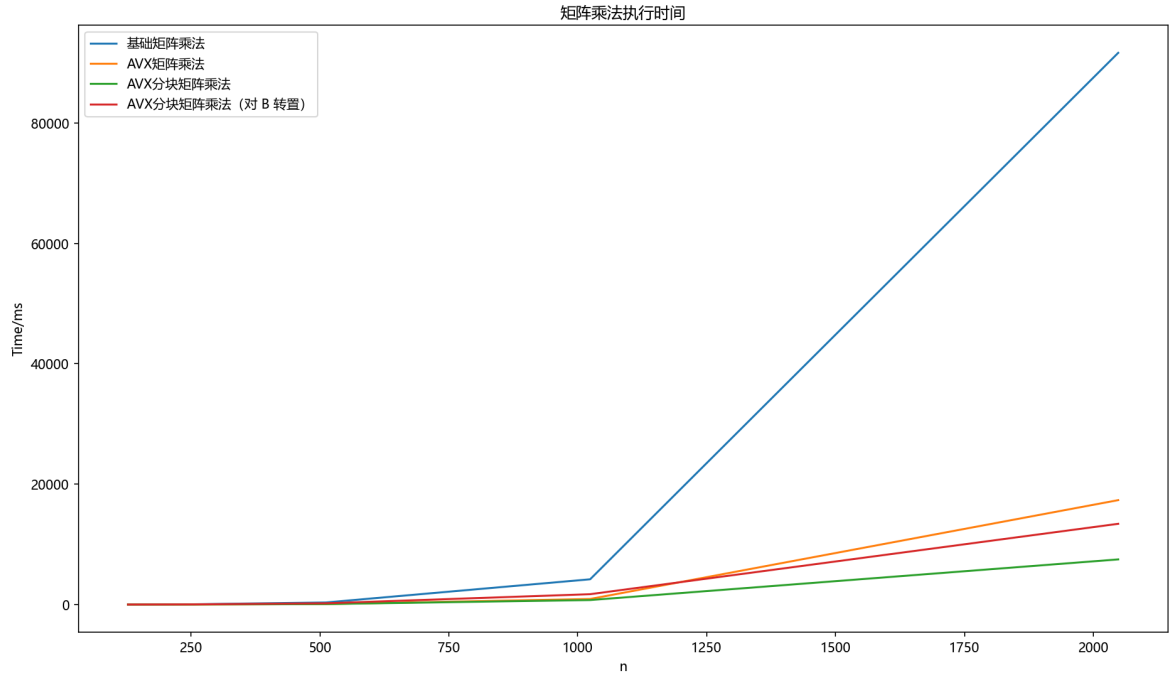
任务 4. 测试结果与比较

1. 不同规模的输入，三种实现的性能差异

时间单位：毫秒 ms

实现	128	256	512	1024	2048
基础矩阵乘法	4.83	39.48	344.29	4192	91657
AVX 矩阵乘法	0.99	8.65	72.96	922.9	17343
AVX 分块矩阵乘法	1.28	9.18	78.30	743.2	7496
AVX 分块矩阵乘法（对 B 转置）	3.37	27.44	208.6	1717	13412

做图如下：



可以看到，对于分块 AVX 矩阵乘法，对 B 转置后运行时间反而变长。所以实际采用的版本没有对 B 转置。

分析原因如下：

- 实验文档中提到“可能需要对 B 转置”，这样对 B 的访问确实是行主序的，看上去更有助于分块后的局部性，但是在使用 `_mm256_fmadd_ps`，对 B 矩阵原本是需要从 `&B[jj*N+kk]` 开始的连续的 8 个值，转置后，变成 `&B[jj*N+kk]` 以及 `&B[(jj+1)*N+kk]`，`&B[(jj+2)*N+kk]`…`&B[(jj+7)*N+kk]`。若想继续使用 `_mm256_fmadd_ps`，则需要额外引入以下语句处理：

```
float B_elements[8] = { B[(jj)*N + kk], B[(jj + 1) * N + kk], B[(jj + 2) * N
+ kk], B[(jj + 3) * N + kk], B[(jj + 4) * N + kk], B[(jj + 5) * N + kk],
B[(jj + 6) * N + kk], B[(jj + 7) * N + kk] };
__m256 b = _mm256_set_ps(B_elements[7], B_elements[6], B_elements[5],
B_elements[4], B_elements[3], B_elements[2], B_elements[1], B_elements[0]);
```

这额外增加了访存开销，实际测试结果不如不转置。所以之后分析不再考虑对 B 转置。

对于三个版本的实现，可以看到，整体趋势上，运算效率：

AVX 分块矩阵乘法 > AVX 矩阵乘法 > 基础矩阵乘法

分析原因如下：

1. **基础矩阵乘法：** 这是最简单的实现方式，使用了三层嵌套循环，没有任何特别的优化。这种实现中，矩阵元素的存取模式会导致较低的缓存命中率，尤其是对于大型矩阵。此外，此实现没有利用现代 CPU 的向量化能力，所以效率也比较低。
2. **AVX矩阵乘法：** 这种实现利用了 AVX 指令，使得在每个操作中能够并行处理多个数据。这显著提高了代码的并行性，从而提高了性能。然而，这种实现并没有解决缓存利用率问题。
3. **AVX分块矩阵乘法：** 这种实现进一步提高了性能，因为它使用了分块技术来提高缓存利用率。通过将矩阵划分为多个小块，可以确保在计算一个小块的时候，所有的数据都能够在缓存中。此外，由于在计算一个小块时都是在一个连续的内存区域上工作，所以预取（prefetching）效果更好，这也有助于提高性能。

2. AVX 分块矩阵乘法，不同的分块参数对性能的影响

以下测试是在输入规模 N = 1024，时间单位为毫秒：

分块大小	8	16	32	64	128	256	512	1024
时间	748.3	691.7	631.4	586.9	714.1	608.6	665.1	838.7

以下测试是在输入规模 N = 2048，时间单位为毫秒：

分块大小	8	16	32	64	128	256	512	1024
时间	8220	6245	5405	5052	4980	5090	5779	8392

可以看到，两种输入规模下，随着分块大小逐渐变大，运行时间先下降后增大。

分析原因：增加分块大小时，以下两个效应会影响性能：

1. **更好的数据局部性：** 当分块大小增加时，在每个块内执行更多的操作。因为数据在内存中是连续存储的，程序可以充分利用缓存的空间和时间局部性。这意味着在同一时间片内，更多的数据可以从缓存（而不是主内存）中读取，从而提高性能。
2. **缓存溢出：** 然而，如果分块大小过大，那么整个块的数据可能无法完全装入缓存。这时， CPU 需要从主内存中频繁地读取数据，造成缓存失效。这会极大地降低性能，因为主内存的访问速度远低于缓存。

因此，随着分块大小的增大，运行时间先下降后增大。开始时，增加分块大小会提高数据局部性，从而提高性能。但是，当分块大小超过一定阈值（通常接近或等于缓存大小）时，缓存溢出效应开始主导，性能开始下降。

任务 5. 调研 CPU 平台上其它矩阵乘法的优化手段

线程并行化： 在具有多个处理核心的现代 CPU 上，可以使用多线程来并行化矩阵乘法。每个线程处理矩阵的一个或多个块。这种方式可以充分利用 CPU 的多核心特性。

算法上的优化：

- **Strassen算法：**这是一种快速矩阵乘法算法，它通过将矩阵分解成更小的矩阵，并通过一系列的矩阵加法和乘法运算，从而减少了基本乘法运算的次数。其时间复杂度为$O(n^{log2(7)})$，低于经典矩阵乘法的 $O(n^3)$。
- **Coppersmith-Winograd算法：**这是一种更快的矩阵乘法算法，其时间复杂度接近 $O(n^{2.376})$。虽然理论上它比 Strassen 算法更快，但由于它的常数因子较大，在实际应用中并不总是更快。

使用更高级别的矩阵库： 有许多经过优化的矩阵运算库，如 Intel 的 MKL（Math Kernel Library）和 OpenBLAS 等。这些库使用了许多针对特定硬件特性的优化技术，通常可以提供超过手写代码的性能。

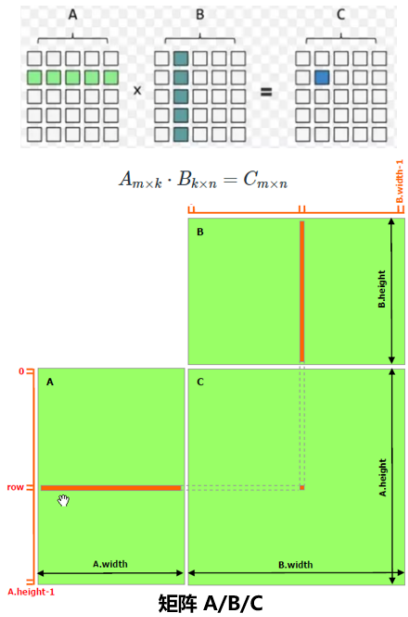
循环展开（Loop Unrolling）： 在矩阵乘法中，可以将内部循环展开到一定的宽度，可以减少循环的开销并增加并行性。

三、GPU

任务 1. 基础矩阵乘法

初步了解 GPU 的 SIMT 的编程模型以及 GPU 中的层级结构，写一个简单的矩阵乘法 kernel：

- 任务1-基础矩阵乘法



矩阵 A/B/C

GPU 中执行矩阵乘法运算操作：

a) 在 Global Memory 中分别为矩阵 A、B、C 分配存储空间。

b) 由于矩阵 C 中每个元素的计算均相互独立, NVIDIA GPU 采用的 SIMT (单指令多线程)的体系结构来实现并行计算的, 因此在并行度映射中, 我们让每个 Thread 对应矩阵 C 中1个元素的计算。

c) 执行配置 (execution configuration)中 gridSize 和 blockSize 均有 x(列向)、y(行向)两个维度. 其中,

$$gridSize.x \times blockSize.x = n$$
$$gridSize.y \times blockSize.y = m$$

每个 Thread 需要执行的：从矩阵 A 中读取一行向量 (长度为k), 从矩阵 B 中读取一列向量 (长度为k), 对这两个向量做点积运算 (单层 k 次循环的乘累加), 最后将结果写回矩阵 C。

参考：[GitHub - fanghao6666/CUDA-Matirx-Multiplication]

kernel 代码：

```
__global__ void matrixMulOnGPU(float* m_a, float* m_b, float* m_r, unsigned int
m, unsigned int n, unsigned int k)
{
    int threadId = (blockIdx.y * blockDim.y + threadIdx.y) * gridDim.x *
blockDim.x + blockIdx.x * blockDim.x + threadIdx.x;
    if (threadId >= m * k)
        return;
```

```
    int row = threadId / k;
    int col = threadId % k;

    m_r[threadId] = 0;
    for (size_t i = 0; i < n; ++i)
    {
        m_r[threadId] += m_a[row * n + i] * m_b[i * k + col];
    }
}
```

在 Host 端调用：

```
dim3 blockSize(16, 16);
dim3 gridSize((m + blockSize.x - 1) / blockSize.x, (k + blockSize.y - 1) /
blockSize.y);

matrixMulOnGPU<<<gridSize, blockSize>>>(d_a, d_b, d_c, m, n, k);
```

编译运行：

```
nvcc GPU0.cu -o GPU0
```

这段代码在 n <= 7 时返回 correct，即 GPU 计算的结果和 CPU 计算的结果相同，但是 n>=8 之后，二者的结果不同

这个问题很可能是由于浮点数的累加顺序引起的。

当计算矩阵乘法的和时，每个元素是由许多浮点数的乘积累加而成的。在数学上，加法是具有结合性的，即 `(a + b) + c == a + (b + c)`。但在浮点数运算中，由于精度限制，这个等式可能并不成立。也就是说，改变浮点数加法的顺序可能会改变结果。

在本程序中，CPU 版本的矩阵乘法和 GPU 版本的矩阵乘法可能在计算元素的和时使用了不同的顺序。特别地，由于 GPU 并行化的特性，某些加法可能被重新排序了。这可能导致当增大矩阵的大小时（从而增加了每个和的项数），两个版本的计算结果开始出现差异。

所以修改判断函数的精度：

```
if(abs(h_c[i] - v_c[i]) > 0.1)
```

或者如果任务本身需要更高的精度，那么可以更改 float 为 double 类型。

运行结果：

```
root@kkeGPU:~/Arch_Labs/Lab5# nvcc GPU0.cu -o GPU0
root@kkeGPU:~/Arch_Labs/Lab5# ./GPU0
9
Results are correct.
```

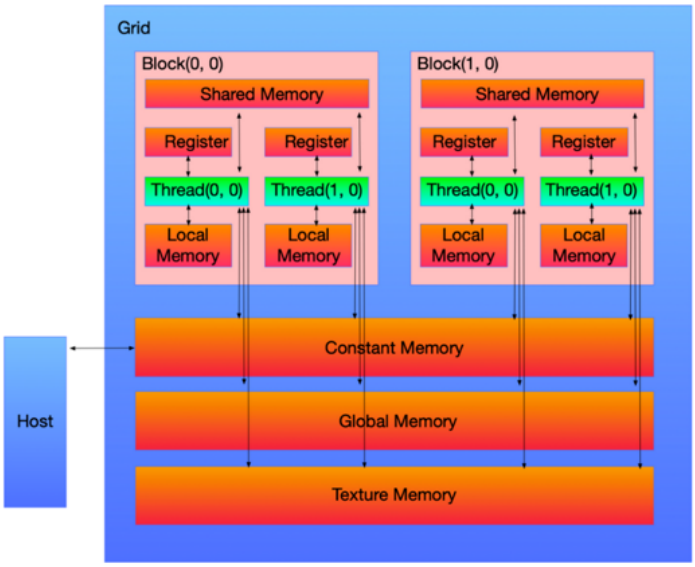
注：在最后的性能测试阶段，正确性验证部分的代码被被注释。

任务 2. 分块矩阵乘法

任务 1 完成了一个简单的 kernel 以实现矩阵乘法，但是其在访存的性能上是糟糕的，所有数据都在 `global memory` 中，加载数据非常耗时。在 GPU 的存储层次中，有访问相对更快的 `shared memory`，但其通常较小，不能存储整个矩阵，因此需要实现一个分块矩阵乘法，以达成更高的性能，代码框架与任务 1 相似，需要额外定义一个分块因子 `BLOCK` 来控制矩阵分块的粒度。

GPU 的计算核心都在 Streaming Multiprocessor（SM）上，SM 里有计算核心可直接访问的寄存器（Register）和共享内存（Shared Memory）；多个 SM 可以读取显卡上的显存，包括全局内存（Global Memory）。每个 SM 上的 Shared Memory 相当于该 SM 上的一个缓存，一般都很小，Telsa V100 的 Shared Memory 也只有 96KB。

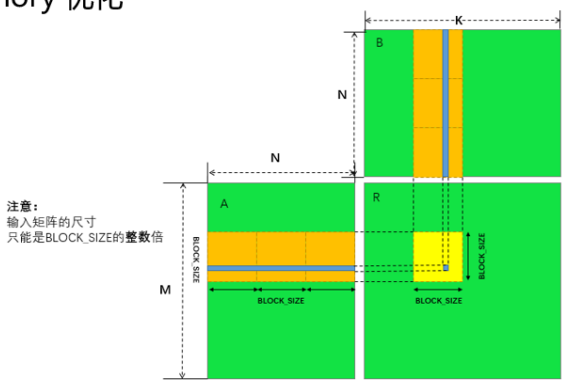
CUDA 的线程可以访问不同级别的存储，每个 Thread 有独立的私有内存；每个 Block 中多个 Thread 都可以在该 Block 的 Shared Memory 中读写数据；整个 Grid 中所有 Thread 都可以读写 Global Memory。



GPU CUDA的架构图

- ✓ **Grid**内部有Constant Memory, **Global Memory** 和 Texture Memory
- ✓ **Block**内部有**Shared Memory**
- ✓ Thread对应有Local Memory和Register

- Shared Memory 优化



矩阵 A/B/C分块处理

采用Shared Memory 优化的GPU矩阵乘法：

- 矩阵C分为若干小块，每个小块的大小为32 * 32 （1024），每个小块作为一个Block,计算这个block内部的数据只需要A矩阵中的32 * N的子矩阵和B矩阵中的N * 32的子矩阵，于是我们将A和B的子矩阵一次性从Global Memory 导入到Block的shared Memory内部，然后再对Block内部的所有Thread进行计算。接下来让子矩阵块分别在矩阵 A 的行向以及矩阵 B 的列向上滑动。利用shared memory可以大大的减少线程对于Global Memory的数据传输，极大的减少了时间消耗。

参考：[GitHub - fanghao6666/CUDA-Matirx-Multiplication]

kernel 代码：

```
__global__ void matrixMulOnGPUWithShared(float* m_a, float* m_b, float* m_r,
unsigned int m, unsigned int n, unsigned int k)
{
    if ((blockIdx.y * blockDim.y + threadIdx.y) * k + blockIdx.x * blockDim.x +
 threadIdx.x >= m * k)
        return;

    const int begin_a = blockIdx.y * blockDim.y * n;
    const int end_a = begin_a + n - 1;
    const int step_a = blockDim.x;

    const int begin_b = blockIdx.x * blockDim.x;
    const int step_b = blockDim.y * k;

    float result_temp = 0.0f;

    for (int index_a = begin_a, index_b = begin_b; index_a < end_a; index_a +=
step_a, index_b += step_b)
    {
        __shared__ float SubMat_A[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float SubMat_B[BLOCK_SIZE][BLOCK_SIZE];
        // 每轮循环处理A和B的BLOCK_SIZE*BLOCK_SIZE的小矩阵

        SubMat_A[threadIdx.y][threadIdx.x] = m_a[index_a + threadIdx.y * n +
threadIdx.x];
        SubMat_B[threadIdx.y][threadIdx.x] = m_b[index_b + threadIdx.y * k +
threadIdx.x];
        // Share Memory是每个Block内部的，所以只需要区分Block内部的线程号

        __syncthreads(); // 确保所有线程都已经完成了数据的读取

        for (int i = 0; i < BLOCK_SIZE; ++i)
```

```
        {
            result_temp += SubMat_A[threadIdx.y][i] * SubMat_B[i][threadIdx.x];
        }

        __syncthreads(); // 确保所有线程都已经完成了计算
    }

    int begin_result = blockIdx.y * blockDim.y * k + begin_b;
    m_r[begin_result + threadIdx.y * k + threadIdx.x] = result_temp;
}
```

运行结果：

```
root@kkeGPU:~/Arch_Labs/Lab5# nvcc GPU1.cu -o GPU1
root@kkeGPU:~/Arch_Labs/Lab5# ./GPU1
9
Results are correct.
```

注：在最后的性能测试阶段，正确性验证部分的代码被被注释。

任务 3. 测试结果与比较

1. 不同规模的输入，两种实现的性能差异

两种实现对于相同的输入规模，运行时间的主要区别在于 kernel 的执行时间。

测试时间性能结果如下，以下是不同输入规模、不同 blocksize 的时间性能（时间单位：毫秒）：

输入规模	blocksize	基础实现的 kernel	分块后的 kernel
512	4	6.785	3.82
512	8	3.480	1.16
512	16	2.100	0.74
512	32	1.737	0.70
1024	4	56.59	30.33
1024	8	27.21	9.11
1024	16	16.20	5.79
1024	32	14.73	5.26
2048	4	383.5	231.6
2048	8	93.50	73.0
2048	16	74.77	46.3
2048	32	71.19	41.5
4096	4	2461	960
4096	8	843	377

输入规模	blocksize	基础实现的 kernel	分块后的 kernel
4096	16	647	310
4096	32	585	272

注：分块矩阵的实现中，每个 Block 的大小与其要处理的矩阵 C 的大小相同。

可以看到，分块实现的矩阵乘法的 kernel 执行时间明显少于基础版本。前者的执行效率基本为后者的两倍。

分析原因：分块实现使用了共享内存（shared）来缓存 A 和 B 矩阵的一部分，而这部分内存比全局内存访问更快。这种方式可以减少全局内存访问次数，并且增加了访问共享内存的次数，所以可以加速矩阵乘法操作。对于基础实现，每次计算都需要从全局内存中获取数据，这会导致较高的内存访问延迟。

2. 不同的 `gridsize` 和 `blocksize` 对基础矩阵乘法性能的影响

```
dim3 blockSize(BLOCK_SIZE, BLOCK_SIZE);
dim3 gridSize((N + blockSize.x - 1) / blockSize.x, (N + blockSize.y - 1) /
blockSize.y);
```

所以 `gridsize` 是由输入规模 `N` 和 `blocksize` 决定的。

而工具 `nvprof` 提供了代码运行的不同部分的时间性能，消耗时间的操作主要为：

- kernel 执行
- 将数据从 Host 拷贝到 Device
- 将数据从 Device 拷贝到 Host
- Host 端的调用的 cudaMalloc

所以之后的测试结果会分别列出以上四种时间。

以下是不同输入规模、不同 blocksize 的时间性能（时间单位：毫秒）：

输入规模	blocksize	kernel	CUDA memcpy HtoD	CUDA memcpy DtoH	cudaMalloc
512	4	6.785	0.176	0.118	254
512	8	3.480	0.178	0.117	229
512	16	2.100	0.177	0.117	234
512	32	1.737	0.176	0.117	224
1024	4	56.59	1.333	0.607	227
1024	8	27.21	1.312	0.599	231
1024	16	16.20	1.279	0.605	236
1024	32	14.73	1.309	0.604	229
2048	4	383.5	5.771	2.374	229
2048	8	93.50	5.740	2.410	236
2048	16	74.77	5.780	2.414	228

输入规模	blocksize	kernel	CUDA memcpy HtoD	CUDA memcpy DtoH	cudaMalloc
2048	32	2461	5.747	2.528	227
4096	4	2461	23.45	9.777	230
4096	8	843	23.25	9.826	229
4096	16	647	23.34	9.582	228
4096	32	585	23.54	9.576	234

kernel 执行时间：

即 GPU 上的基础矩阵乘法的计算时间。对于每种输入规模，当 `BlockSize` 大小增加时，内核执行时间明显减少。这是因为，更大的 `BlockSize` 大小可以更好地利用 GPU 的并行处理能力，从而更快地完成计算。

从 Host 和 Device之间的数据复制：

这个时间主要与输入规模有关。输入规模越大，复制的数据就越多，所以时间越长。与此同时，`BlockSize` 大小对这个时间的影响不大，因为无论 `BlockSize` 的大小如何，都需要复制所有的数据。

在 Host 端调用 cudaMalloc：

这是在 CPU 上分配 GPU 内存所需的时间。根据测试结果，cudaMalloc 的时间与输入规模或块大小无关。首先，这个时间肯定与 `BlockSize` 无关，因为只是在 Device 上开辟空间。这个时间与输入规模也没有关系， 这是因为 `cudaMalloc` 操作的主要时间开销来自于在 CPU 和 GPU 之间建立上下文，以及在 GPU 内存管理系统中找到一个足够大的连续内存块。因此， `cudaMalloc` 的执行时间通常被认为是常数，与分配的空间大小无关。

综上，在 GPU 设备允许的范围内，BlockSize 越大，kernel 运行时间越少。

3. 不同的 `gridsize` 和 `BLOCK` 对分块矩阵乘法性能的影响

首先，设置 GPU 上的 Block 大小与其处理的 C 矩阵块大小相同。

```
dim3 blockSize(BLOCK_SIZE, BLOCK_SIZE);
dim3 gridSize((N + blockSize.x - 1) / blockSize.x, (N + blockSize.y - 1) /
blockSize.y);
```

所以 `gridsize` 是由输入规模 `N` 和 `blocksize` 决定的。

由于之前的分析可知：

- 从 Host 和 Device之间的数据复制的时间主要与输入规模有关。
- cudaMalloc 与输入规模或块大小无关.

以下是不同输入规模、不同 `BLOCK` 的 kernel 时间性能（时间单位：毫秒）：

输入规模	BLOCK	kernel
512	4	3.82
512	8	1.16
512	16	0.74
512	32	0.70
1024	4	30.33
1024	8	9.11
1024	16	5.79
1024	32	5.26
2048	4	231.6
2048	8	73.0
2048	16	46.3
2048	32	41.5
4096	4	960
4096	8	377
4096	16	310
4096	32	272
8192	4	8209
8192	8	2709
8192	16	1650
8192	32	1284

kernel 执行时间：

即 GPU 上的分块矩阵乘法的计算时间。对于每种输入规模，当 `BlockSize` 大小增加时，内核执行时间明显减少。这是因为，更大的`BlockSize`大小可以更好地利用 GPU 的并行处理能力，从而更快地完成计算。

四、附录

- `CPU0.c、CPU1.c、CPU2.c、CPU2_1.c`：依次为 CPU 版本的基础矩阵乘法、AVX 矩阵乘法、AVX 分块矩阵乘法（不转置 B 矩阵）、AVX 分块矩阵乘法（转置 B 矩阵）。
- `Makefile`：编译上述代码的 Makefile 文件
- `GPU0.cu、GPU1.cu`：依次为 GPU 版本的基础矩阵乘法、分块矩阵乘法。
- `PB20061343_徐奥_lab5.pdf`：实验报告