Verilog 实验 lab5 实验报告

PB19071405 王昊元

2022 年 05 月 25 日

1 实验目的

- 通过实现矩阵乘法，学习并体会数据级并行的原理及效果

2 实验要求

- 在 CPU 平台实现基础矩阵乘法、AVX 矩阵乘法、AVX 分块矩阵乘法
- 在 GPU 平台实现基础矩阵乘法、分块矩阵乘法

3 实验环境

CPU

- MacBook Air(13-inch, 2017)
- macOS Big Sur 11.2.3
- 1.8 GHz Dual-Core Intel Core i5
- 8 GB 1600 MHz DDR3

GPU

- 学校 GPU 集群

4 实验核心实现

4.1 CPU 部分

基础矩阵乘法 使用三层循环，按照矩阵乘法定义实现即可。

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

```
        {
            // C[i][j]
            int idx = idxs2idx(i, j, N);
            C[idx] = 0.0f;
            for(int k = 0; k < N; k++)
            {
                C[idx] += A[idxs2idx(i, k, N)] * B[idxs2idx(k, j, N)];
            }
        }
    }
    return;
}
```

AVX 矩阵乘法 在实现 AVX 矩阵乘法时，进行了一定的优化，具体算法为：用 A_{ij} 乘 B 的第 j 行的向量，结果加到 C 的第 i 行。

```
void gemm_avx(float *A, float *B, float *C, int N)
{
    __m256 vecA, vecB, vecC;
    // 先假设 n >= 3 即 N >= 8, 向量不需要补0
    // 对于 n < 3的情况, 因为会向量会初始化为0, 所以不需要额外处理
    for(int i = 0; i < N; i++)
    {
        // init Matrix C
        for(int j = 0; j < N; j++)
        {
            // cout << idxs2idx(i, j, N) << endl;
            C[idxs2idx(i, j, N)] = 0.0f;
        }
        for(int j = 0; j < N; j++)
        {
            vecA = _mm256_set1_ps(A[idxs2idx(i, j, N)]);
            // 8 = 256 / 32
            for(int k = 0; k < N; k += 8)
            {
                vecB = _mm256_loadu_ps(B + idxs2idx(j, k, N));
                vecC = _mm256_loadu_ps(C + idxs2idx(i, k, N));
                vecC = _mm256_fmadd_ps(vecA, vecB, vecC);
                _mm256_storeu_ps(C + idxs2idx(i, k, N), vecC);
            }
        }
    }
    return;
}
```

AVX 分块矩阵乘法 在采用上述算法的基础（使用上述算法，可以避免矩阵转置，同时也能兼顾一定的局部性）上，进行矩阵分块运算，同时进行了一定的循环展开，在展开时，考虑数据 IO 的耗时，选择对

A_{ij} 进行展开，可以多次复用 B 的行向量。

```
void block(
    float *A, float *B, float *C,
    int N, int si, int sj, int sk,
    int block_size
)
{
    for(int i = si; i < si + block_size; i += UNROLL)
    {
        for(int j = sj; j < sj + block_size; j++)
        {
            // unroll a
            __m256 a[UNROLL];
            for(int x = 0; x < UNROLL; x++)
            {
                // A 的一列元素各自对应的向量
                a[x] = _mm256_set1_ps(A[idxs2idx(i + x, j, N)]);
            }
            for(int k = sk; k < sk + block_size; k += 8)
            {
                __m256 b;
                // b = [ B[j][k] ... B[j][k+7] ]
                b = _mm256_loadu_ps(B + idxs2idx(j, k, N));
                // unroll c
                __m256 c[UNROLL];
                for(int x = 0; x < UNROLL; x++)
                {
                    c[x] = _mm256_loadu_ps(C + idxs2idx(i + x, k, N));
                    c[x] = _mm256_fmadd_ps(a[x], b, c[x]);
                    _mm256_storeu_ps(C + idxs2idx(i + x, k, N), c[x]);
                }
            }
        }
    }
    return;
}
```

4.2 GPU 部分

基础矩阵乘法

```
__global__ void gemm_baseline(float *A, float *B, float *C, int N)
{
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    if(x >= N || y >= N)
```

```
    {
        return;
    }
    C[x * N + y] = 0.0f;
    float *pa = A + x * N;
    float *pb = B + y;
    for(int i = 0; i < N; i++, pa++, pb += N)
    {
        C[x * N + y] += (*pa) * (*pb);
    }
    return;
}
```

分块矩阵乘法

```
__global__ void blocked_gemm_baseline(float *A, float *B, float *C, int N)
{
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    if(x >= N || y >= N)
    {
        return;
    }

    int tmp_x = threadIdx.x;
    int tmp_y = threadIdx.y;
    int block_num = (N + blockDim.x - 1) / blockDim.x;

    // const int block_size = (1 << 3);
    const int block_size = size;

    __shared__ float blockA[block_size][block_size];
    __shared__ float blockB[block_size][block_size];
    int A_start = blockIdx.x * block_size * N;
    int B_start = blockIdx.y * block_size;
    int A_step = block_size;
    int B_step = block_size * N;

    // 使用tmp减少与数组的交互，提升速度
    // 矩阵规模为 2^13 时，可以从2s+提升到1s+
    float tmp = 0.0f;
    for(int i = 0; i < block_num; i++)
    {
        blockA[tmp_x][tmp_y] = A[A_start + i * A_step + tmp_x * N + tmp_y];
        blockB[tmp_x][tmp_y] = B[B_start + i * B_step + tmp_x * N + tmp_y];
        __syncthreads();
        for(int j = 0; j < blockDim.x; j++)
```

```
        {
            // C[x * N + y] += blockA[tmp_x][j] * blockB[j][tmp_y];
            tmp += blockA[tmp_x][j] * blockB[j][tmp_y];
        }
        __syncthreads();
    }
    C[x * N + y] = tmp;
}
```

5 实验结果及分析

5.1 CPU 部分

结果展示

表 1: CPU 上不同规模的三种矩阵乘法的时间（其中 n 为指数，分块大小为 2^6）

n	基础矩阵乘法/s	AVX 矩阵乘法/s	AVX 分块矩阵乘法/s
0	5e-06	2.1e-05	0.000433
1	3e-06	1.3e-05	0.000293
2	3e-06	1.4e-05	0.000284
3	7e-06	1.2e-05	4.2e-05
4	3.8e-05	1.6e-05	2.9e-05
5	0.000283	2.2e-05	3.5e-05
6	0.002573	6.6e-05	3.4e-05
7	0.018028	0.000239	0.000202
8	0.206072	0.001937	0.001514
9	2.04034	0.018129	0.01266
10	42.2682	0.322516	0.14661

图1仅展示了 n 从 0 到 6 的三种矩阵乘法的时间变化趋势，因为 n 大于 6 时基础矩阵乘法时间较长，会导致图中 n 从 0 到 6 时的时间相对大小不明显。

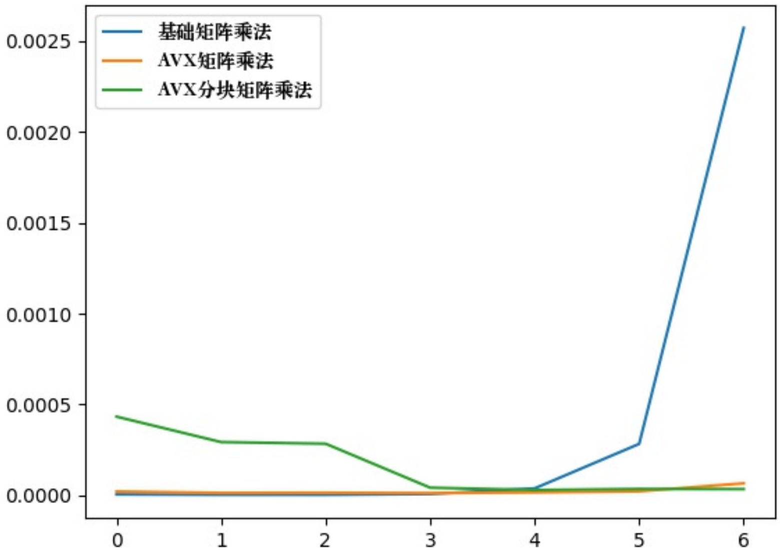

图 1: n 从 0 到 6 的三种矩阵乘法的时间趋势图

表 2: 分块矩阵乘法不同块大小的时间（矩阵规模为 2^8）

块大小	2^3	2^4	2^5	2^6
时间	0.002098s	0.001625s	0.001632s	0.002396s

结果分析

- 从图1中可以看出，
  1. 起初分块矩阵乘法明显慢于其他两种，这是由于分块矩阵每次运算的块的大小是一定的，所以对于较小的矩阵规模会做一些无用的工作。
  2. 在 n 较小时 AVX 矩阵乘法与基础矩阵乘法速度相差不大，且在 n 小于 3（矩阵规模小于 8）时，AVX 矩阵乘法更慢，这是由于 AVX 每次执行是以向量为单位，而我在实现时使用 `__m256` 存储大小为 32bytes 的 float，一个向量存储 8 个单精度浮点数，所以当 n 大于 3 时，能发现 AVX 矩阵乘法明显快于基础矩阵乘法。
  3. 从图中数据发现即使当 n 较大时，分块矩阵乘法与 AVX 矩阵乘法相差无几，这是由于程序中的分块大小恰好为 2^6 （属于无意之举，分析时才发现），从原理上讲确实应相差无几，甚至在 n 为 6 时应该时间几乎一致才对，但比较数据会发现，分块矩阵乘法时间约为 AVX 矩阵乘法的一半，这是由于在分块矩阵乘法中进行了循环展开操作，一定程度上减少了数据 IO 的时间，比较 n 大于 6 时的数据，可以看到分块矩阵明显快于 AVX 矩阵，并且 n 越大效果更加明显。
- 从表2中可以看到，随着分块大小的增大，矩阵乘法先变快后变慢，这可能是由于开始分块大小小于 cache 大小，随着分块越来越大，局部性利用得越来越好，速度越来越快，但慢慢分块大小超过了 cache 大小，计算分块时仍需要不断进行数据 IO，且并行数越来越小，导致速度越来越慢。

- CPU 平台上矩阵乘法的优化手段还有一些优化的细节或技巧，

  1. 三层循环时，j（列索引）的循环放在外面，在矩阵较小时，可以利用 cache 来提高运算性能。
  2. 进行循环展开，同时对元素进行复用，可以大幅减少数据 IO 的次数，从而提高运算效率。
  3. 使用寄存器变量也可以在一定程度上提升矩阵乘法的性能。
  4. 使用指针代替数组索引可以在一定程度上提升性能。

5.2 GPU 部分

结果展示

表 3: GPU 上不同规模的两种矩阵乘法的时间（其中 n 为指数，分块大小为 2^4）

n	基础矩阵乘法/s	分块矩阵乘法/s
6	1.4400e-5	5.5680e-6
7	3.0016e-5	1.1999e-5
8	2.1897e-4	7.4143e-5
9	1.6277e-3	5.5228e-4
10	0.012601	4.2891e-3
11	0.10053	0.034157
12	0.62913	0.27016
13	5.14445	1.64694

表3中，`blocksize` 均为 8，`gridsize` 均为 $\lceil \frac{N}{blocksize} \rceil$。

表 4: GPU 上不同 `gridsize`、`blocksize` 的两种矩阵乘法的时间

blocksize	gridsize	基础矩阵乘法/ms	分块矩阵乘法/ms
2^3	2^7	12.602	4.2901
2^4	2^7	24.327	8.8383
2^5	2^7	53.835	27.590
2^5	2^6	53.780	27.583
2^5	2^5	53.805	27.582

表4中，矩阵规模为 2^{10}，分块大小为 `blocksize`。

结果分析

- 1. 从表3中可以看出，随着矩阵规模增大，矩阵乘法越来越慢，这是由于矩阵规模的增大带来的计算量增大，导致计算开销变多。
  2. 分块矩阵乘法明显快于基础矩阵乘法，这是由于分块矩阵乘法考虑了 Cache，利用了数据的局部性，减少了数据 IO 的时间，从而提高了效率。

- 从表4中可以看出，随着 blocksize 的增大，矩阵乘法速度变慢，这可能是由于块越大，并行度越低，同时每个块的计算时间越长，最后导致矩阵乘法速度越来越慢。
- 从表4中可以看出，gridsize 对矩阵乘法速度影响不大。
- 因为分块矩阵乘法需要约束分块大小与 blocksize 相同，故不单独分析分块大小对矩阵乘法的印象。

6 实验总结

1. 本次实验自己分别实现了 CPU 和 GPU 平台下的基础矩阵乘法和不同优化程度的矩阵乘法，切实体会到了数据级并行的效果。
2. 在实验前的调研阶段让我大开眼界，对矩阵乘法方面的优化有了更深的了解。