Verilog-Lab5 实验报告

彭浩然 **PB19051055**

1 CPU矩阵乘法

1.1 实验环境

计算机：ThinkPax X1 Yoga 4th Gen
CPU：Intel(R) Core(TM) i5-8265U CPU @ 1.60Ghz
CPU核心数：4
内存：8GB
操作系统：WSL Ubuntu 20.04.3 LTS
编译器：gcc (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0
编译选项：-std=c17 -mfma -O2

1.2 算法设计

1.2.1 基础矩阵乘法

实现一个不带优化的三重嵌套循环的矩阵乘法作为正确性验证与性能测量基准线：

$$C_{ij} = \sum_{k=0}^{N-1} A_{ik} B_{kj}$$

```
void gemm_baseline(const float * a, const float * b, float * c)
{
    memset(c, 0, N * N * sizeof(float));
    for (int i = 0; i < N; ++i)
    {
        for (int j = 0; j < N; ++j)
        {
            for (int k = 0; k < N; ++k)
            {
                c[N * i + j] += a[N * i + k] * b[N * k + j];
            }
        }
    }
}
```

其中三个矩阵都采用一维数组的形式表示，所以显式计算了数组偏移量。

1.2.2 AVX矩阵乘法

按照行对矩阵进行分块，可以在矩阵乘法的过程中对B进行连续访存，并借助AVX指令加速：

$$C_i = \sum_{k=0}^{N-1} A_{ik} B_i$$

```
void gemm_avx(const float * a, const float * b, float * c)
{
    memset(c, 0, N * N * sizeof(float));
    for (int i = 0; i < N; ++i)
    {
        for (int j = 0; j < N; ++j)
```

```
        {
            __m256 vecA = _mm256_set1_ps(a[N * i + j]);
            for (int k = 0; k < N; k += 8)
            {
                __m256 vecB = _mm256_loadu_ps(&b[N * j + k]);
                __m256 vecC = _mm256_loadu_ps(&c[N * i + k]);
                vecC = _mm256_fmadd_ps(vecA, vecB, vecC);
                _mm256_storeu_ps(&c[N * i + k], vecC);
            }
        }
    }
}
```

1.2.3 AVX分块矩阵乘法

考虑到访存的局部性，将矩阵C进行分块，按行、列顺序计算每个块内所有元素的结果。块内的计算方式与AVX矩阵乘法相同。其中，`blockL`表示每个子矩阵的行列长度，`blockN`表示原矩阵的每行或者列分为了多少个子矩阵。

```
void gemm_avx_block(const float * a, const float * b, float * c, int blockL)
{
    int blockN = N / blockL;
    memset(c, 0, N * N * sizeof(float));
    for (int blkI = 0; blkI < blockN; ++blkI)
    {
        int blkIBase = blkI * blockL;
        for (int blkJ = 0; blkJ < blockN; ++blkJ)
        {
            int blkJBase = blkJ * blockL;
            for (int blkK = 0; blkK < blockN; ++blkK)
            {
                int blkKBase = blkK * blockL;
                for (int i = blkIBase; i < blkIBase + blockL; ++i)
                {
                    for (int j = blkJBase; j < blkJBase + blockL; ++j)
                    {
                        __m256 vecA = _mm256_set1_ps(a[N * i + j]);
                        for (int k = blkKBase; k < blkKBase + blockL; k += 8)
                        {
                            __m256 vecB = _mm256_loadu_ps(&b[N * j + k]);
                            __m256 vecC = _mm256_loadu_ps(&c[N * i + k]);
                            vecC = _mm256_fmadd_ps(vecA, vecB, vecC);
                            _mm256_storeu_ps(&c[N * i + k], vecC);
                        }
                    }
                }
            }
        }
```

```
    }
}
```

1.3 实验评测

1.3.1 规模影响

在不同规模矩阵下，三种算法的耗时（以毫秒计）如下所示。其中分块矩阵乘法的 `blockL` 固定为 `1 << 6`。

N	基础矩阵乘法	AVX矩阵乘法	AVX分块矩阵乘法
1 << 9	250	16	31
1 << 10	4219	156	188
1 << 11	65078	2172	1734
1 << 12	N/A	18109	13703

在各个规模上，AVX矩阵乘法和AVX分块矩阵乘法相比基础矩阵乘法都有非常明显的加速。虽然每个AVX命令的宽度是256位（即可以同时处理8个 `float` 数据），但是加速比却远远超出了8倍。这可能是因为对C和B进行按行分块后，内存访问的局部性已经优于直接矩阵乘法。

在比较小的规模上，AVX分块矩阵乘法并没有体现出优势，甚至可能因为额外的控制成本而慢于AVX矩阵乘法。在规模大于 `1 << 11` 的样例上，AVX分块矩阵乘法表现出了比较明显的加速效果。 `1 << 10` 规模的矩阵大小为4MiB， `1 << 11` 规模的矩阵大小为16MiB。查询本平台上的cache大小，得到如下结果：

```
LEVEL1_ICACHE_SIZE              32768
LEVEL1_ICACHE_ASSOC             8
LEVEL1_ICACHE_LINESIZE          64
LEVEL1_DCACHE_SIZE              32768
LEVEL1_DCACHE_ASSOC             8
LEVEL1_DCACHE_LINESIZE          64
LEVEL2_CACHE_SIZE               262144
LEVEL2_CACHE_ASSOC              4
LEVEL2_CACHE_LINESIZE           64
LEVEL3_CACHE_SIZE               6291456
LEVEL3_CACHE_ASSOC              12
LEVEL3_CACHE_LINESIZE           64
```

L3 cache的尺寸为6MiB，与两者的数量级接近，这或许可以说明该尺寸以上的AVX矩阵乘法会由于局部性欠佳而更多地直接从内存中读取数字，而分块AVX矩阵乘法则继续将常用的数据控制在了cache范围内。

1.3.2 分块参数影响

在 `1 << 11` 的规模上测试不同分块大小对AVX分块矩阵乘法性能的影响，时间按毫秒计，结果如下：

blockL	AVX分块矩阵乘法
1 << 3	3141
1 << 4	2172
1 << 5	1750
1 << 6	1734
1 << 7	1531
1 << 8	1484
1 << 9	1297
1 << 10	2063
1 << 11	2172

性能整体呈现随着分块大小增加而先提升后降低的趋势。分析原因，较小的分块大小可能会因为分块个数较多而有更多的外层循环控制开销，而过大的分块会使内存访问局部性变差，最终退化为AVX矩阵乘法。性能最佳的分块大小是`1 << 9`，该规模的矩阵占用内存为1MiB，刚好可以在L3 cache中存入6个分块。

1.4 其他优化方法

在CPU平台上，矩阵乘法还有基于并行计算和算法优化的加速方式。

上面的AVX矩阵乘法属于SIMD并行，而基于多线程并行计算的矩阵乘法属于MIMD并行。常见的方法有使用OpenMP将循环进行并行化（如C++线性代数库Eigen就采用了这种方式）或者基于MPI的更细致的任务划分。

之前所有的优化方式并没有减少实际的计算量，与经典矩阵乘法一样，都是$\Theta(N^3)$。而Strassen矩阵乘法通过一些巧妙的分块矩阵代数运算将复杂度降低到$\Theta(N^{\log_2 7}) \approx \Theta(N^{2.807})$。到2020年12月为止，已知的复杂度最低的矩阵乘法算法达到了$O(N^{2.373})$，但该算法由于常数过大而并没有很大的实用价值。矩阵乘法的最优复杂度仍是未知的。

2 GPU矩阵乘法

2.1 实验环境

平台：中国科学技术大学计算机学院科研平台
CPU：Intel(R) Xeon(R) Silver 4210 CPU @ 2.20Ghz
CPU核心数：40
内存：394GiB
操作系统：Ubuntu 18.04.6 LTS
编译器：nvcc 10.2.89
编译选项：-O2

2.2 算法设计

2.2.1 基础GPU矩阵乘法

对于边长为`N`的矩阵，启动`N * N`个线程，分别负责计算目标矩阵的每一个元素。代码示意如下。

```cpp
__global__ void gemm_gpu_impl(float * a, float * b, float * c)
{
    int threadId = blockIdx.x * blockDim.x + threadIdx.x;
    if (threadId >= N * N) return;
    int i = threadId / N;
    int j = threadId % N;

    c[threadId] = 0.0f;
    for (int k = 0; k < N; ++k)
    {
        c[threadId] += a[N * i + k] * b[N * k + j];
    }
}
```

2.2.2 GPU分块矩阵乘法

对于边长为`N`的矩阵，将其划分为边长为`blockL`的若干个子矩阵，并对于每一个子矩阵启动`blockL * blockL`个线程。这些线程先将所需的A和B内相应的部分拷贝进该子矩阵的shared memory，再访问shared memory执行矩阵乘法。这样可以增强访存局部性，减少对global memory的访问，从而大大减少访存时间。

```cpp
template<int blockL>
__global__ void gemm_gpu_block_impl(float * a, float * b, float * c)
{
    int blockN = N / blockL;

    int blkI = blockIdx.y;
    int blkJ = blockIdx.x;
    int subI = threadIdx.y;
    int subJ = threadIdx.x;
    int i = blockL * blkI + subI;
    int j = blockL * blkJ + subJ;
    int cId = N * i + j;
    if (cId >= N * N) return;

    __shared__ float aSub[blockL][blockL];
    __shared__ float bSub[blockL][blockL];

    float localSum = 0.0f;
    for (int blkK = 0; blkK < blockN; ++blkK)
    {
        int kBase = blockL * blkK;

        aSub[subI][subJ] = a[N * i + (kBase + subJ)];
        bSub[subI][subJ] = b[N * (kBase + subI) + j];
        __syncthreads();

        for (int subK = 0; subK < blockL; ++subK)
        {
```

```
            localSum += aSub[subI][subK] * bSub[subK][subJ];
        }
        __syncthreads();
    }

    c[cId] = localSum;
}
```

2.3 实验评测

2.3.1 规模影响

在固定和`blocksize`与`blockL`相等且为`1 << 5`的情况下，针对不同的`N`，比较基础CPU矩阵乘法和上面两种GPU矩阵乘法的耗时。由于有CPU算法参与比较，公平起见，此处的耗时全部通过`clock()`计算而出，而不使用nvprof。时间以毫秒计。

N	基础CPU矩阵乘法	基础GPU矩阵乘法	GPU分块矩阵乘法
1 << 9	475	248	3
1 << 10	3781	248	8
1 << 11	49729	346	25
1 << 12	N/A	1333	124
1 << 13	N/A	7593	741
1 << 14	N/A	57881	5713

整体而言，同规模的矩阵乘法，GPU分块矩阵乘法性能最优，基础GPU矩阵乘法次之，基础CPU矩阵乘法最差。对于CPU矩阵乘法而言，计算复杂度是$O(N^3)$级别的，因此CPU矩阵乘法耗时增长很快。对于两种GPU矩阵乘法，更大的矩阵规模也对应了更多的线程，由于线程数是$O(N^2)$级别的，所以理论上讲时间复杂度是$O(N)$级别的。但由于在GPU上对global memory的访问较慢，而对shared memory的访问局部性较好，基础GPU矩阵乘法的常数项要大于GPU分块矩阵乘法。从较大规模的样例上看，基础GPU矩阵乘法的常数大约是GPU分块矩阵乘法的10倍。

2.3.2 线程分组影响

在固定`N`为`1 << 13`的情况下，比较不同的gridsize和blocksize对基础CPU矩阵乘法的影响。使用nvprof计时，时间单位为毫秒。

blocksize	gridsize	基础GPU矩阵乘法
1 << 1	1 << 12	65443
1 << 2	1 << 11	34350
1 << 3	1 << 10	19655
1 << 4	1 << 9	13560
1 << 5	1 << 8	6919
1 << 6	1 << 7	5776
1 << 7	1 << 6	5652
1 << 8	1 << 5	5651
1 << 9	1 << 4	5625
1 << 10	1 << 3	5505

可以看到随着每个block尺寸的增加，基础GPU矩阵乘法的耗时单调下降。这说明即使是访问global memory，同一个block内不同线程的访问依然比不同block之间的访问局部性更优，能明显减少访存时间。

但是，同一个block内的线程数量并不能无限增大。实验显示在blocksize超过 `1 << 10` 的情况下，运算结果会变成全0矩阵，无法得出正确结果。查阅资料可知，CUDA下计算能力不同的GPU型号允许的最大blocksize可能是512或1024。根据实验结果猜测，该计算平台使用的GPU是计算能力大于等于2的型号的GPU。

2.3.3 矩阵分块影响

在固定 `N` 为 `1 << 13` 且blocksize与 `blockL` 相等的情况下，比较不同的gridsize和blocksize对CPU分块矩阵乘法的影响。使用nvprof计时，时间单位为毫秒。

blocksize = blockL	gridsize	GPU分块矩阵乘法
1 << 1	1 << 12	20123
1 << 2	1 << 11	3802
1 << 3	1 << 10	1287
1 << 4	1 << 9	644
1 << 5	1 << 8	500

可以看到，随着子矩阵的大小增大，GPU分块矩阵乘法的性能单调提升。这可能是因为每个分块更充分地利用了自己的shared memory，从而减少对global memory的访问次数。

然而，shared memory的大小比较有限，继续增大子矩阵的大小会导致运行错误（出现全0矩阵）或者编译失败（编译器提示shared memory使用过多）。如果我们在硬件上拥有更多的shared memory资源，则可以进一步增大分块的大小，获得更好的性能。这个实验反映出来的硬件资源与运行效率的关系是符合常理的。