

实验六：cuda实验编程

17341046 郭梓煜

实验目的

- 使用下面一种或多种优化方法完成CUDA的矩阵向量乘法 $A * b = C$ ，其中A是 $2^{14} * 2^{14}$ 的方阵，b为 2^{14} 维向量。假设矩阵A的元素为 $a_{ij} = i - 0.1 * j + 1$ ，向量b的元素为 $b_i = \log(\text{sqrt}(i * i - i + 2))$ 。
 - 使用global memory
 - 使用合并访存
 - 使用constant memory存放向量
 - 使用shared memory存放向量和矩阵

实验环境

- 图形化管理文件: winscp
- 编辑器: vscode
- 语言: cuda
- 打开命令行界面: putty
- 编译器: nvcc
- 集群: 222.200.180.115
- source : /public/software/profile.d/cuda10.0.sh

实验代码实现与分析

使用global memory

- 实现涉及函数
 - `__global__` : 核函数限定符
 - `<<< grid, block >>>` : 执行结构参数
 - `cudaMalloc` : 在GPU显存中申请空间
 - `cudaMemcpy` : 将CPU中的数据copy到GPU中
 - `cudaEventRecord`, `cudaEventSynchronize`, `cudaEventRecord`, `cudaEventElapsedTime` : cuda程序计时
 - `cudaFree` : 释放空间
- 实现思路
 - 使用global memory的实现与之前的cuda矩阵加法实验类似，只需要修改其核函数，其他的仍然是从全局调用内存空间，分配到不同线程内部执行不同的任务。
- 主要代码：
 - 核函数
 - 一个线程实现对矩阵一行的乘加操作并将乘加结果累计起来存储于对应的向量C数组中。
 - 根据内置函数`blockIdx`, `blockDim`, `threadIdx`来获取线程所需要处理的矩阵元素再相加

```

1  __global__ void MatMulti(double *A, double *B, double *C, const int
   N)
2  {
3      int row = blockIdx.x * blockDim.x + threadIdx.x;
4      int col = blockIdx.y * blockDim.y + threadIdx.y;
5      double sum = 0.0;
6      if (row < N && col < N && row == col)
7          for (int i = 0; i < N; i++)
8              sum += A[row * N + i] * B[i];
9      C[row] = sum;
10 }

```

◦ host中矩阵申请空间并赋值

- 使用malloc申请即可
- 注意这里A为二维矩阵，仍用一维矩阵来表示

```

1  //cpu initialize matrix
2  double *A = (double *)malloc(N * N * sizeof(double));
3  double *B = (double *)malloc(N * sizeof(double));
4  double *C = (double *)malloc(N * sizeof(double));
5  double *answer = (double *)malloc(N * sizeof(double));
6
7  for (int i = 0; i < N; ++i)
8  {
9      for (int j = 0; j < N; ++j)
10     {
11         A[i * N + j] = i - 0.1 * j + 1;
12     }
13     B[i] = log(sqrt(i * i - i + 2));
14     C[i] = 0.0;
15 }

```

◦ Device中矩阵申请空间并赋值

- 使用cudaMemcpy以及cudaMalloc实现

```

1  // malloc matrix dev_A, dev_B, dev_C at device
2  double *dev_A, *dev_B, *dev_C;
3  cudaMalloc((void **)&dev_A, N * N * sizeof(double));
4  cudaMalloc((void **)&dev_B, N * sizeof(double));
5  cudaMalloc((void **)&dev_C, N * sizeof(double));
6
7  // Memcpy CPU -> GPU
8  cudaMemcpy(dev_A, A, N * N * sizeof(double),
   cudaMemcpyHostToDevice);
9  cudaMemcpy(dev_B, B, N * sizeof(double), cudaMemcpyHostToDevice);
10 cudaMemcpy(dev_C, C, N * sizeof(double), cudaMemcpyHostToDevice);

```

◦ 初始化grid, block

- grid, block的赋值需要考虑线程块的最大线程数。可以写一个测试程序得到最大线程数。具体文件可见代码文件中。

```

1  int N = 16384;
2
3  // Initialize block, grid
4  // Run kernel function
5  dim3 block(32, 32);
6  dim3 grid(N / block.x, N / block.y);
7  MatMulti<<<grid, block>>>(dev_A, dev_B, dev_C, N);

```

○ 从GPU中获取结果

- 同样应用cudaMemcpy即可

```

1  //Memcpy result GPU -> CPU
2  cudaMemcpy(C, dev_C, N * sizeof(double), cudaMemcpyDeviceToHost);

```

○ 检测输出结果正确与否

- 将结果与正确答案比对，正确输出correct，错误输出wrong。

```

1  // Check the result of GPU
2  bool flag = true;
3  for (int i = 0; i < N; ++i)
4  {
5      float a = answer[i];
6      float b = C[i];
7      if (a != b)
8          flag = false;
9  }
10 if (flag)
11     printf("correct\n");
12 else
13     printf("wrong\n");

```

○ 获取CPU计算矩阵的时间

- 使用clock()计时(单位ms)

```

1  clock_t start = clock();
2  ...
3  clock_t end = clock();
4  return (end - start) * 1000 / CLOCKS_PER_SEC;

```

○ 获取GPU计算的时间

- GPU的时间从拷贝矩阵A,B到显存开始至将计算结果复制到host为止
- 利用cudaEventRecord, cudaEventSynchronize, cudaEventRecord, cudaEventElapsedTime对cuda程序计时

```

1  // GPU calculate start
2  cudaEvent_t start, stop;
3  float elapsedTime = 0.0;
4  cudaEventCreate(&start);
5  cudaEventCreate(&stop);
6  cudaEventRecord(start, 0);
7  ...
8  // GPU calculate end

```

```

9  cudaEventRecord(stop, 0);
10 cudaEventSynchronize(stop);
11 cudaEventElapsedTime(&elapsedTime, start, stop);
12
13 // free
14 cudaEventDestroy(start);
15 cudaEventDestroy(stop);

```

使用合并访问

- 实现思路
 - 合并访问是指所有线程访问连续的对齐的内存块，对于L1 cache，内存块大小支持32字节、64字节以及128字节，分别表示线程束中每个线程以一个字节（ $1 \times 32 = 32$ ）、16位（ $2 \times 32 = 64$ ）、32位（ $4 \times 32 = 128$ ）为单位读取数据。前提是，访问必须连续，并且访问的地址是以32字节对齐。（类似于SSE\AVX的向量指令，cuda中的合并访存也是向量指令）
 - 每个 thread 一次读取的内存数据量，可以是 32 bits、64 bits、或 128 bits。不过，32 bits 的效率是最好的。
 - 实现合并访问我们可以定义一个结构体，每次读取32位，即将double类型的四个数合为一组即可,然后改变核函数以及相应的变量类型即可。

- 主要代码
(此处仅展示与使用global memory不一致之处)
 - 定义存储矩阵和向量的结构体

```

1  typedef struct
2  {
3      double A1;
4      double A2;
5      double A3;
6      double A4;
7  } Matrix;

```

- 核函数

```

1  __global__ void MatMulti(Matrix *A, Matrix *B, double *C, const int N)
2  {
3      int row = blockIdx.x * blockDim.x + threadIdx.x;
4      int col = blockIdx.y * blockDim.y + threadIdx.y;
5      double sum = 0.0;
6      if (row < N && col < N && row == col)
7          for (int i = 0; i < N / 4; i++)
8              sum += A[row * N / 4 + i].A1 * B[i * 4].A1 +
9                  A[row * N / 4 + i].A2 * B[i * 4 + 1].A1 +
10                 A[row * N / 4 + i].A3 * B[i * 4 + 2].A1 +
11                 A[row * N / 4 + i].A4 * B[i * 4 + 3].A1;
12      C[row] = sum;
13  }

```

- 矩阵以及向量的赋值较为简单，此处就不再多加展示
- 其他部分与global memory无异

使用constant memory存放向量

- 实现思路

- constant Memory和global Memory一样都位于DRAM，并且有一个独立的on-chip cache，比直接从constant Memory读取要快得多。每个SM上constant Memory cache大小限制为64KB。
- 根据constant memory的大小限制，我将变量类型改成了float，这样就可以保证 2^{14} 向量可以直接存储到constant memory中。
- 主要代码
(此处仅展示与其他代码不一致之处)
 - constant memory声明

```
1 | __constant__ float con_B[16384];
```

- 核函数

```
1 | __global__ void MatMulti(float *A, float *C, const int N)
2 | {
3 |     int row = blockIdx.x * blockDim.x + threadIdx.x;
4 |     if (row < N)
5 |     {
6 |         float sum = 0.0;
7 |         for (int i = 0; i < N; i++)
8 |             sum += A[row * N + i] * con_B[i];
9 |         C[row] = sum;
10 |    }
11 | }
```

- 其余代码与global memory无异

使用shared memory存放向量和矩阵

- 实现思路
 - shared memory (SMEM) 是GPU的重要组成部分之一。当一个block开始执行时，GPU会分配其一定数量的shared memory，shared memory的地址空间会由block中的所有thread 共享。shared memory是划分给SM中驻留的所有block的。
 - 实现矩阵向量乘，访问时多次访问到向量，所以可以在每个block中缓存向量B中来提高计算速度。
 - 事实上，经过我的尝试，集群的shared memory没有办法一次性存储向量B，为了保证结果正确，只好分批次缓存向量B来进行计算。
- 主要代码
(此处仅展示与其他代码不一致之处)
 - share memory声明

```
1 | extern __shared__ double shareB[];
```

- 核函数

```
1 | __global__ void MatMulti(double *A, double *B, double *C, const int N,
2 |   int num)
3 | {
4 |     int row = blockIdx.x * blockDim.x + threadIdx.x;
5 |     int col = blockIdx.y * blockDim.y + threadIdx.y;
6 | }
```

```

7      for (int i = 0; i < 4096; i++)
8          shareB[i] = B[i + num * 4096];
9
10     __syncthreads();
11     double sum = 0.0;
12     if (row < N && col < N && row == col)
13     {
14         for (int i = 0; i < 4096; i++)
15             sum += A[row * N + i + num * 4096] * shareB[i];
16         C[row] += sum;
17     }
18 }

```

- 分多批调用核函数

```

1  for (int i = 0; i < N / 4096; ++i)
2  {
3      cudaMemcpy(dev_C, C, N * sizeof(double), cudaMemcpyHostToDevice);
4      dim3 block(32, 32);
5      dim3 grid(N / block.x, N / block.y);
6      MatMulti<<<grid, block, 4096 * sizeof(double)>>>(dev_A, dev_B,
dev_C, N, i);
7      cudaMemcpy(C, dev_C, N * sizeof(double), cudaMemcpyDeviceToHost);
8  }

```

实验过程

1. 在本机中vscode，cuda环境中编写好代码。

- 编译时使用命令

```

source /public/software/profile.d/cuda10.0.sh
nvcc -g gzy_ex7.cu -o gzy_ex7

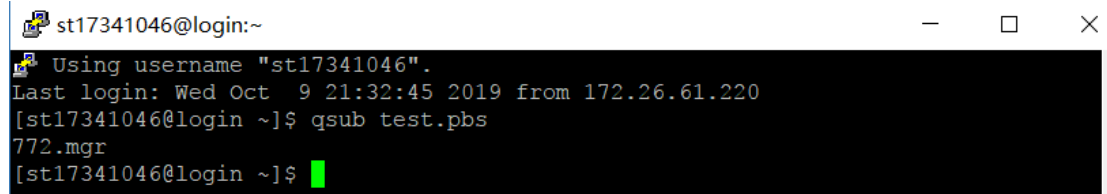
```

2. 利用winscp上传至集群中

3. 打开putty命令行界面，编写test.pbs,使用qsub等命令将作业提交给PBS服务器，进行运行，在自己的文件夹中查看结果，并记录。

具体操作见下图：

- 使用qsub提交作业给服务器

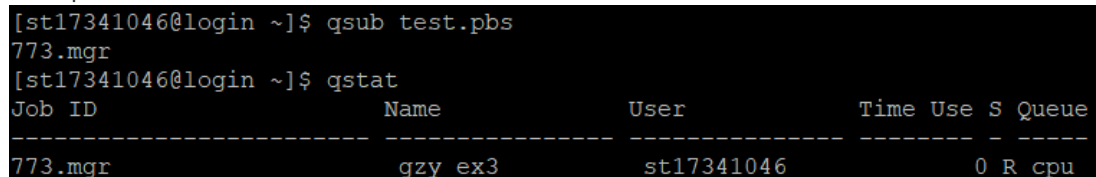


```

st17341046@login:~
Using username "st17341046".
Last login: Wed Oct  9 21:32:45 2019 from 172.26.61.220
[st17341046@login ~]$ qsub test.pbs
772.mgr
[st17341046@login ~]$

```

- 使用qstat查看状态



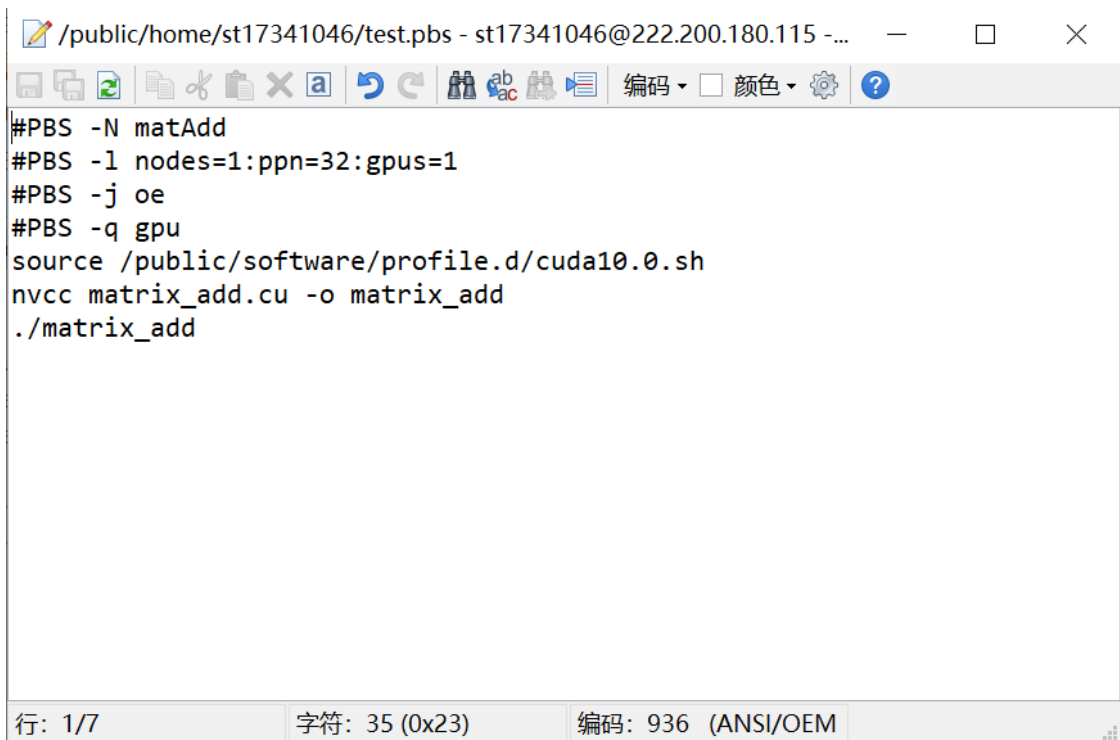
```

[st17341046@login ~]$ qsub test.pbs
773.mgr
[st17341046@login ~]$ qstat
Job ID          Name          User          Time Use S Queue
-----
773.mgr         gzy_ex3       st17341046    0 R  cpu

```

- 编写test.pbs脚本

下图为一个pbs脚本示例：



```
/public/home/st17341046/test.pbs - st17341046@222.200.180.115 - ...
#PBS -N matAdd
#PBS -l nodes=1:ppn=32:gpus=1
#PBS -j oe
#PBS -q gpu
source /public/software/profile.d/cuda10.0.sh
nvcc matrix_add.cu -o matrix_add
./matrix_add
```

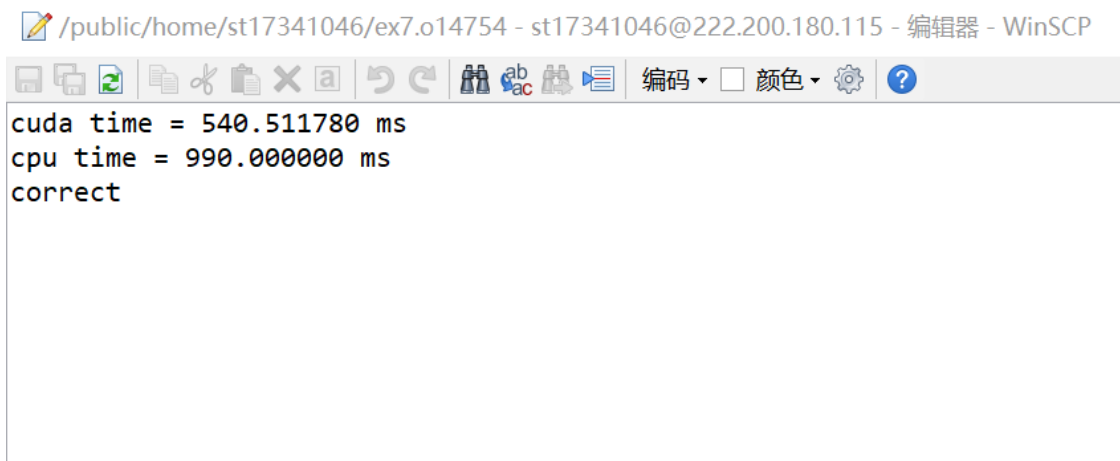
行: 1/7 字符: 35 (0x23) 编码: 936 (ANSI/OEM)

- 得到输出文件
- 打开文件夹中的输出文件并记录数据。
要得到可靠的运行时间数据只需要qsub多次求平均和即可。

实验结果及所得结论

- 使用global memory

输出结果截图如下：

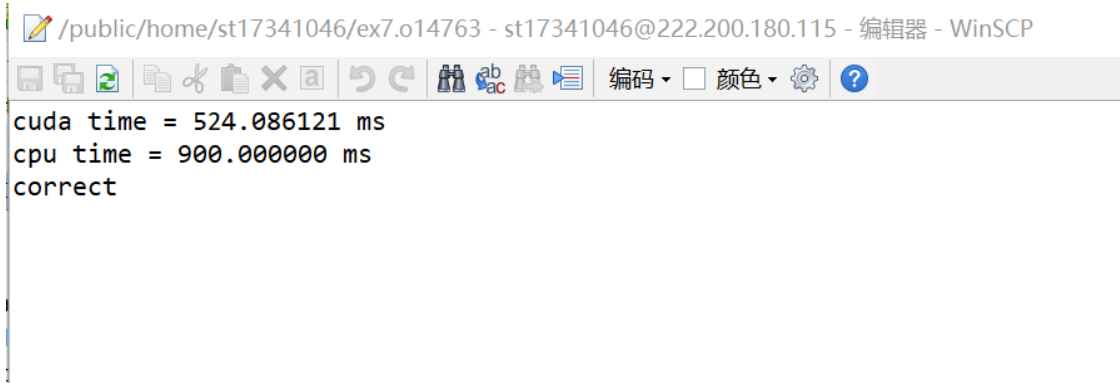


```
/public/home/st17341046/ex7.o14754 - st17341046@222.200.180.115 - 编辑器 - WinSCP
cuda time = 540.511780 ms
cpu time = 990.000000 ms
correct
```

可以看到经过优化的时间比CPU的短一些

- 使用合并访存

输出结果截图如下：



```
/public/home/st17341046/ex7.o14763 - st17341046@222.200.180.115 - 编辑器 - WinSCP
cuda time = 524.086121 ms
cpu time = 900.000000 ms
correct
```

可以看到合并访问比global memory的时间短一些

- 使用constant memory

输出结果截图如下：

```
/public/home/st17341046/ex7.o14762 - st17341046@222.200.180.115 - 编辑器 - WinSCP  
cuda time = 292.325958 ms  
cpu time = 1060.000000 ms  
correct
```

可以看到constant memory比合并访存的时间短一些

- 使用shared memory

输出结果截图如下：

```
/public/home/st17341046/ex7.o14763 - st17341046@222.200.180.115 - 编辑器 - WinSCP  
cuda time = 272.054824 ms  
cpu time = 900.000000 ms  
correct
```

可以看到shared memory比constant memory的时间短一些

- 结论
 - shared memory略快于constant memory快于合并访存 快于 global memory

所遇问题及解决方案

- 在实现constant memory时，一开始使用double类型表示矩阵的每个元素，结果导致不能完全存储，如果非要使用double就得使用分批处理，这样时间又会变长了，所以我选择使用float存储以减少时间。
- 使用shared memory时，由于shared memory大小的限制需要进行分批处理，根据测试得到的shared memory大小， 2^{14} 的向量需要分为4批进行处理。所以调用了四次核函数。

心得体会

这是cuda编程的第二次实验。这次实验在上一次的实验基础上进行扩展，进行矩阵向量乘法，只不过新增的要求是使用多种优化方法，或者说使用不同的存储单元来存储矩阵以及向量。这次的实验让我学习到了多种提高效率的方法，同时也发现了一些问题，在接下来的实验中我会继续努力，做好每一个。