

实验六：cuda实验编程

17341046 郭梓煜

- 实验六：cuda实验编程
 - 实验目的
 - 实验环境
 - 实验代码实现与分析
 - cuda的 Helloworld 程序
 - cuda 的矩阵加法
 - 核函数
 - host中矩阵申请空间并赋值
 - Device中矩阵申请空间并赋值
 - 初始化grid, block
 - 从GPU中获取结果
 - 检测输出结果正确与否
 - 获取CPU计算矩阵相加的时间
 - 获取GPU计算矩阵相加的时间
 - 实验过程
 - 实验结果及所得结论
 - 所遇问题及解决方案
 - 心得体会

实验目的

- 完成cuda的“Hello world”程序
 - 编译运行grid= (2,4) , block= (8,16) , 给出输出结果文件
- 完成cuda的两个矩阵加法A+B=C
 - 其中A, B是 $2^{13} \times 2^{13}$ 的方阵。
 - 假设矩阵A的元素为 $a_{ij}=i-0.1 \times j+1$, 矩阵B的元素为 $b_{ij}=0.2 \times j-0.1 \times i$ 。
 - 比较cpu计算A+B=C的时间和GPU计算的时间
 - 比较CPU计算结果和GPU计算结果

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

cuda的 Helloworld 程序

- `__global__` : 核函数限定符
- `<<< grid , block >>>` : 执行结构参数
- `cudaDeviceSynchronize()` : 停止CPU端线程的执行, 直到GPU端完成之前CUDA的任务, 包括kernel函数、数据拷贝等。
- 在核函数中输出线程坐标, 便于观察输出数目, 即 (0,0)->(15,63)

```
#include <stdio.h>
__global__ void helloworld(void)
{
    // 二维线程的坐标
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    printf("Hello World from (%d,%d)!\n", i , j);
}
int main()
{
    dim3 grid(2,4);
    dim3 block(8,16);
    helloworld <<<grid,block>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

cuda 的矩阵加法

- `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

核函数

- 将矩阵分块, 分别分配给不同grid的不同block来执行
- 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

```
__global__ void MatAdd(double *A, double *B, double *C, const int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
```

```

        C[i * N + j] = A[i * N + j] + B[i * N + j];
    }

```

host中矩阵申请空间并赋值

- 使用malloc申请即可
- 利用一维数组来表示二维矩阵

```

// malloc matrix a b c at host
double *a = (double *)malloc(N * N * sizeof(double));
double *b = (double *)malloc(N * N * sizeof(double));
double *c = (double *)malloc(N * N * sizeof(double));
for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
    {
        a[i * N + j] = i - 0.1 * j + 1;
        b[i * N + j] = 0.2 * j - 0.1 * i;
        c[i * N + j] = 0;
    }

```

Device中矩阵申请空间并赋值

- 使用cudaMemcpy以及cudaMalloc实现

```

// malloc matrix A B C at device
double *A, *B, *C;
cudaMalloc((void **)&A, N * N * sizeof(double));
cudaMalloc((void **)&B, N * N * sizeof(double));
cudaMalloc((void **)&C, N * N * sizeof(double));

// Memcpy CPU -> GPU
cudaMemcpy(A, a, N * N * sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(B, b, N * 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。

```

int N = 8192;
int WIDTH = 16;

// Initialize block, grid
dim3 block(WIDTH, WIDTH);
dim3 grid(N / block.x, N / block.y);
MatAdd<<<grid, block>>>(A, B, C, N);

```

从GPU中获取结果

- 同样应用cudaMemcpy即可

```
// Memcpy result GPU -> CPU
cudaMemcpy(c, C, N * 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True, 错误输出错误之处。

```
// Check the result of GPU add
void Check_GPU_CPU_result(double *a, double *b, double *c)
{
    bool flag = true;
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
        {
            if (a[i * N + j] + b[i * N + j] != c[i * N + j])
            {
                flag = false;
                printf("Fail at (%d,%d)\n", i, j);
                printf("Correct Answer :%lf , My Answer :%lf\n",
                    (i - 0.1 * j + 1) + (0.2 * j - 0.1 * i), c[i * N + j]);
            }
        }
    if (flag == true)
        printf("GPU Matrix add result: True\n");
}
```

获取CPU计算矩阵相加的时间

- 使用clock()计时(单位ms)

```
double Get_CPU_add_time(double *a, double *b, double *c)
{
    clock_t start = clock();
    for (int i = 0; i < N * N; ++i)
        c[i] = a[i] + b[i];
    clock_t end = clock();
    return (end - start) * 1000 / CLOCKS_PER_SEC;
}
```

获取GPU计算矩阵相加的时间

- 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

```
// GPU calculate start
cudaEvent_t start, stop;
float elapsedTime = 0.0;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);
...
// GPU calculate end
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsedTime, start, stop);

// free
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

实验过程

1. 在本机中vscode, cuda环境中编写好代码。

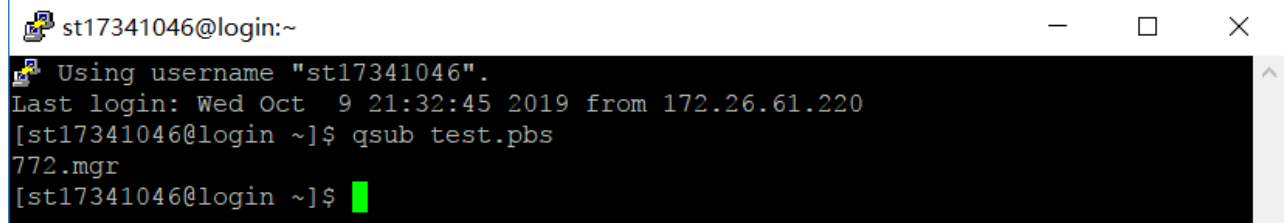
- 编译时使用命令

```
source /public/software/profile.d/cuda10.0.sh
nvcc -g gzy_ex6.cu -o gzy_ex6
```

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。

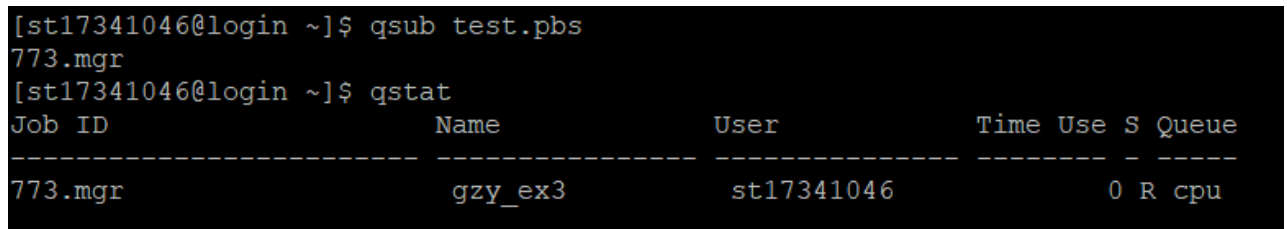
具体操作见下图：

- 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
```

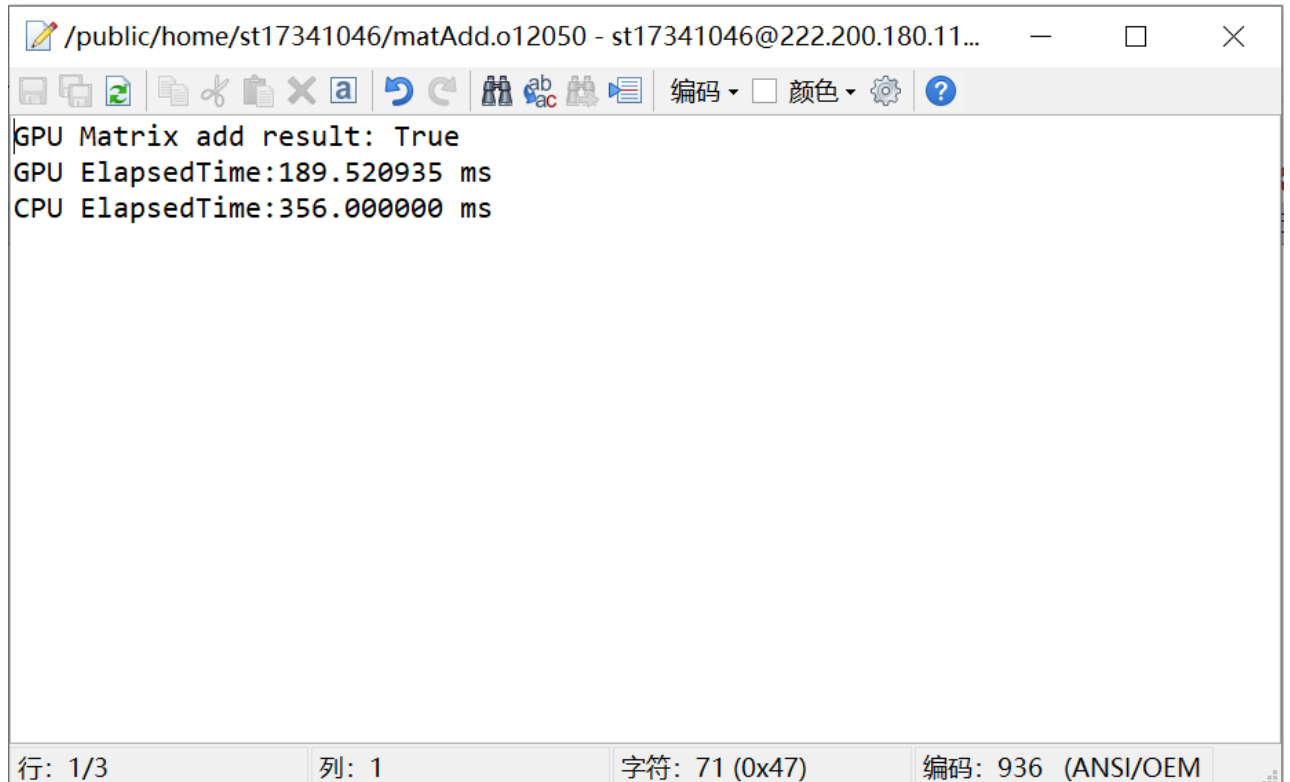
Job ID	Name	User	Time Use	S	Queue
773.mgr	gzy_ex3	st17341046	0 R	cpu	

- 编写test.pbs脚本

下图为一个pbs脚本示例：

- 矩阵加法

输出结果截图如下：



The screenshot shows a terminal window with the following text:

```
/public/home/st17341046/matAdd.o12050 - st17341046@222.200.180.11...  
GPU Matrix add result: True  
GPU ElapsedTime:189.520935 ms  
CPU ElapsedTime:356.000000 ms
```

The terminal window has a standard toolbar at the top and a status bar at the bottom showing "行: 1/3", "列: 1", "字符: 71 (0x47)", and "编码: 936 (ANSI/OEM)".

这只是其中一例，通过多次测量求平均值得到GPU运行时间平均为250.785822 ms，CPU运行平均时间为375.333333 ms。

同时可以看到矩阵相加结果经过检验是正确的。

- 结论

- 在处理的矩阵相对较大的情况下，GPU运行时间比CPU运行时间较短，GPU处理得更快。
- 尽管存在集群波动的影响(甚至会出现GPU运行比CPU慢的情况)，通过多次测试以及到其他环境运行得出的结果，仍然是GPU运行较快。

所遇问题及解决方案

- 在Helloworld程序中，一开始忘记加上cudaDeviceSynchronize()了，也不清楚其具体作用，所以文件经常没有输出。在查阅了相关文件后，了解了其停止CPU端线程的执行，直到GPU端完成之前CUDA的任务的作用，加上该语句后终于得到了Helloworld的结果。
- cudaMalloc只能申请一维数组。一开始我想申请用cudaMalloc申请二维数组，结果出现Segment Fault的错误，查阅之后才知道cudaMalloc一般用来申请一维数组，二维数组一般用cudaMallocPitch来分配存储空间。

心得体会

这是cuda编程的第一次实验。刚开始接触还是有些陌生，但通过老师的讲解以及到网上查阅的函数才开始慢慢入手，发现也不是很难，毕竟是基于C编程上，写起来还是比较简单。希望在以后的实验中也能不断学习，学到更多知识。