

实验八：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的矩阵乘法 $A * B = C$,其中A, B, C是 $2^{12} * 2^{12}$ 的方阵。矩阵A, B按下面定义元素：
 - $a_{ij} = (i - 0.1 * j + 1) / (i + j + 1)$,
 - $b_{ij} = (j - 0.2 * i + 1) * (i + j + 1) / (i * i + j * j + 1)$
- 使用global memory合并访存
- 采用分块乘法, 使用shared memory
- 请找出最佳的执行配置参数: grid 和 blocks

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合并访存

- 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
 - 与上一次实验使用global memory实现矩阵向量乘类似, 只需修改其中的核函数, 每个线程负责计算结果矩阵的一个元素值即可, 其他的仍然是与之类似, 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A一行以及矩阵B一列的乘加操作并将乘加结果累计起来存储于对应的矩阵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进行乘加操作

```

1  __global__ void MatrixMulKernel(int m, int n, int k, float *A,
   float *B, float *C)
2  {
3      int Row = blockIdx.y * blockDim.y + threadIdx.y;
4      int Col = blockIdx.x * blockDim.x + threadIdx.x;
5
6      if ((Row < m) && (Col < k))
7      {
8          float Cvalue = 0.0;
9          for (int i = 0; i < n; ++i)
10             Cvalue += A[Row * n + i] * B[Col + i * k];
11             C[Row * k + Col] = Cvalue;
12     }
13 }

```

○ host中矩阵申请空间并赋值

- 使用malloc申请即可
- 注意这里的二维矩阵，仍用一维矩阵来表示

```

1  //这里将矩阵按照行优先转换成了一维的形式
2  int m = 4096, n = 4096, k = 4096;
3  float *A = (float *)malloc(m * n * sizeof(float));
4  float *B = (float *)malloc(n * k * sizeof(float));
5  float *C = (float *)malloc(m * k * sizeof(float));
6  float *result = (float *)malloc(m * k * sizeof(float));
7  for (int i = 0; i < m; ++i)
8      for (int j = 0; j < m; ++j)
9      {
10         A[i * m + j] = (i - 0.1 * j + 1) / (i + j + 1);
11         B[i * m + j] = (j - 0.2 * i + 1) * (i + j + 1) / (i * i + j
12 * j + 1);
13         C[i * m + j] = 0.0;
14     }

```

○ Device中矩阵申请空间并赋值

- 使用cudaMemcpy以及cudaMalloc实现

```

1  //分配显存空间
2  int size = sizeof(float);
3  float *d_a;
4  float *d_b;
5  float *d_c;
6  cudaMalloc((void **)&d_a, m * n * size);
7  cudaMalloc((void **)&d_b, n * k * size);
8  cudaMalloc((void **)&d_c, m * k * size);
9  ...
10 //把数据从Host传到Device
11 cudaMemcpy(d_a, A, size * m * n, cudaMemcpyHostToDevice);
12 cudaMemcpy(d_b, B, size * n * k, cudaMemcpyHostToDevice);
13 cudaMemcpy(d_c, C, size * m * k, cudaMemcpyHostToDevice);

```

○ 初始化grid, block, 调用核函数

- 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。

```

1  #define TILE_WIDTH 16
2  //分配网格结构
3  dim3 dimGrid((k - 1) / TILE_WIDTH + 1, (m - 1) / TILE_WIDTH + 1, 1);
4  dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);
5
6  //调用内核函数
7  MatrixMulKernel<<<dimGrid, dimBlock>>>(m, n, k, d_a, d_b, d_c);

```

◦ 从GPU中获取结果

- 同样应用cudaMemcpy即可

```

1  //将结果传回到主机端
2  cudaMemcpy(C, d_c, size * m * k, cudaMemcpyDeviceToHost);

```

◦ 获取CPU计算矩阵乘法的时间

- 使用clock()计时(单位ms)

```

1  //CPU计算正确结果
2  clock_t begin = clock();
3  for (int i = 0; i < m; ++i)
4  {
5      for (int j = 0; j < m; ++j)
6      {
7          float sum = 0;
8          for (int k = 0; k < m; ++k)
9              sum += A[i * m + k] * B[k * m + j];
10         result[i * m + j] = sum;
11     }
12 }
13 clock_t end = clock();
14 cout << "CPU time: " << (end - begin) * 1000 / CLOCKS_PER_SEC << "
    ms" << endl;

```

◦ 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。
- 由于集群上计算存在一定误差，这里当误差在0.001之内就认为结果为正确的。

```

1  // Check the result of GPU
2  //比较结果
3  bool flag = true;
4  for (int i = 0; i < m * k; ++i)
5  {
6      if (abs(result[i] - C[i]) > 0.001)
7      {
8          flag = false;
9          cout << result[i] << "-" << C[i] << endl;
10     }
11 }
12 if (flag)
13     cout << "Check answer: Correct!" << endl;
14 else
15     cout << "Check answer: Error!" << endl;

```

- 获取GPU计算的时间

- GPU的时间从拷贝矩阵A, B, C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 cout << "GPU time: " << elapsedTime << " ms" << endl;
14 // free
15 cudaEventDestroy(start);
16 cudaEventDestroy(stop);
```

使用shared memory分块乘法

- 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。
- 实现矩阵分块乘法, 可以将A,B矩阵瓦片化的结果放入shared memory中, 每个线程加载相应于C元素的A/B矩阵元素, 将A,B子矩阵均加载到shared memory中。
- 然后在shared memory中取值进行乘加操作, 并存储于C矩阵中返回给host。

- 实现涉及函数

- __shared__ : 声明shared memory函数

- 实验代码

(此处展示与global memory不一致之处)

- 核函数

- 使用shared memory即在每个block中将该block相对应的子矩阵块的值存储在声明的shared memory数组中, 在所有元素被加载完毕后(即在此处执行一次同步), 再从shared memory中取值进行乘加操作, 再将结果存入结果C矩阵对应的数组中。
- 代码如下:

```
1 __global__ void MatrixMulKernel(int m, int n, int k, float *A,
2 float *B, float *C)
3 {
4     //申请共享内存, 存在于每个block中
5     __shared__ float ds_A[TILE_WIDTH][TILE_WIDTH];
6     __shared__ float ds_B[TILE_WIDTH][TILE_WIDTH];
7
8     //简化坐标记法, 出现下面6个表示的地方就是并行的地方。
9     int bx = blockIdx.x;
10    int by = blockIdx.y;
11    int tx = threadIdx.x;
12    int ty = threadIdx.y;
```

```

12
13 //确定结果矩阵中的行和列
14 int Row = by * TILE_WIDTH + ty;
15 int Col = bx * TILE_WIDTH + tx;
16
17 //临时变量
18 float Cvalue = 0;
19
20 //循环读入A,B瓦片，计算结果矩阵，分阶段进行计算
21 for (int t = 0; t < (n - 1) / TILE_WIDTH + 1; ++t)
22 {
23     //将A,B矩阵瓦片化的结果放入shared memory中，每个线程加载相应于C元素
    的A/B矩阵元素
24     if (Row < m && t * TILE_WIDTH + tx < n) //越界
    处理，满足任意大小的矩阵相乘
25
26     //ds_A[tx][ty] = A[t*TILE_WIDTH + tx][Row];
    ds_A[tx][ty] = A[Row * n + t * TILE_WIDTH + tx]; //以合
    并的方式加载瓦片
27     else
28         ds_A[tx][ty] = 0.0;
29
30     if (t * TILE_WIDTH + ty < n && Col < k)
31         //ds_B[tx][ty] = B[Col][t*TILE_WIDTH + ty];
32         ds_B[tx][ty] = B[(t * TILE_WIDTH + ty) * k + Col];
33     else
34         ds_B[tx][ty] = 0.0;
35
36     //保证tile中所有的元素被加载
37     __syncthreads();
38
39     for (int i = 0; i < TILE_WIDTH; ++i)
40         Cvalue += ds_A[i][ty] * ds_B[tx][i]; //从shared memory中
    取值
41
42     //确保所有线程完成计算后，进行下一个阶段的计算
43     __syncthreads();
44
45     if (Row < m && Col < k)
46         C[Row * k + Col] = Cvalue;
47 }
48 }

```

找出最佳的执行配置参数

- 实现思路
 - 只需对代码文件中的 `TILE_WIDTH` 进行修改，我将其设置为define值便于修改，分别测试当其为4, 8, 16, 32的情况，通过比较运行时间找到最佳的配置参数。
 - 若强行测试64, 256, 512的情况，将得到错误的结果（全为0），因为一个block的线程是有限的。
- 实验相关代码

```
1 | #define TILE_WIDTH 16
```

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

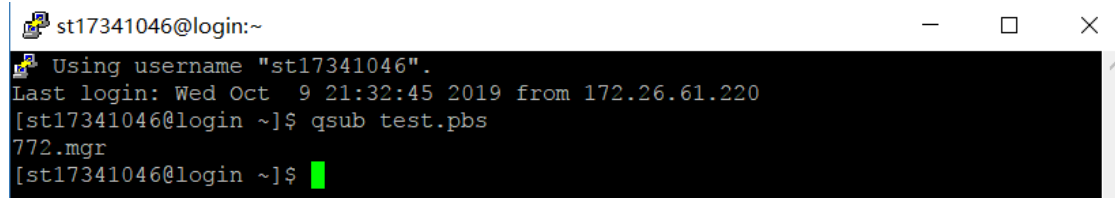
```
source /public/software/profile.d/cuda10.0.sh
nvcc -g gzy_ex8.cu -o gzy_ex8
```

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。

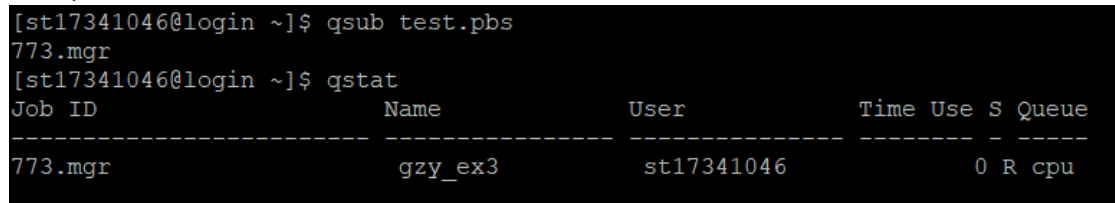
具体操作见下图:

- 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:



```
#PBS -N ex8
#PBS -l nodes=1:ppn=32:gpu=1
#PBS -j oe
#PBS -q gpu

source /public/software/profile.d/cuda10.0.sh
nvcc shared_memory.cu -o shared_memory
./shared_memory
```

- 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。

4. 将代码文件中的 `TILE_WIDTH` 进行修改, 分别测试当其为4, 8, 16, 32的时间, 其值的范围受限于block中的线程数目, 通过多次测试比较时间找到最佳的配置参数。

实验结果及所得结论

- 使用global memory合并访存

输出结果截图如下：

```

/public/home/st17341046/ex8.o15190 - st17341046@222.200.180.115 - 编辑器 - WinSCP
GPU time: 209.144 ms
CPU time: 1413260 ms
Check answer: Correct!

```

可以看到运行时间比CPU运行时间快非常地多，并且经过与CPU计算结果的对比检验，计算结果正确

- 使用shared memory分块乘法

输出结果截图如下：

```

/public/home/st17341046/ex8.o15191 - st17341046@222.200.180.115 - 编辑器 - WinSCP
GPU time: 191.578 ms
CPU time: 1501280 ms
Check answer: Correct!

```

可以看到运行时间比CPU运行时间快非常地多，并且经过与CPU计算结果的对比检验，计算结果正确

- 找到最佳配置参数
 - global memory合并访存

TILE_WIDTH	4	8	16	32
TIME(ms)	291.251	253.342	209.144	184.743

- shared memory分块乘法

TILE_WIDTH	4	8	16	32
TIME(ms)	287.245	262.631	191.578	163.972

- 结论
 - 使用global memory合并访存以及使用shared memory分块乘法对一个 $2^{12} \times 2^{12}$ 的矩阵进行乘法运算，他们的运行时间是大致相等的，但都远远快于CPU时间。
 - 关于最佳参数的问题，通过实验，在矩阵大小为 2^{12} 且GPU资源充足的情况下，随着TILE_WIDTH的增加，运行时间逐步递减。所以当在一个block有限时，为确保能得到正确答案，最佳的参数为TILE_WIDTH为32。

所遇问题及解决方案

- 这一次的实验在上一次实验的基础上来实现，只需进行些许的改动，并不难解决。其中有一些感到困惑的是，在使用shared memory那里，一开始我仅仅将上一次的代码中的向量改为了矩阵，这便遇到了bug，而这一次使用矩阵，进行一个结果矩阵元素的计算时，需要等待其他线程的载入A,B矩阵元素成功，所以需要进行同步等待。加上这一步之后，再与使用CPU计算的结果比较便得到了正确结果。

- 最后，求解最佳的grid，block配置参数。随着block中thread的增加运行时间不断地递减，当然这取决于硬件上的局限。一开始我没有每一个文件都比较答案的正确性，当 `TILE_WIDTH` 为64及以上时，尽管程序可以运行，但是得不到有效的结果。

心得体会

这是cuda编程的第三次实验。这次实验在上一次的实验基础上进行扩展，进行矩阵之间的乘法，新增的要求是找到最佳的grid和block配置参数。这次的实验让我更加深入地学习到了global memory和shared memory的使用方法以及grid和block的配置参数对代码运行效率的影响，同时也发现了一些问题，在接下来的实验中我会继续努力，做好每一次试验。