体系结构Lab5

CPU

baseline核心代码：

C++ 复制代码

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

AVX核心代码：

C++ 复制代码

```cpp
void gemm_avx(float *A, float *B, float *C)
{
    __m256 tmp;
    for (int i = 0; i < N; ++i)
    {
        for (int j = 0; j < N; j += 8)
        {
            tmp = _mm256_loadu_ps(C + i * N + j);
            for (int k = 0; k < N; ++k)
            {
                tmp = _mm256_add_ps(tmp, _mm256_mul_ps(_mm256_broadcast_ss(A + i * N + k), _mm256_loadu_ps(B + k * N + j)));
                _mm256_storeu_ps(C + i * N + j, tmp);
            }
        }
    }
}
```

AVX_BLOCK核心代码：

C++ 复制代码

```cpp
void gemm_avx_one_block(int i0, int j0, int k0, float *A, float *B, float *C)
{
    __m256 tmp;
    for (int i = i0; i < i0 + BLOCKSIZE; ++i)
    {
        for (int j = j0; j < j0 + BLOCKSIZE; j += 8)
        {
            tmp = _mm256_loadu_ps(C + i * N + j); //取出16个float
            for (int k = k0; k < k0 + BLOCKSIZE; ++k)
            {
                tmp = _mm256_add_ps(tmp, _mm256_mul_ps(_mm256_broadcast_ss(A + i * N + k), _mm256_loadu_ps(B + k * N + j)));
            }
            _mm256_storeu_ps(C + i * N + j, tmp);
        }
    }
}

void gemm_avx_block(float *A, float *B, float *C)
{
    for (int i = 0; i < N; i += BLOCKSIZE)
    {
        for (int j = 0; j < N; j += BLOCKSIZE)
        {
            for (int k = 0; k < N; k += BLOCKSIZE)
            {
                gemm_avx_one_block(i, j, k, A, B, C);
            }
        }
    }
}
```

不同规模矩阵结果

三种矩阵乘法实现见 `cpu.c` 源文件。

取 `BLOCKSIZE=1<<3` ,分别取 `N=1<<n,n={6,7,8,9,10}` 三种实现的耗时及优化效率如下：

定义优化效率为 $\eta = \frac{t - t0}{t0} * 100\%$

```
N=64,BLOCKSIZE=8
0.00017
0.00006 65.06%
0.00003 79.52%
N=128,BLOCKSIZE=8
0.00145
0.00036 75.12%
0.00033 77.47%
N=256,BLOCKSIZE=8
0.01446
0.00372 74.30%
0.00341 76.39%
N=512,BLOCKSIZE=8
0.12085
0.03496 71.07%
0.01957 83.81%
N=1024,BLOCKSIZE=8
2.14926
0.53419 75.15%
0.27251 87.32%
```

不难看出AVX和AVX_BLOCK都能显著提高矩阵乘法速率，且AVX_BLOCK比AVX更高效，但提升率并不完全与规模成正比，即N增大，提升率不一定增大。当然，上述数据可能明确说明AVX和AVX_BLOCK能提高效率，至于提升了多少，上面的数值由于存在较大误差，只能作为简单参考。

原因分析

在单次乘法中，AVX能够同时计算8个单精度或4个双精度浮点数，大大减少了计算的时间。因此，使用AVX指令集能够提高单次矩阵乘法的计算性能。

采用分块AVX算法可以进一步提高性能，原因是分块AVX算法将矩阵乘法划分为多个小矩阵，每个小矩阵使用AVX指令集进行计算，这种算法在计算时可以使用CPU缓存，因此可以大大减少访问内存的次数，从而进一步减少运算时间。同时，由于分块操作实现了更好的数据局部性，能够更好的使矩阵缓存，从而能够更快的进行计算，提高矩阵计算的速度。

不同分块大小结果

接下来，取 `N=1<<8` ,分别取 `BLOCKSIZE=1<<b,b={3,4,5,6,7}` ,三种方式的耗时及优化率如下所示：（注意，BLOCKSIZE不能小于8，否则将导致结果计算错误）

理论上，BLOCKSIZE参数只会影响第三种方式（AVX_BLOCK），但从下面的数据可以看出，即使N不变，前两种乘法方式的耗时依旧有变化，这是因为计算机内部的复杂机制造成的无法消除的误差，即使采用多次计算取平均值的做法依然只能小幅减少该误差。

即便有上述所述的误差影响，我们依旧能看出，BLOCKSIZE对乘法计算性能的影响不完全是正相关的。在一定范围内，BLOCKSIZE的增大能提高性能，但当BLOCKSIZE进一步增大时，性能反而降低。就本次实验而言，在 `N=256` 的情况下，当 `BLOCKSIZE=32` 时，程序性能最佳。

C++ | 复制代码

```
N=256,BLOCKSIZE=8
0.01430
0.00374 73.85%
0.00202 85.87%
N=256,BLOCKSIZE=16
0.01447
0.00233 83.93%
0.00169 88.31%
N=256,BLOCKSIZE=32
0.01485
0.00380 74.43%
0.00249 83.23%
N=256,BLOCKSIZE=64
0.01537
0.00226 85.27%
0.00191 87.58%
N=256,BLOCKSIZE=128
0.01517
0.00231 84.78%
0.00236 84.47%
```

更改N的大小，发现在 `N={1<<n,n={8,9,10}` 时，都是取 `BLOCKSIZE=32` 可以取得最佳结果，当N进一步增大时，耗时明显增大，实验成本增加，不在继续。不过目前测试结果已经表明， `BLOCKSIZE=32` 是一个不错的选择。

原因分析

当BLOCKSIZE 较小时，能够利用CPU缓存，减少了访问内存的频率，因此性能有所提高。但是，每个块中的计算量并不是特别稠密，所以它在利用AVX指令时也会浪费一些计算能力。

当BLOCKSIZE 适中时，采用的块的大小适中，可以利用一个256位向量并行计算32个乘积，显著提高了程序并行计算的效率。同时，块的大小也适合CPU缓存，可以更好的利用CPU缓存，从而减少了对内存的访问，同时具有较高的并行度和稠密的计算量，因此效果最好。

当BLOCKSIZE继续增加时，块中的计算量将变得更加稀疏，这样在利用AVX指令时也会浪费一些计算能力，同时由于块的数量减少，程序的并行度也会降低，因此性能并没有得到显著提升。

总之，BLOCKSIZE的取值太大也不能太小，但没有严格的确切数值可以保证BLOCKSIZE在该值时取得最佳结果，实际应用中，应采用实验测试的方式确定BLOCKSIZE的取值。

其它矩阵乘法优化方法

1. 预取数据。通过预取矩阵乘法中需要用到的数据到CPU缓存中，可以减少对内存的频繁访问，从而提高了程序的性能。
2. 循环重排序。通过调整循环顺序来优化空间访问局部性，从而提高效率
3. 循环展开。通过循环展开来加速循环过程。这一内容已在LAB4中详细探究
4. 使用多线程。对于特别大的数组，可以使用多线程计算分别计算各个部分，利用同步技术，互斥锁技术等保证结果正确性即可。

GPU

baseline

核心代码：

Markdown | 复制代码

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

    float sum = 0.f;
    if (idx_x < N && idx_y < N)
    {
        for (int kk = 0; kk < N; ++kk)
        {
            sum += A[idx_y * N + kk] * B[kk * N + idx_x];
        }
        C[idx_y * N + idx_x] = sum;
    }
}
```

取`N={128,256,512,1024},SIZE=32,blockSize=(SIZE,SIZE,1),gridSize=(5,5,1)`时的结果如下：

```
N=128
Time(%)      Total Time   Instances         Average         Minimum
  Maximum  Name

-------  --------------  ----------  --------------  --------------  -----
---------  ------------------------------------------------------------
-----------------
  100.0           14816           1         14816.0           14816
    14816  gemm_baseline

N=256
Time(%)      Total Time   Instances         Average         Minimum
  Maximum  Name

-------  --------------  ----------  --------------  --------------  -----
---------  ------------------------------------------------------------
-----------------
  100.0           29728           1         29728.0           29728
    29728  gemm_baseline
N=512
Time(%)      Total Time   Instances         Average         Minimum
  Maximum  Name

-------  --------------  ----------  --------------  --------------  -----
---------  ------------------------------------------------------------
-----------------
  100.0          218142           1        218142.0          218142
   218142  gemm_baseline
N=1024
Time(%)      Total Time   Instances         Average         Minimum
  Maximum  Name

-------  --------------  ----------  --------------  --------------  -----
---------  ------------------------------------------------------------
-----------------
  100.0         1714327           1       1714327.0         1714327
  1714327  gemm_baseline
```

矩阵规模越大，时间越长，这是显然的。

取 `N=128,SIZE=32,blockSize=(B,B,1),B={8,16,32},gridSize=(5,5,1)`

结果如下：

```
SIZE=32,N=128
blockSIZE=(8,8,1)
Time(%)      Total Time   Instances         Average         Minimum
  Maximum  Name

-------  --------------  ----------  --------------  --------------  -----
---------  ----------------------------------------------------------------
-----------------
  100.0            4128           1          4128.0            4128
     4128  gemm_baseline
SIZE=32,N=128
blockSIZE=(16,16,1)
Time(%)      Total Time   Instances         Average         Minimum
  Maximum  Name

-------  --------------  ----------  --------------  --------------  -----
---------  ----------------------------------------------------------------
-----------------
  100.0            5600           1          5600.0            5600
     5600  gemm_baseline
SIZE=32,N=128
blockSIZE=(32,32,1)
Time(%)      Total Time   Instances         Average         Minimum
  Maximum  Name

-------  --------------  ----------  --------------  --------------  -----
---------  ----------------------------------------------------------------
-----------------
  100.0           14880           1         14880.0           14880
    14880  gemm_baseline

```

取 `blockSize=(8,8,1)` 时效果最佳。值得注意的时，该程序效率影响因子较多，当取不同的N和不同的gridSize时得到的结果不同。上述结果并不能说明在任何情况下都是取(8,8,1)最好。

仅对上述情况进行分析，当blockSize设置较大时，会导致单个线程块中的线程数变多，从而使线程数超过一个流处理器可容纳的最大线程数。这就会导致无法充分利用GPU资源，从而出现效率下降的情况。

取 `N=128,blockSize=(32,32,1),gridSize=(G,G,1),G={6,7,8,9,10,11,12}`

结果如下：

Markdown | 复制代码

```
G=5
Time(%)      Total Time   Instances         Average         Minimum
  Maximum  Name

-------  --------------  ----------  --------------  --------------  -----
---------  ------------------------------------------------------------
-----------------
  100.0           14880           1         14880.0           14880
    14880  gemm_baseline
G=6
Time(%)      Total Time   Instances         Average         Minimum
  Maximum  Name

-------  --------------  ----------  --------------  --------------  -----
---------  ------------------------------------------------------------
-----------------
  100.0           14880           1         14880.0           14880
    14880  gemm_baseline

G=7
Time(%)      Total Time   Instances         Average         Minimum
  Maximum  Name

-------  --------------  ----------  --------------  --------------  -----
---------  ------------------------------------------------------------
-----------------
  100.0           15296           1         15296.0           15296
    15296  gemm_baseline
G=8
Time(%)      Total Time   Instances         Average         Minimum
  Maximum  Name

-------  --------------  ----------  --------------  --------------  -----
---------  ------------------------------------------------------------
-----------------
  100.0           15040           1         15040.0           15040
    15040  gemm_baseline
g=9
Time(%)      Total Time   Instances         Average         Minimum
  Maximum  Name

-------  --------------  ----------  --------------  --------------  -----
---------  ------------------------------------------------------------
-----------------

```

```
  100.0           14944           1         14944.0           14944
    14944  gemm_baseline
G=10
Time(%)      Total Time   Instances         Average         Minimum
  Maximum  Name

-------  --------------  ----------  --------------  --------------  -----
---------  ----------------------------------------------------------------
-----------------
  100.0           14944           1         14944.0           14944
    14944  gemm_baseline
G=11
Time(%)      Total Time   Instances         Average         Minimum
  Maximum  Name

-------  --------------  ----------  --------------  --------------  -----
---------  ----------------------------------------------------------------
-----------------
  100.0           15360           1         15360.0           15360
    15360  gemm_baseline
G=12
Time(%)      Total Time   Instances         Average         Minimum
  Maximum  Name

-------  --------------  ----------  --------------  --------------  -----
---------  ----------------------------------------------------------------
-----------------
  100.0           15455           1         15455.0           15455
    15455  gemm_baseline
```

分析可知，G=5和6时效率最高，而G=9和10时效率相同且高于上下的结果。可以发现，gridSize与效率并非完全正相关。实际使用过程过，应该结合N,SIZE,blockSize等的取值，灵活决定。

block

核心代码：

C++ 复制代码

```cpp
__global__ void gemm_block(float *A, float *B, float *C, int N) {

    __shared__ float s_a[SIZE][SIZE];
    __shared__ float s_b[SIZE][SIZE];

    int idx_x = blockIdx.x * blockDim.x + threadIdx.x;
    int idx_y = blockIdx.y * blockDim.y + threadIdx.y;

    float sum = 0.0;
    for (int bk = 0; bk < N; bk += SIZE) {
        s_a[threadIdx.y][threadIdx.x] = A[idx_y * N + bk + threadIdx.x];
        s_b[threadIdx.y][threadIdx.x] = B[(bk + threadIdx.y) * N + idx_x];
        __syncthreads();

        for (int i = 0; i < SIZE; ++i) {
            sum += s_a[threadIdx.y][i] * s_b[i][threadIdx.x];
        }
        __syncthreads();
    }

    if (idx_x < N && idx_y < N) {
        C[idx_y * N + idx_x] = sum;
    }
}
```

取 `N=512,SIZE={8,16,32},blockSize=(32,32,1),gridSize=(5,5,1)` 时结果如下：
(SIZE>32将无法获得正确结果)

```
SIZE=8
Time(%)      Total Time   Instances         Average         Minimum
  Maximum  Name

-------  --------------  ----------  --------------  --------------  -----
---------  ------------------------------------------------------------
-----------------
  100.0           18592           1         18592.0           18592
    18592  gemm_block
SIZE=16
Time(%)      Total Time   Instances         Average         Minimum
  Maximum  Name

-------  --------------  ----------  --------------  --------------  -----
---------  ------------------------------------------------------------
-----------------
  100.0           17760           1         17760.0           17760
    17760  gemm_block
SIZE=32
Time(%)      Total Time   Instances         Average         Minimum
  Maximum  Name

-------  --------------  ----------  --------------  --------------  -----
---------  ------------------------------------------------------------
-----------------
  100.0           39424           1         39424.0           39424
    39424  gemm_block
```

从上述结果可以发现，分块不能过大，一方面，分块过大可能导致计算速率降低，另一方面，当分块大于32时，计算结果将不在正确。

取 `N=512,SIZE=32,blockSize=(SIZE,SIZE,1),gridSize=(G,G,1),G={4,8,10,12}` 时，结果如下：

```
G=4
Time(%)      Total Time   Instances         Average         Minimum
  Maximum  Name

-------  --------------  ----------  --------------  --------------  -----
---------  ----------------------------------------------------------------
-----------------
  100.0           47211           1         47211.0           47211
    47211  gemm_block
G=8
Time(%)      Total Time   Instances         Average         Minimum
  Maximum  Name

-------  --------------  ----------  --------------  --------------  -----
---------  ----------------------------------------------------------------
-----------------
  100.0           38880           1         38880.0           38880
    38880  gemm_block
G=10
Time(%)      Total Time   Instances         Average         Minimum
  Maximum  Name

-------  --------------  ----------  --------------  --------------  -----
---------  ----------------------------------------------------------------
-----------------
  100.0           39168           1         39168.0           39168
    39168  gemm_block
G=12
Time(%)      Total Time   Instances         Average         Minimum
  Maximum  Name

-------  --------------  ----------  --------------  --------------  -----
---------  ----------------------------------------------------------------
-----------------
  100.0           46431           1         46431.0           46431
    46431  gemm_block

```

较大的gridSize会导致较大的内存带宽负载，从而降低性能。每个线程块都需要访问内存来读取和写入数据，当gridSize变大时，需要访问内存的频率也会随之增加，从而使内存成为性能瓶颈。故实际应用中gridSize不能取得过大。

较小的gridSize会导致无法有效利用GPU资源，同样会导致效率降低。

在上述条件下，gridSize=(8,8,1)时结果最佳。

取 `N=512,SIZE=32,blockSize=(B,B,1),B={4,8,16,32},gridSize=(5,5,1)` 时结果如下：

Markdown | 复制代码

```
B=4
Time(%)      Total Time   Instances         Average         Minimum
  Maximum  Name

-------  --------------  ----------  --------------  --------------  -----
---------  ------------------------------------------------------------
------------------
  100.0            7936           1          7936.0            7936
     7936  gemm_block
B=8
Time(%)      Total Time   Instances         Average         Minimum
  Maximum  Name

-------  --------------  ----------  --------------  --------------  -----
---------  ------------------------------------------------------------
------------------
  100.0            7808           1          7808.0            7808
     7808  gemm_block
B=16
Time(%)      Total Time   Instances         Average         Minimum
  Maximum  Name

-------  --------------  ----------  --------------  --------------  -----
---------  ------------------------------------------------------------
------------------
  100.0           13792           1         13792.0           13792
    13792  gemm_block
B=32
Time(%)      Total Time   Instances         Average         Minimum
  Maximum  Name

-------  --------------  ----------  --------------  --------------  -----
---------  ------------------------------------------------------------
------------------
  100.0           39008           1         39008.0           39008
    39008  gemm_block
```

结果显示，blockSize=(8,8,1)时结果最佳。

blockSize过小，无法有效利用GPU资源，blockSize过大，线程块需要频繁地访问内存，从而导致内存带宽负载较高，使GPU的效率降低。