

浙江大学



课程综合实践 实验报告

实验名称 CUDA 使用基础

姓名学号 陶泓宇 3200103929

实验日期 2021 年 7 月 26 日

目录

1	实验目的和要求	1
2	实验内容和原理	1
2.1	Shared Memory	1
2.2	Blocking	2
3	基准代码	3
4	操作方法和实验数据记录	3
4.1	shared memory 优化	3
4.2	多线程优化	5
4.3	openmp 线程修改优化	7
5	实验结果与分析	9

插图

1	1x4 的 blocking 示意图	2
2	shared memory 优化代码截图	4
3	shared memory 运行截图	5
4	多线程优化代码截图	6
5	多线程优化代码截图	6
6	多线程优化运行结果截图	7
7	优化运行结果截图	8
8	优化运行结果截图	8
9	环境基本信息	9

1 实验目的和要求

卷积 (Convolution) 是一种基本的数学运算, 想必大家在微积分、概率论与数理统计等数学基础课程中都一定程度上接触过。作为一种基本的数学计算, 其在图像处理、机器学习等领域都有重要应用。本次实验需要你使用 CUDA 完成一个 GPU 上的二离散卷积。

2 实验内容和原理

2.1 Shared Memory

GPU 中每个线程对应一个 register, 而且对程序员不可见, 每个 block 对应一个 share memry, 这个由程序操作, 每个网格对应一个 global memory, 也就是说, 所有线程使用同一个 global memroy

thread, block, grid 是 GPU 中线程布局, register, share memory, global memory 是 GPU 中内存布局。block 由若干线程 (thread) 组成, grid 由若干 block 组成, block 是 GPU 并行运算调度的最小单元, 也就是说一个 block 内的所有线程必须同步执行, 而不同 block 之间根据任务不同可以同步可以异步。

在 CUDA 编程中, 如果将一个变量声明为 share 变量, 那么它将被存放在 share memroy 中, 便于一个 block 中线程取数据, 同时减少访问 global memroy 次数, 节省运行时间。

shared memory 使用 "shared" 关键字定义, 这样可以使操作的数据常驻缓存空间; 由于同一个线程块内的线程共享同一个内存区域; 当出现多个线程访问同一个内存 t 空间时, 需要使用 "syncthreads()" 控制线程同步; 所有启动的线程将第一个 syncthreads() 之前的所有代码执行完毕时, 所有线程再继续执行之后的代码

要利用 shared memory 时需要调用的代码如下:

```
1 __shared__ float xxx;  
2 ...  
3 __syncthreads();
```

2.2 Blocking

Blocking 可以增加访存局部性，提高元素的复用，加快矩阵索引速度。
下面是 1x4 的 blocking 的示意图以及 demo 代码

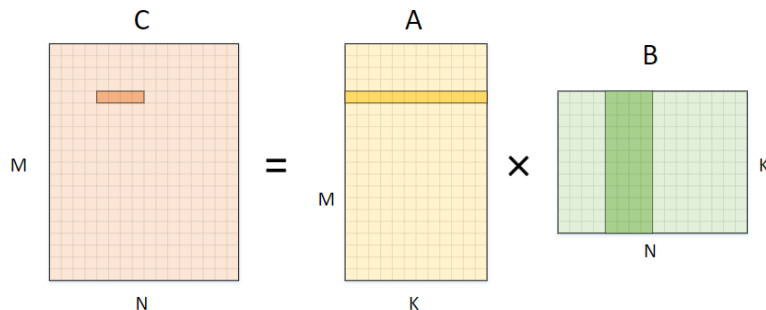


图 1: 1x4 的 blocking 示意图

使用 1x4 的 blocking 的 demo 代码:

```
1  for (int i=0;i<N;i++){
2      for (int j=0;j<N;j+=4) {
3          int sum[4];
4          //clear sum, 使用sum寄存器减小索引时间
5          for(int m=0;m<4;m++){
6              sum[m] = 0;
7          }
8          for (int k=0;k<N;++k){
9              int index = matC[i*N+k];
10             //使用index寄存器减少索引时间
11             for(int x_inner = 0;x_inner<4;x_inner++){
12                 sum[x_inner] += index * matA[(j+x_inner)*N+k];
13             }
14         }
15         for(int fuzhi = 0;fuzhi<4;fuzhi++){
16             matC2[i*N+j+fuzhi] = sum[fuzhi];
17         }
18     }
```

当采用 1x4 的 blocking 时，可以观察到 `matC[i*N+k]` 被复用了四次，由此采用 index 寄存器储存这个值，以减少索引的时间；nxn 时的情况同理。

3 基准代码

需要优化的部分代码如下：

```
1 void Conv(const float *const a, const float *const w, float *const b) {
2     #pragma omp parallel for
3     for (int i = 0; i < kSize; ++i) {
4         for (int j = 0; j < kSize; ++j) {
5             float conv = 0;
6             int x = i - kKernelSize / 2, y = j - kKernelSize / 2;
7             for (int k = 0; k < kKernelSize; ++k) {
8                 for (int l = 0; l < kKernelSize; ++l) {
9                     if (!(x < 0 || x >= kSize || y < 0 || y >= kSize))
10                        conv += a[x * kSize + y] * w[k * kKernelSize + l];
11                 }
12             }
13             x++;
14             y -= kKernelSize;
15         }
16         b[i * kSize + j] = conv;
17     }
18 }
19 }
```

4 操作方法和实验数据记录

4.1 shared memory 优化

尝试使用 shared memory 优化的部分代码如下：

```

__global__
void Conv_GPU(const float *const a, const float *const w, float *const b) {
#pragma omp parallel for
    int index_i = blockIdx.x * blockDim.x + threadIdx.x;
    int index_j = blockIdx.y * blockDim.y + threadIdx.y;

    int gridSize = blockDim.x * blockDim.y;

    __shared__ float b_temp[16][16];

    for(int i = index_i; i < kSize; i += gridSize){
        for(int j = index_j; j < kSize; j += gridSize){
            if(i < kSize && j < kSize){
                b[i * kSize + j] = 0;
                int x = i - kKernelSize / 2;
                int y = j - kKernelSize / 2;

                for (int k = 0; k < kKernelSize; ++k) {
                    for (int l = 0; l < kKernelSize; ++l) {
                        if (!(x < 0 || x >= kSize || y < 0 || y >= kSize)){
                            b_temp[threadIdx.x][threadIdx.y] += a[x * kSize + y] * w[k * kKernelSize + l];
                        }
                        y++;
                    }
                    x++;
                    y -= kKernelSize;
                }

                b[i * kSize + j] = b_temp[threadIdx.x][threadIdx.y];
                b_temp[threadIdx.x][threadIdx.y] = 0;
            }
        }
    }
}

```

图 2: shared memory 优化代码截图

由于我设置的 `threads_per_block` 数量是 16x16, `b_temp` 数组大小也得是 16x16; 在 `b_temp` 数组中每一个元素都被赋值以后再将 `b_temp` 数组中的值赋给 `b` 矩阵, 然后再将 `b_temp` 矩阵归零。

运行结果如下:

```
root@3dclbctkc9t8a-0:/chenjh08/thy_test_1/test2# make
nvcc conv.cu -o conv -O3 -cudart=shared -Xcompiler -fopenmp
root@3dclbctkc9t8a-0:/chenjh08/thy_test_1/test2# ./conv
Correct
421.997 milliseconds
root@3dclbctkc9t8a-0:/chenjh08/thy_test_1/test2# ./conv
Correct
167.482 milliseconds
root@3dclbctkc9t8a-0:/chenjh08/thy_test_1/test2# ./conv
Correct
395.685 milliseconds
root@3dclbctkc9t8a-0:/chenjh08/thy_test_1/test2# ./conv
Correct
355.153 milliseconds
root@3dclbctkc9t8a-0:/chenjh08/thy_test_1/test2# ./conv
Correct
347.589 milliseconds
root@3dclbctkc9t8a-0:/chenjh08/thy_test_1/test2# █
```

图 3: shared memory 运行截图

最快时 167ms 可以运行出来，最慢时 421ms 运行出来，平均运行时间约 300 多 ms，比一开始大概快了十几倍

4.2 多线程优化

优化代码截图如下：

```

__global__
void Conv_GPU(const float *const a, const float *const w, float *const b) {
#pragma omp parallel for
    for (int i = 0; i < kSize; ++i) {
        for (int j = 0; j < kSize; ++j) {
            int i = blockIdx.x * blockDim.x + threadIdx.x;
            int j = blockIdx.y * blockDim.y + threadIdx.y;

            int size = kSize * kSize * sizeof (float);
            int size_w = kKernelSize * kKernelSize * sizeof (float);
            __shared__ float cache[10000];

            if(i < kSize && j < kSize){
                b[i * kSize + j] = 0;
                int x = blockIdx.x * blockDim.x + threadIdx.x - kKernelSize / 2;
                int y = blockIdx.y * blockDim.y + threadIdx.y - kKernelSize / 2;
                for (int k = 0; k < kKernelSize; ++k) {
                    for (int l = 0; l < kKernelSize; ++l) {
                        if (!(x < 0 || x >= kSize || y < 0 || y >= kSize)){
                            b[i * kSize + j] += a[x * kSize + y] * w[k * kKernelSize + l];
                            cache[i * kSize + j] += a[x * kSize + y] * w[k * kKernelSize + l];
                        }
                        __syncthreads();
                        b[i * kSize + j] = cache[i * kSize + j];
                    }
                    y++;
                }
                x++;
                y -= kKernelSize;
            }
        }
    }
}

```

图 4: 多线程优化代码截图

线程的编织方式如下:

```

dim3 threads_per_block (16, 16, 1); // A 16 x 16 block threads=====
dim3 number_of_blocks ((5000 / threads_per_block.x)+1, (5000 / threads_per_block.y)+1, 1);//=

```

图 5: 多线程优化代码截图

运行结果如下图所示:


```

root@3dclbctkc9t8a-0:/chenjh08/thy_test_1/test1# ./conv
Correct
484.135 milliseconds
root@3dclbctkc9t8a-0:/chenjh08/thy_test_1/test1# ./conv
Correct
294.947 milliseconds
root@3dclbctkc9t8a-0:/chenjh08/thy_test_1/test1# ./conv
Correct
395.121 milliseconds
root@3dclbctkc9t8a-0:/chenjh08/thy_test_1/test1# ./conv
Correct
393.293 milliseconds
root@3dclbctkc9t8a-0:/chenjh08/thy_test_1/test1# ./conv
Correct
395.654 milliseconds
root@3dclbctkc9t8a-0:/chenjh08/thy_test_1/test1# ./conv
Correct
387.758 milliseconds
root@3dclbctkc9t8a-0:/chenjh08/thy_test_1/test1# █

```

图 6: 多线程优化运行结果截图

几次测试中，最快 294ms 跑出结果，最慢 484ms 跑出结果，平均约 400ms 运行出结果，比原始代码快了大概十倍。

4.3 openmp 线程修改优化

将原代码中的：

```
1 #pragma omp parallel for
```

改为：

```
1 #pragma omp parallel for num_threads(8)
```

修改线程数后发现运行速度得到了进一步的提高，下图分别为未使用 shared memory 与使用 shared memory 情况下的运行结果：

```

root@3dclbctkc9t8a-0:/tmp/chenjh08/thy_test_1/test1# ./conv
Correct
133.179 milliseconds
root@3dclbctkc9t8a-0:/tmp/chenjh08/thy_test_1/test1# ./conv
Correct
132.733 milliseconds
root@3dclbctkc9t8a-0:/tmp/chenjh08/thy_test_1/test1# ./conv
Correct
136.183 milliseconds
root@3dclbctkc9t8a-0:/tmp/chenjh08/thy_test_1/test1# ./conv
Correct
132.611 milliseconds
root@3dclbctkc9t8a-0:/tmp/chenjh08/thy_test_1/test1# ./conv
Correct
133.042 milliseconds
root@3dclbctkc9t8a-0:/tmp/chenjh08/thy_test_1/test1# █

```

图 7: 优化运行结果截图

```

root@3dclbctkc9t8a-0:/tmp/chenjh08/thy_test_1/test2# ./conv
Correct
115.352 milliseconds
root@3dclbctkc9t8a-0:/tmp/chenjh08/thy_test_1/test2# ./conv
Correct
111.424 milliseconds
root@3dclbctkc9t8a-0:/tmp/chenjh08/thy_test_1/test2# ./conv
Correct
112.891 milliseconds
root@3dclbctkc9t8a-0:/tmp/chenjh08/thy_test_1/test2# ./conv
Correct
111.259 milliseconds
root@3dclbctkc9t8a-0:/tmp/chenjh08/thy_test_1/test2# ./conv
Correct
112.251 milliseconds
root@3dclbctkc9t8a-0:/tmp/chenjh08/thy_test_1/test2# ./conv
Correct
111.63 milliseconds
root@3dclbctkc9t8a-0:/tmp/chenjh08/thy_test_1/test2# ./conv
^[[ACorrect
111.237 milliseconds
root@3dclbctkc9t8a-0:/tmp/chenjh08/thy_test_1/test2# ./conv
Correct
111.165 milliseconds
root@3dclbctkc9t8a-0:/tmp/chenjh08/thy_test_1/test2# █

```

图 8: 优化运行结果截图

可以观察到, 使用 shared memory 后运行速度大约加快了 20ms 左右, 比原始代码大概快了 40 多倍。

5 实验结果与分析

写代码时遇到以下两个奇怪的点：

- 1、使用 shared memory 优化时不同步线程结果也跑得出来。
- 2、使用多线程时，32x32 的线程编织方式比 16x16,8x8 要慢很多；16x16 的与 8x8 的速度差不多；而 4x4 的则与 32x32 的速度差不多。

除此之外，我还发现，使用 shared memory 进行优化后似乎也并没有加速太多。

环境基本信息如下：

环境名称	20210723153248_thy	创建时间	2021-07-23 15:33:11
镜像	10.202.210.100:5000/tensorflow/tensorflow:1.14-cuda10-py36	状态	运行中
资源组	Tra_group_chenjh	GPU	GeForce RTX 2080 Ti:1
CPU	4	挂载路径	/chenjh08
副本数	1	shm_size	TotalMemory/2
数据集路径			

图 9: 环境基本信息