计算机体系结构 Lab5

PB20111686 黄瑞轩

1 实验环境

操作系统：Ubuntu 20.04.5 LTS (Focal Fossa)

CPU：Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz

GPU：NVIDIA Tesla V100 SXM2 32GB

2 编译说明

CPU-Phase1（基础矩阵乘法）

```
gcc cpu_phase1.c -o cpu_phase1
./cpu_phase1 6 # 第二个参数指定 N = 1 << 6
```

CPU-Phase2（AVX 矩阵乘法）

```
gcc cpu_phase2.c -o cpu_phase2 -mavx -mavx2 -mfma -msse -msse2 -msse3
./cpu_phase2 6 # 第二个参数指定 N = 1 << 6
```

CPU-Phase3（AVX 分块矩阵乘法）

```
gcc cpu_phase3.c -o cpu_phase3 -mavx -mavx2 -mfma -msse -msse2 -msse3
./cpu_phase3 6 3
# 第二个参数指定 N = 1 << 6
# 第三个参数指定 BLOCK_SIZE = 1 << 3
```

GPU-Phase1（基础矩阵乘法）

```
nvcc gpu_phase1.cu -o gpu_phase1
./gpu_phase1 6 3
# 第二个参数指定 N = 1 << 6
# 第三个参数指定 BLOCK_SIZE = 1 << 3
```

GPU-Phase2（分块矩阵乘法）

```
nvcc gpu_phase2.cu -o gpu_phase2
./gpu_phase2 6
# 第二个参数指定 N = 1 << 6
```

已经提供 Makefile，可以直接使用：

```
make
```

3 CPU 实验

3.1 基础矩阵乘法

基础矩阵乘法核心代码：

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

因为有三重循环，所以时间复杂度为 $O(N^3)$。

使用 $i-k-j$ 的下标顺序比使用 $i-j-k$ 的顺序来说，可以使得内存访问更加连续，减少 Cache Miss。

3.2 AVX 矩阵乘法

AVX 矩阵乘法核心代码：

```
#include <immintrin.h>

void gemm_avx(float *A, float *B, float *C, int N) {
    __m256 a, b, c;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            a = _mm256_set1_ps(A[i * N + j]);
            for (int k = 0; k < N; k += 8) {
                b = _mm256_loadu_ps(&B[j * N + k]);
                c = _mm256_loadu_ps(&C[i * N + k]);
                c = _mm256_fmadd_ps(a, b, c);
                _mm256_storeu_ps(&C[i * N + k], c);
            }
        }
    }
}
```

对于矩阵 A 中的每个元素 $A[i*N+j]$，使用 `_mm256_set1_ps` 函数将其赋值给寄存器 a，然后每次分别加载 8 个浮点数到寄存器 b 和 c 中。接着，使用 `_mm256_fmadd_ps` 函数将 a 和 b 相乘并加上 c，将其存回矩阵 C 第 i 行的对应位置中。

3.3 AVX 分块矩阵乘法

AVX 分块矩阵乘法核心代码：

```
void gemm_avx_block(float *A, float *B, float *C, int si, int sj, int sk, int N, int BLOCK_SIZE) {
    __m256 a, b, c;
    for (int i = si; i < si + BLOCK_SIZE; i++) {
        for (int j = sj; j < sj + BLOCK_SIZE; j++) {
            a = _mm256_set1_ps(A[i * N + j]);
            for (int k = sk; k < sk + BLOCK_SIZE; k += 8) {
                b = _mm256_loadu_ps(&B[j * N + k]);
                c = _mm256_loadu_ps(&C[i * N + k]);
                c = _mm256_fmadd_ps(a, b, c);
                _mm256_storeu_ps(&C[i * N + k], c);
            }
        }
    }
}
```

`BLOCK_SIZE` 即为分块大小。在 main 函数中，如下使用：

```
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)
            gemm_avx_block(A, B, C, i, j, k, N, BLOCK_SIZE);
```

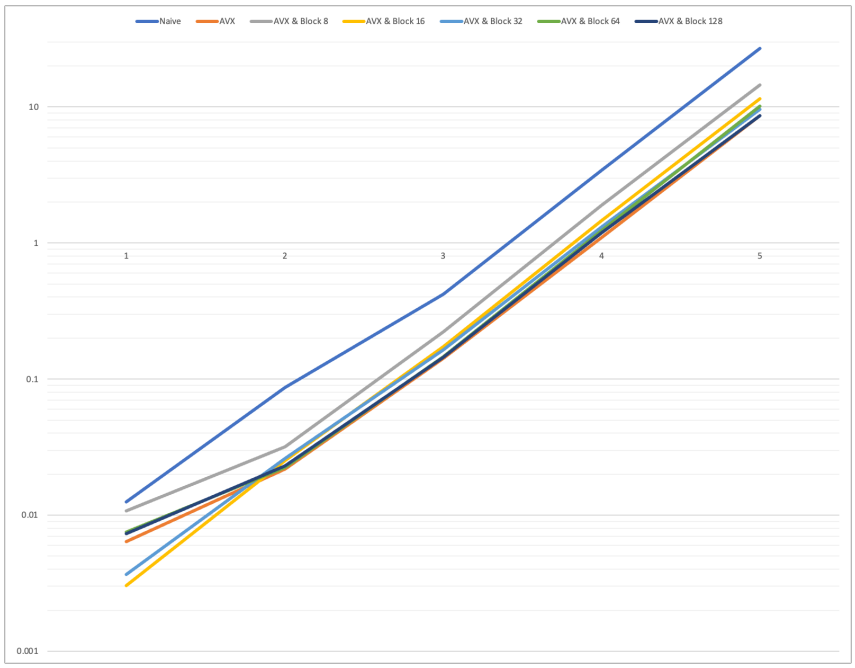
3.4 性能差异分析

正确性验证说明：因为是浮点数，考虑到浮点误差以及浮点交换律不满足，只要对应元素差的绝对值在 $\delta = 10^{-3}$ 以内就认为相等。

对于不同的 N，三种不同实现的性能列表如下，以运行秒记：

N	Naive	AVX	AVX & Block 8	AVX & Block 16	AVX & Block 32	AVX & Block 64	AVX & Block 128
128	0.012575	0.006394	0.010720	0.003025	0.003668	0.007502	0.007336
256	0.087057	0.021843	0.031913	0.025224	0.026226	0.022359	0.023008
512	0.421040	0.142477	0.221955	0.173599	0.164002	0.146463	0.145062
1024	3.427190	1.091758	1.888622	1.460929	1.311071	1.230862	1.191223
2048	26.820708	8.608423	14.514162	11.463270	9.598451	10.079686	8.645347

画成折线图（对数坐标）如下所示：



分析结果如下：

- 在测试范围内，Naive（第一种实现）总是比其他方法要慢，AVX 实现和 AVX 分块实现相对于 Naive 具有更高的性能，尤其是在矩阵规模较大时，它们的性能优势更为明显。这是因为 AVX 指令集可以充分利用 CPU 的向量化计算能力，并且通过分块矩阵乘法可以充分利用 CPU 的缓存，减少了数据的访问次数，提高了计算效率
- BLOCK_SIZE 也会影响性能。如果分块过小，会增加函数调用的开销，从而降低性能；如果分块过大，理论上会接近于不分块的情况，所以 BLOCK_SIZE 大小需要适中
- 其它的矩阵乘法优化手段：
  - OpenMP 并行化：OpenMP是 一种并行编程框架，可以将矩阵乘法等计算密集型任务并行化，从而提高计算效率
  - Cache 优化：矩阵乘法中的数据访问是瓶颈之一，可以通过优化数据布局和访问方式来减少缓存未命中的次数，从而提高计算效率
  - 算法优化：矩阵乘法有多种算法，例如 Strassen 算法、Coppersmith-Winograd 算法等，这些算法在矩阵规模较大时可以提高计算效率

4 GPU 实验

4.1 基础矩阵乘法

4.1.1 核心代码

GPU 上基础矩阵乘法核心代码如下：

```cpp
__device__ int dev_N;

__global__ void gemm_baseline(float *A, float *B, float *C) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < dev_N && j < dev_N ) {
        C[j * dev_N  + i] = 0;
        for (int k = 0; k < dev_N ; k++) {
            C[j * dev_N  + i] += A[j * dev_N  + k] * B[k * dev_N  + i];
        }
    }
}
```

首先计算当前线程的行号 i 和列号 j，由于每个线程处理的数据量较小，可以使用 blockIdx 和 threadIdx 来计算。然后，判断当前线程的行号和列号是否在矩阵 C 的范围内，如果在范围内，则将 `C[j * dev_N + i]` 初始化为 0，然后计算。

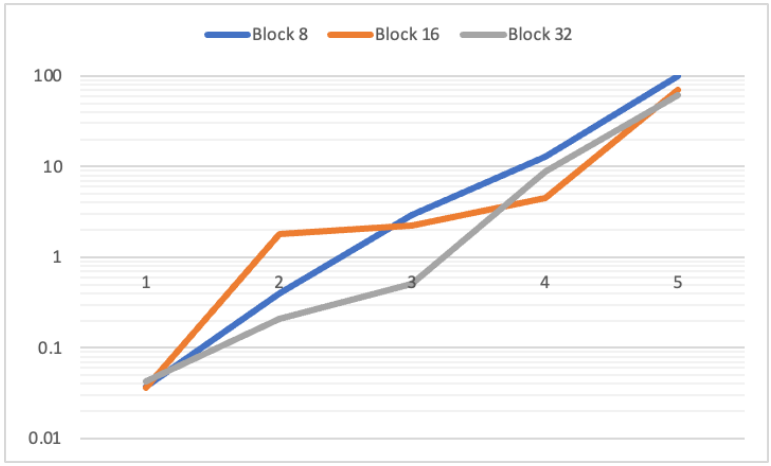
测试时需要注意：线程数不能超过 $1024 = 32 \times 32$，故分块大小不能超过 32。

4.1.2 性能测试

对于不同的 N，不同 BLOCK_SIZE 下的性能列表如下，以运行毫秒记：

N	Block 8	Block 16	Block 32
128	0.037	0.036	0.043
256	0.395	1.819	0.208
512	2.905	2.253	0.510
1024	12.875	4.530	8.835
2048	99.260	70.470	61.883

画成折线图（对数坐标）如下所示：



不同的 gridsize 和 blocksize 对性能的影响：

1. **gridsize：** 当 N 增大时，对于同一个 blocksize，gridsize 也会相应增大。blocksize 相同，运行时间随着 gridsize 的增加而增加。这是因为随着问题规模的扩大，计算量也会相应增加，从而导致运行时间增加。对于较小的 N 值，gridsize 增加的速度较慢，运行时间的增加也较缓慢；而对于较大的 N 值，gridsize 增加的速度较快，运行时间的增加会更明显
2. **blocksize：** 考虑确定的 N 值。在某些情况下，增加 blocksize 可以降低运行时间，例如 $N = 512$ 和 $N = 1024$ 的情况。这是因为较大的 blocksize 可以减少线程之间的通信开销和同步等待时间。然而，在其他情况下，增加 blocksize 反而会导致运行时间增加，例如 $N = 256$ 的情况。这可能是由于计算资源的利用率降低，以及线程束内的分支发散等因素导致的

4.2 分块矩阵乘法

4.2.1 核心代码

这里 BLOCK_SIZE 使用宏定义，因为共享内存大小不能在运行时再确定。

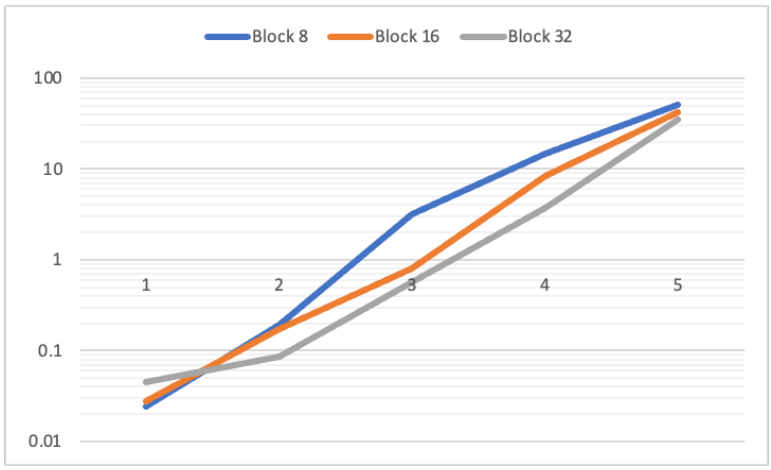
```
#define BLOCK_SIZE 8

__global__ void gemm_baseline(float *A, float *B, float *C) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < dev_N && j < dev_N) {
        C[j * dev_N + i] = 0;
        int b_idx = blockIdx.x * blockDim.x;
        for (int a_idx = blockIdx.y * blockDim.y * dev_N;
             a_idx < blockIdx.y * blockDim.y * dev_N + dev_N - 1; a_idx += blockDim.x, b_idx += blockDim.y * dev_N) {
            __shared__ float Sub_A[BLOCK_SIZE][BLOCK_SIZE];
            __shared__ float Sub_B[BLOCK_SIZE][BLOCK_SIZE];

            Sub_A[threadIdx.y][threadIdx.x] = A[a_idx + threadIdx.y * dev_N + threadIdx.x];
            Sub_B[threadIdx.y][threadIdx.x] = B[b_idx + threadIdx.y * dev_N + threadIdx.x];

            __syncthreads();

            for (int k = 0; k < BLOCK_SIZE; k++) {
                C[j * dev_N + i] += Sub_A[threadIdx.y][k] * Sub_B[k][threadIdx.x];
            }

            __syncthreads();
        }
    }
}
```

4.2.2 性能测试

对于不同的 N，不同 BLOCK_SIZE 下的性能列表如下，以运行毫秒记：

N	Block 8	Block 16	Block 32
128	0.024	0.028	0.045
256	0.194	0.175	0.086
512	3.131	0.797	0.561
1024	14.626	8.412	3.759
2048	50.175	41.699	34.573

画成折线图（对数坐标）如下所示：



不同的 gridsize 和 blocksize 对性能的影响：

1. **gridsize：** 从表中可以看出较小的 gridsize 在较小的矩阵上可能会表现得更好，因为较少的线程块数量可以减少启动和上下文开销。但是在较大的矩阵上，较大的 gridsize 可能会更好，因为它可以更好地利用 GPU 的并行计算能力
2. **blocksize：** 在较小的 N 取值情况下，增加 blocksize 可能会导致运行时间增加，较小的矩阵可能无法充分利用 GPU 的并行计算能力。当 blocksize 增加时，线程块的数量会减少，这会导致 GPU 的并行计算能力无法得到充分利用；在 N 较大时，提高 blocksize 有助于性能提升