石邢越 16337208

张永东教授

高性能计算程序设计基础

2018 年 12 月 18 日星期二

实验 8: CUDA 优化

1 实验目的

- 尝试对 CUDA 程序进行算法级别的优化。更具体地，对 CUDA 实现的矩阵-向量乘法$\mathbf{A}*\mathbf{b}=\mathbf{c}$尝试使用以下优化方法：
  - 首先对矩阵进行转置操作，从而实现合并访存，以此减少访存的耗时；
  - 在 device 中使用 constant memory 存放向量；
  - 在 device 中使用 shared memory 存放向量和矩阵。
- 在矩阵没有被转置的前提下，尝试借助 shared memory 实现带合并访存的矩阵-向量乘法。

2 实验要求

- 用实验目的中的一种或多种方法优化 CUDA 的矩阵-向量乘法$\mathbf{A}*\mathbf{b}=\mathbf{c}$。其中矩阵$\mathbf{A}$为$10000*10000$的矩阵,$\mathbf{b}$为$10000$维的向量，它们的元素的值由以下公式得到：

$$a_{ij}=i\text{-}0.1*j+1$$
$$b_i=log(sqrt(i*i-i+2))$$

- 考虑如果不转置矩阵，如何借助 shared memory 实现带合并访存的矩阵-向量乘法。

3 算法原理

3.1 CUDA 矩阵-向量乘法的优化

3.1.1 合并访存

合并访存可以更好地利用数据的局部性，我期望达到的效果是“相邻线程访问段对齐的相邻地址”，根据数组在内存中的存储方式，我首先需要将矩阵转置。

3.1.2 使用 constant memory

在线程中计算结果向量中的各个元素时，都需要用到整个向量 **b**，即每个线程都会访问一系列相同的地址（即数组 **b** 所在的地址）。那么可以考虑将向量 **b** 的数据放在 constant memory 中。

如果向量 **b** 过大，在 constant memory 中放不下，则可以分批多次、传输和启动内核。在实际实验中我没有遇到这个问题（10000 个 float 类型数据只需要 40KB 的空间）。

3.1.3 使用 shared memory

可以在使用 constant memory 的尝试上更进一步，使用 shared meory 进一步减少访存时间。对于 block 内线程来说,向量都是共享的,因此我们可以使用比 constant 更快的 shared memory 来存储,此时相比使用 constant，我们免掉了在向量比较大时多次数据拷贝和启动 kernel 的开销,而且没有使用全局变量,代码的可扩展性更好。

3.1.4 借助 shared memory 实现带合并访存的矩阵-向量乘法

结合老师课堂上的讲解，我的理解是如果使用了 shared memory，就没有所谓的“合并访存”了，合并访存应当是针对使用 global memory 的算法而言的（这是因为 SMEM 的读取速度比 GMEM 快得多）。这时我需要考虑的就不

是如何从 shared memory 中合并地读取数据来使用，而是一开始如何合并地将 GMEM 中的数据读取到 SMEM 中备用。

4 实验过程

在这次实验中，我尝试的了以下几种优化：

- 转置矩阵以实现合并访存
- 使用 shared memory 放置矩阵和向量
- 使用编译优化参数-O3

4.1 原始的矩阵-向量乘法

和上一次实验的矩阵乘法基本十分类似，直接将其中一个方阵的 width 改成 1 并修改向量的初始化函数即可。具体代码请查看 codes 文件夹中的“naïve.cu”文件。

4.2 使用矩阵转置实现合并访存

在 naïve 中，计算结果向量 **c** 的过程中有以下代码（代码 1）。为了使相邻线程访问矩阵 A 中相邻的位置，首先让 GPU 中的线程对矩阵进行转置，这部分的代码参考了 NVIDIA 提供的 slides（代码 2）。转置后矩阵 A 中要访问的位置符合合并访存（代码 3）.

```
1  for(int i = 0; i < columnSize; i++){
2        Csub += A[t_id * wA + i] * B[i];
3  }
4  C[t_id] = Csub;
```

代码 1 原始的算法。

第 2 行中，每个线程访问的存储空间是不连续的，需要更多的访存。转置可以解决这个问题。

```
__global__ void transpose(float *odata, float *idata, int width, int height){
    __shared__ float block[(BLOCK_SIZE+1)*BLOCK_SIZE];
    unsigned int xBlock = __mul24(blockDim.x, blockIdx.x);
    unsigned int yBlock = __mul24(blockDim.y, blockIdx.y);
    unsigned int xIndex = xBlock + threadIdx.x;
    unsigned int yIndex = yBlock + threadIdx.y;
    unsigned int index_out, index_transpose;
    if (xIndex < width && yIndex < height){
        unsigned int index_in = __mul24(width, yIndex) + xIndex;
        unsigned int index_block = __mul24(threadIdx.y, BLOCK_SIZE+1) +
threadIdx.x;
        block[index_block] = idata[index_in];
        index_transpose = __mul24(threadIdx.x, BLOCK_SIZE+1) + threadIdx.y;
        index_out = __mul24(height, xBlock + threadIdx.y) + yBlock + threadIdx.x;
    }
    __syncthreads();
    if (xIndex < width && yIndex < height)
        odata[index_out] = block[index_transpose];
}
```

代码 2 矩阵转置

```
      float Csub = 0;
    for(int i = 0; i < columnSize; i++){
            Csub += A[i * wA + t_id] * B[i];
    }
    C[t_id] = Csub;
```

代码 3 转置之后的代码。这样相邻线程访问相邻位置上的 A 矩阵的数据。

4.3 使用 shared memory

在上一次实验中我就尝试了使用 shared memory 的矩阵乘法，和使用没有合并方寸的 global memory 相比得到非常显著的加速效果（图 1）。

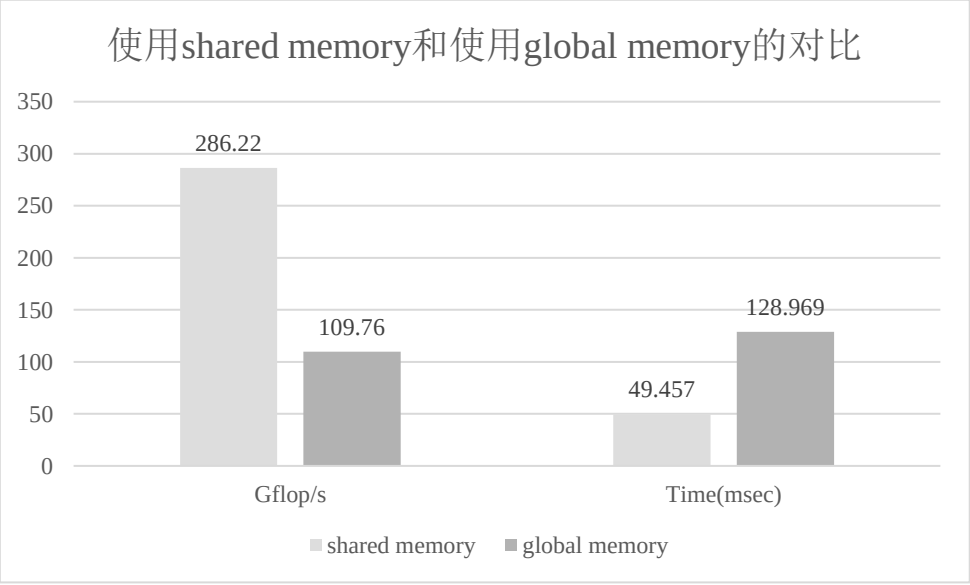

图 1 在实验 7 的矩阵乘法中使用 SMEM 和 GMEM 的计算性能对比

5 实验数据分析与问题讨论

5.1 数据分析

矩阵尺寸	10000*10000
数据类型	float（32 位）
节点数目	1
CPU 核数	56
dimGrid	10
dimBlock	10000

表格 1 运行环境

5.2 实验结果

```
This job is 5986.login@students
GPU running time= 1.554239988327 msec
Total running time: 0.262959000000 sec
```

图 2 naïve 运行结果

```
This job is 5982.login@students
GPU running time for transposition: 7.13411212 msec
GPU running time= 1.451135993004 msec
CPU running time: 0.529044000000 sec
```

图 3 使用转置合并访存的运行结果

5.3 结果分析与总结

- shared memory 可以显著提高性能，但很可能需要解决空间不够的问题
- 我得到的合并访存的优化效果似乎没有老师 slides 上的那么明显（加速比高达 26.1）。可能和矩阵、向量的尺寸不同有关？

  而且如果矩阵本身还没又转置好的话，可以看到转置的代价还是挺高的（见图 3）。
- 用参数-O3 编译优化看起来就像并行程序里的 openmp LOL

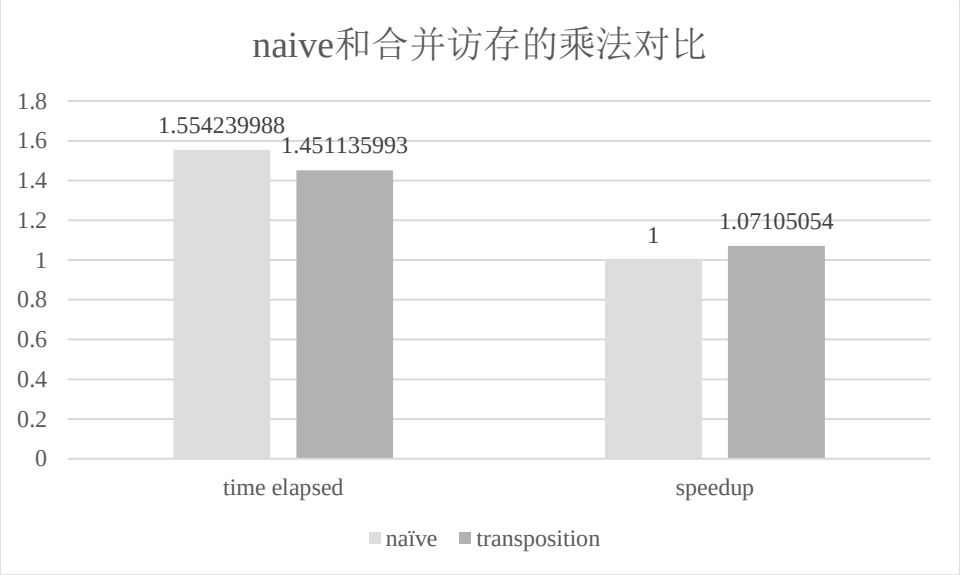

图 4 naive 和转置后合并访存的矩阵-向量乘法的性能对比