并行与分布式计算

第六次作业

姓名：张洪宾
班级：18 级计科超算
学号：18340208

1 问题描述

1.1

Start from the provided skeleton code error-test.cu that provides some convenience macros for error checking. The macros are defined in the header file error_checks_1.h. Add the missing memory allocations and copies and the kernel launch and check that your code works.

1.1.1

What happens if you try to launch kernel with too large block size? When do you catch the error if you remove the cudaDeviceSynchronize() call?

1.1.2

What happens if you try to dereference a pointer to device memory in host code?

1.1.3

What if you try to access host memory from the kernel? Remember that you can use also cuda-memcheck! If you have time, you can also check what happens if you remove all error checks and do the same tests again.

1.2

In this exercise we will implement a Jacobi iteration which is a very simple finite-difference scheme. Familiarize yourself with the provided skeleton. Then implement following things:

1.2.1

Write the missing CUDA kernel sweepGPU that implements the same algorithm as the sweepCPU function. Check that the reported averate difference is in the order of the numerical accuracy.

1.2.2

Experiment with different grid and block sizes and compare the execution times.

2 实验环境

这次实验我使用了老师提供的 jupyterhub 服务器，配置如下：

- 操作系统：Ubuntu 18.04
- GPU 型号: Tesla V100-SXM2-32GB
- NVIDIA (R) Cuda compiler driver

  Copyright (c) 2005-2019 NVIDIA Corporation

  Cuda compilation tools, release 10.1, V10.1.243

3 解决方案和实验结果

3.1 补全 error-test.cu，修改并测试

3.1.1 补全 error-test.cu

我们需要对三个位置进行修改，

- 对 GPU 中的 memory 的申请并将 CPU 中 memory 中的数据拷贝到 GPU
- 调用核函数
- 将 GPU 中 memory 中的结果传回 CPU 的 memory 中去，并释放 GPU 的 memory。

对 GPU 的 memory 进行申请并将 CPU 的 memory 中数据拷贝进去的代码如下：

```
CUDA_CHECK(cudaMalloc((void**)&dA, sizeof(double)*N));
CUDA_CHECK(cudaMalloc((void**)&dB, sizeof(double)*N));
CUDA_CHECK(cudaMalloc((void**)&dC, sizeof(double)*N));
CUDA_CHECK(cudaMemcpy(dA, hA, sizeof(double)*N, cudaMemcpyHostToDevice));
CUDA_CHECK(cudaMemcpy(dB, hB, sizeof(double)*N, cudaMemcpyHostToDevice));
```

调用核函数的代码如下：

```
vector_add<<<1,ThreadsInBlock>>>(dC, dA, dB, N);
```

将 GPU 中 memory 中的结果传回 CPU 的 memory 中去，并释放 GPU 的 memory 的代码如下：

```
CUDA_CHECK(cudaMemcpy(hC, dC, sizeof(double)*N, cudaMemcpyDeviceToHost));
CUDA_CHECK(cudaFree(dA));
CUDA_CHECK(cudaFree(dB));
CUDA_CHECK(cudaFree(dC));
```

所以补全后的代码如下:

```
#include <cstdio>
#include <cmath>
#include "error_checks.h" // Macros CUDA_CHECK and CHECK_ERROR_MSG


__global__ void vector_add(double *C, const double *A, const double *B, int N)
{
    // Add the kernel code
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    // Do not try to access past the allocated memory
    if (idx < N) {
        C[idx] = A[idx] + B[idx];
    }
}


int main(void)
{
    const int N = 20;
    const int ThreadsInBlock = 128;
    double *dA, *dB, *dC;
    double hA[N], hB[N], hC[N];

    for(int i = 0; i < N; ++i) {
        hA[i] = (double) i;
        hB[i] = (double) i * i;
    }

    /*
        Add memory allocations and copies. Wrap your runtime function
        calls with CUDA_CHECK( ) macro
    */
```

```
34      CUDA_CHECK(cudaMalloc((void**)&dA, sizeof(double)*N));
35      CUDA_CHECK(cudaMalloc((void**)&dB, sizeof(double)*N));
36      CUDA_CHECK(cudaMalloc((void**)&dC, sizeof(double)*N));
37
38      // Note the maximum size of threads in a block
39      dim3 grid, threads;
40
41      CUDA_CHECK(cudaMemcpy(dA, hA, sizeof(double)*N, cudaMemcpyHostToDevice));
42      CUDA_CHECK(cudaMemcpy(dB, hB, sizeof(double)*N, cudaMemcpyHostToDevice));
43
44      //// Add the kernel call here
45      vector_add<<<1,ThreadsInBlock>>>(dC, dA, dB, N);
46
47
48      // Here we add an explicit synchronization so that we catch errors
49      // as early as possible. Don't do this in production code!
50      cudaDeviceSynchronize();
51      CHECK_ERROR_MSG("vector_add kernel");
52
53      //// Copy back the results and free the device memory
54      CUDA_CHECK(cudaMemcpy(hC, dC, sizeof(double)*N, cudaMemcpyDeviceToHost));
55
56      CUDA_CHECK(cudaFree(dA));
57      CUDA_CHECK(cudaFree(dB));
58      CUDA_CHECK(cudaFree(dC));
59      for (int i = 0; i < N; i++)
60          printf("%5.1f\n", hC[i]);
61
62      return 0;
63  }
```

在服务器终端中输入如下命令行：

```
nvcc -w error-test.cu -o error-test
./error-test
```

输出的结果如下：

```
jovyan@jupyter-zhanghb55:~/hw6$ vim error-test.cu
jovyan@jupyter-zhanghb55:~/hw6$ nvcc -w error-test.cu -o error-test
jovyan@jupyter-zhanghb55:~/hw6$ ./error-test
  0.0
  2.0
  6.0
 12.0
 20.0
 30.0
 42.0
 56.0
 72.0
 90.0
110.0
132.0
156.0
182.0
210.0
240.0
272.0
306.0
342.0
380.0
```

图 1: 补全 error-test.cu 后的运行结果

可以看到, 根据代码的内容，对每个 $i \in [0, 19]$, 正确计算了每个 $i^2 + i$。

3.1.2 调整块的大小并尝试

通过查询,我发现 GPU 的每个 block 的最大线程数量是 1024。我在这里将 ThreadsInBlock 改成了 12800，然后重新编译运行，结果如下：

```
jovyan@jupyter-zhanghb55:~/hw6$ vim error-test.cu
jovyan@jupyter-zhanghb55:~/hw6$ nvcc -w error-test.cu -o error-test
jovyan@jupyter-zhanghb55:~/hw6$ ./error-test
Error: vector_add kernel at error-test.cu(51): invalid configuration argument
```

图 2: 修改块大小后的运行结果

根据题目要求，注释掉 cudaDeviceSynchronize()，重新编译运行，结果如下：

```
jovyan@jupyter-zhanghb55:~/hw6$ vim error-test.cu
jovyan@jupyter-zhanghb55:~/hw6$ nvcc -w error-test.cu -o error-test
jovyan@jupyter-zhanghb55:~/hw6$ ./error-test
Error: vector_add kernel at error-test.cu(51): invalid configuration argument
```

图 3: 注释掉 cudaDeviceSynchronize() 后的运行结果

发现没有发生变化。根据报错信息，在 51 行捕获错误，于是再去掉 CHECK_ERROR_MSG(”vector_add kernel”); 这行代码。

```
jovyan@jupyter-zhanghb55:~/hw6$ nvcc -w error-test.cu -o error-test
jovyan@jupyter-zhanghb55:~/hw6$ ./error-test
  0.0
  0.0
  0.0
  0.0
  0.0
  0.0
  0.0
  0.0
  0.0
  0.0
  0.0
  0.0
  0.0
  0.0
  0.0
  0.0
  0.0
  0.0
  0.0
```

图 4: 注释掉 CHECK_ERROR_MSG(”vector_add kernel”); 后的运行结果

可以发现并没有进行计算。

通过上网查资料我找到了原因：在前两种情况中，GPU 发现了线程数量过多，抛出了异常并中止运行，然后该异常被 CHECK_ERROR_MSG(”vector_add kernel”) 捕获，所以就退出了程序。

而在第三种情况中，异常未被捕获，而 GPU 也没有运行，所以 CPU 中的 memory 没有变化，但是仍然需要输出，于是就是我们看到的全 0。

3.1.3 在 CPU 的代码中引用指向 GPU 的指针

我在 main 函数加入了一行代码：

```
1 printf("Pointer to device memory: %d",*dA);
```

在这里饮用了 GPU 中的指针 dA。编译并运行，结果如下：

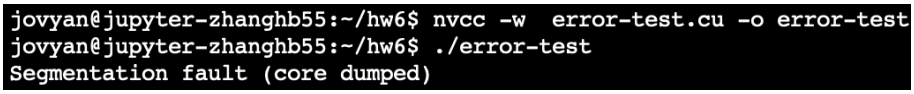

图 5: 在 main 函数饮用 GPU 中指针的结果

发现出现了段错误。因为在 CPU 和 GPU 中的 memory 是不一样的，如果把 GPU 的指针拿到 CPU 中使用，会指向无效的区域或者没有访问权限的区域，就会出现段错误。

3.1.4 在 GPU 的核函数和线程中访问 host 内存

在核函数的参数列表中加入一个 int 型指针，然后让他参与加法运算。然后在 main 函数声明一个 int 型指针，然后让它指向一个一个 int 型整数，该整数赋值为 1。然后将该指针加入调用核函数的参数列表，编译并运行，结果如下：

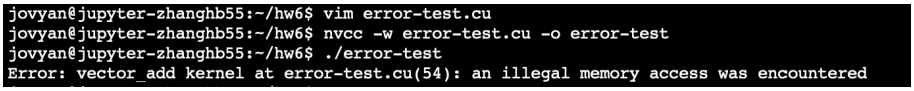

图 6: 在核函数中访问 host 内存

可以看到，提示出现了访问了无效内存。

如果我们把 GPU 异常检测的代码去掉的话，得到如下的结果：

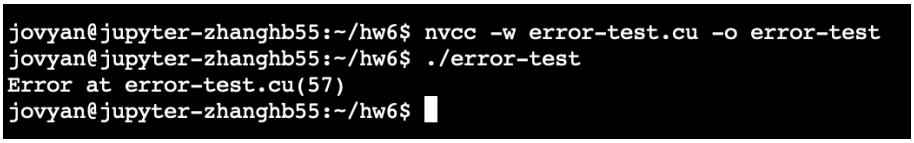

图 7: 去掉异常检测

而代码第 57 行代码如下：

```
1 CUDA_CHECK(cudaMemcpy(hC, dC, sizeof(double)*N, cudaMemcpyDeviceToHost));
```

可以得出：当 GPU 核函数中访问了无效的内存的时候，可能会导致计算出错，然后使得结果无法拷贝回 CPU 的 memory。

3.2 实现 Jacobi 迭代并进行性能测试

3.2.1 补全 Jacobi 迭代的代码

在这里有四个地方需要补充:

- 写出 Jacobi 迭代函数的 GPU 版本代码
- 调用核函数
- 将运算结果拷贝回 CPU 的 memory
- 释放 GPU 资源

写出 Jacobi 迭代函数对 GPU 代码:

```
__global__  void sweepGPU(double *phi, const double *phiPrev, const double *source,
              double h2, int N)
{
    //#error Add here the GPU version of the update routine (see sweepCPU above)
    int i, j;
    int index, i1, i2, i3, i4;

    i = blockIdx.x * blockDim.x + threadIdx.x;
    j = blockIdx.y * blockDim.y + threadIdx.y;

    if(i < N - 1 && j < N - 1 && i > 0 && j > 0) {
        index = i + j*N;
        i1 = (i-1) +   j   * N;
        i2 = (i+1) +   j   * N;
        i3 =   i   + (j-1) * N;
        i4 =   i   + (j+1) * N;
        phi[index] = 0.25 * (phiPrev[i1] + phiPrev[i2] +
                             phiPrev[i3] + phiPrev[i4] -
                             h2 * source[index]);
    }
}
```

调用核函数的代码如下:

```
sweepGPU<<<dimGrid,dimBlock>>>(phiPrev_d, phi_d, source_d, h*h, N);
sweepGPU<<<dimGrid,dimBlock>>>(phi_d, phiPrev_d, source_d, h*h, N);
```

将结果拷贝回 CPU 的 memory：

```
CUDA_CHECK(cudaMemcpy(phi, phi_d, sizeof(double)*N*N, cudaMemcpyDeviceToHost));
CUDA_CHECK(cudaMemcpy(phiPrev,phiPrev_d,sizeof(double)*N*N,cudaMemcpyDeviceToHost));
```

释放 GPU 资源：

```
CUDA_CHECK( cudaFree(phi_d));
CUDA_CHECK( cudaFree(phiPrev_d));
CUDA_CHECK( cudaFree(source_d));
```

然后编译运行，结果如下：

```
jovyan@jupyter-zhanghb55:~/hw6$ vim jacobi.cu
jovyan@jupyter-zhanghb55:~/hw6$ nvcc jacobi.cu -o jacobi
jovyan@jupyter-zhanghb55:~/hw6$ ./jacobi
100 0.00972858
200 0.00479832
300 0.00316256
400 0.00234765
500 0.00186023
600 0.00153621
700 0.0013054
800 0.00113277
900 0.000998881
1000 0.000892078
1100 0.00080496
1200 0.000732594
1300 0.000671564
1400 0.000619434
1500 0.000574415
1600 0.000535167
1700 0.000500665
1800 0.000470113
CPU Jacobi: 3.41855 seconds, 1800 iterations
100 0.00972858
200 0.00479832
300 0.00316256
400 0.00234765
500 0.00186023
600 0.00153621
700 0.0013054
800 0.00113277
900 0.000998881
1000 0.000892078
1100 0.00080496
1200 0.000732594
1300 0.000671564
1400 0.000619434
1500 0.000574415
1600 0.000535167
1700 0.000500665
1800 0.000470113
GPU Jacobi: 0.172135 seconds, 1800 iterations
Average difference is 4.12709
```

图 8: 补全 Jacobi 迭代的代码的运行结果

可以看到，相比 CPU 的运算，GPU 对向量计算的速度快了非常多。

3.2.2 在不同的 grid 和 block 下运算并比较运行速度

在这里的代码中，N 是矩阵的边长，因此总共有 N*N 个元素。核函数的参数 $dimBlock.x = blocksize, dimGrid.x = (N+blocksize-1)/blocksize$，因此一个 block 的线程数量是 $blocksize*blocksize$，一个 grid 的 block 数量是 $(N + blocksize - 1)/blocksize$。

所以我们只需要修改 blocksize 就可以实现对 gird 和 block 的修改。

而我们可以编写程序查看 GPU 的配置：

```
使用GPU device 0: Tesla V100-SXM2-32GB
设备全局内存总量:  32510MB
SM的数量: 80
每个线程块的共享内存大小: 48 KB
每个线程块的最大线程数: 1024
设备上一个线程块（Block）种可用的32位寄存器数量:  65536
每个EM的最大线程数: 2048
每个EM的最大线程束数: 64
设备上多处理器的数量:  80
================================================
使用GPU device 1: Tesla V100-SXM2-32GB
设备全局内存总量:  32510MB
SM的数量: 80
每个线程块的共享内存大小: 48 KB
每个线程块的最大线程数: 1024
设备上一个线程块（Block）种可用的32位寄存器数量:  65536
每个EM的最大线程数: 2048
每个EM的最大线程束数: 64
设备上多处理器的数量:  80
================================================
使用GPU device 2: Tesla V100-SXM2-32GB
设备全局内存总量:  32510MB
SM的数量: 80
每个线程块的共享内存大小: 48 KB
每个线程块的最大线程数: 1024
设备上一个线程块（Block）种可用的32位寄存器数量:  65536
每个EM的最大线程数: 2048
每个EM的最大线程束数: 64
设备上多处理器的数量:  80
================================================
使用GPU device 3: Tesla V100-SXM2-32GB
设备全局内存总量:  32510MB
SM的数量: 80
每个线程块的共享内存大小: 48 KB
每个线程块的最大线程数: 1024
设备上一个线程块（Block）种可用的32位寄存器数量:  65536
每个EM的最大线程数: 2048
每个EM的最大线程束数: 64
设备上多处理器的数量:  80
================================================
```

图 9: GPU 的配置

可以看到，blocksize 最大应该为 32，因为 $32^2 = 1024$，再大就超过了最大线程数了。

所以我让 blocksize 分别为 1，2，4，8，16，32, 分别运行测试速度，结果如下：

blocksize	一个块中的线程数量	grid 的块的数量	运行时间
1	1	512^2	1.14019s
2	4	256^2	0.310495s
4	16	128^2	0.071512s
8	64	64^2	0.172135s
16	256	32^2	0.02601s
32	1024	16^2	0.018792s

可以看到，运行的时间有波动但总体上来说，让每个块但线程越多越好。

4 总结和遇到的问题

这次作业让我接触了一点 CUDA 编程，对 GPU 的特点有了更加深刻的理解。
遇到的问题有：

- 配置环境的时候出现了问题，于是不得不采用老师的服务器。
- 将 host 的数组作为参数传给核函数，结果发生错误。

通过这次练习也加深了我对 GPU 的并行计算的理解，收获颇丰。