并行与分布式作业

"CUDA 编程"

第六次作业

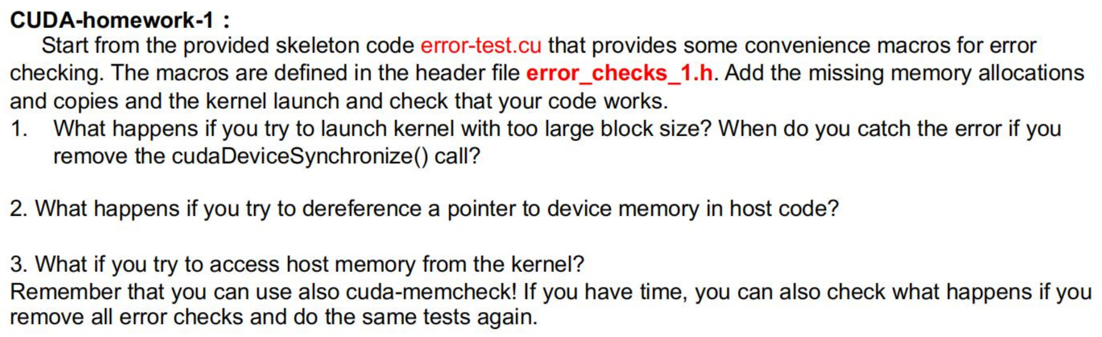
姓名：TRY

班级：18 级计科 7 班

学号：

一、 问题描述

1.基于 error-test.cu 的测试：



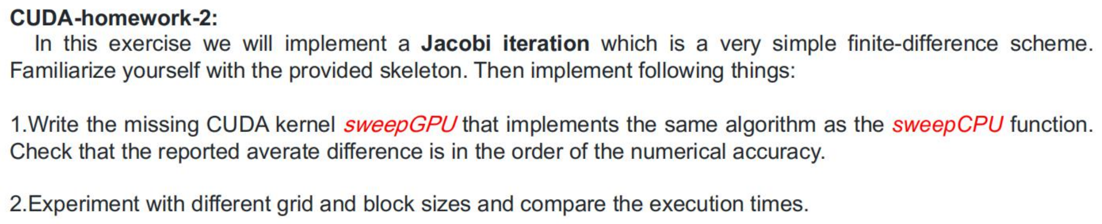
CUDA-homework-1 :
Start from the provided skeleton code error-test.cu that provides some convenience macros for error checking. The macros are defined in the header file **error_checks_1.h**. Add the missing memory allocations and copies and the kernel launch and check that your code works.

1. 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?

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

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.

2. Jacobi 迭代的实现



CUDA-homework-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.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.

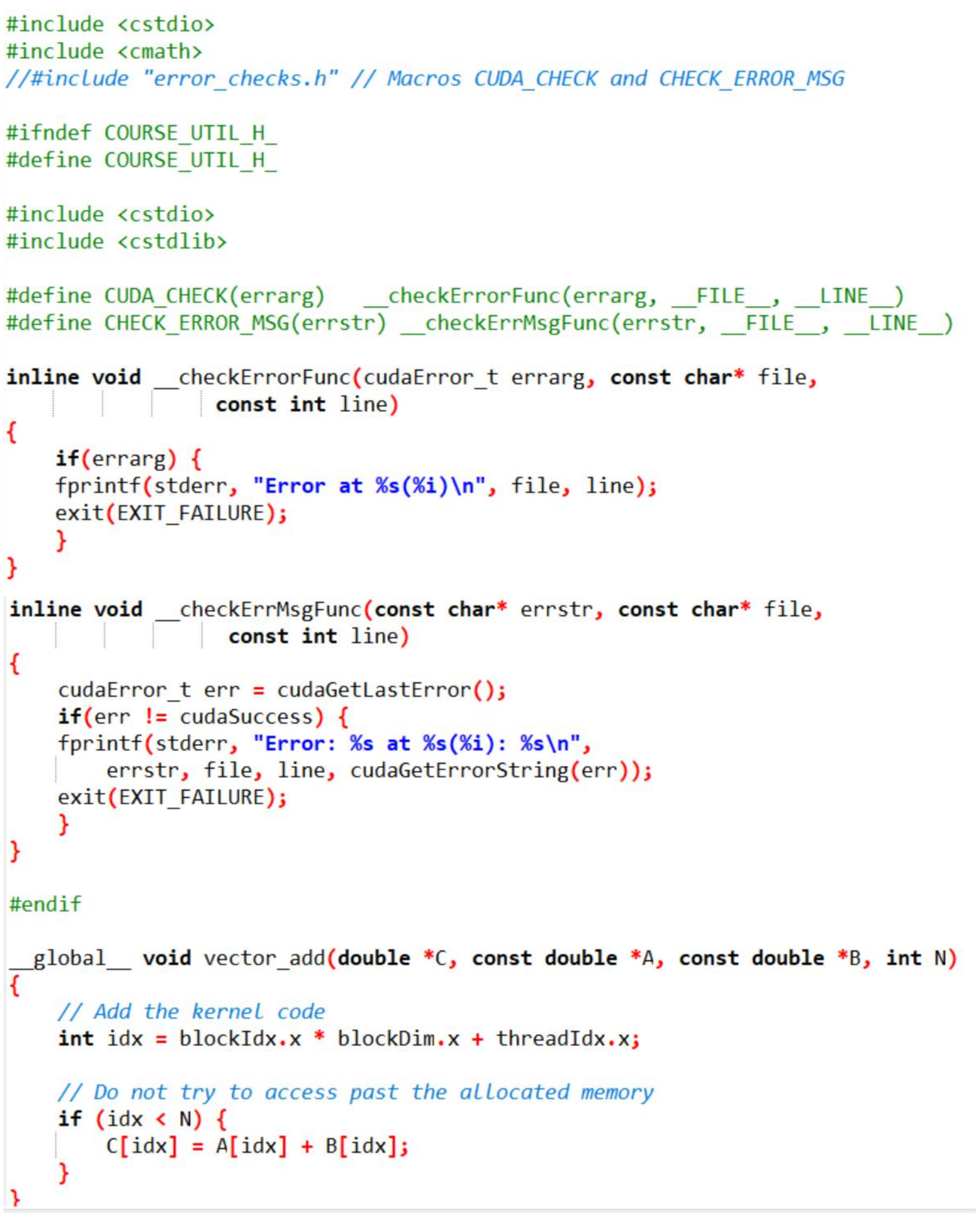
2.Experiment with different grid and block sizes and compare the execution times.

二、 解决方案

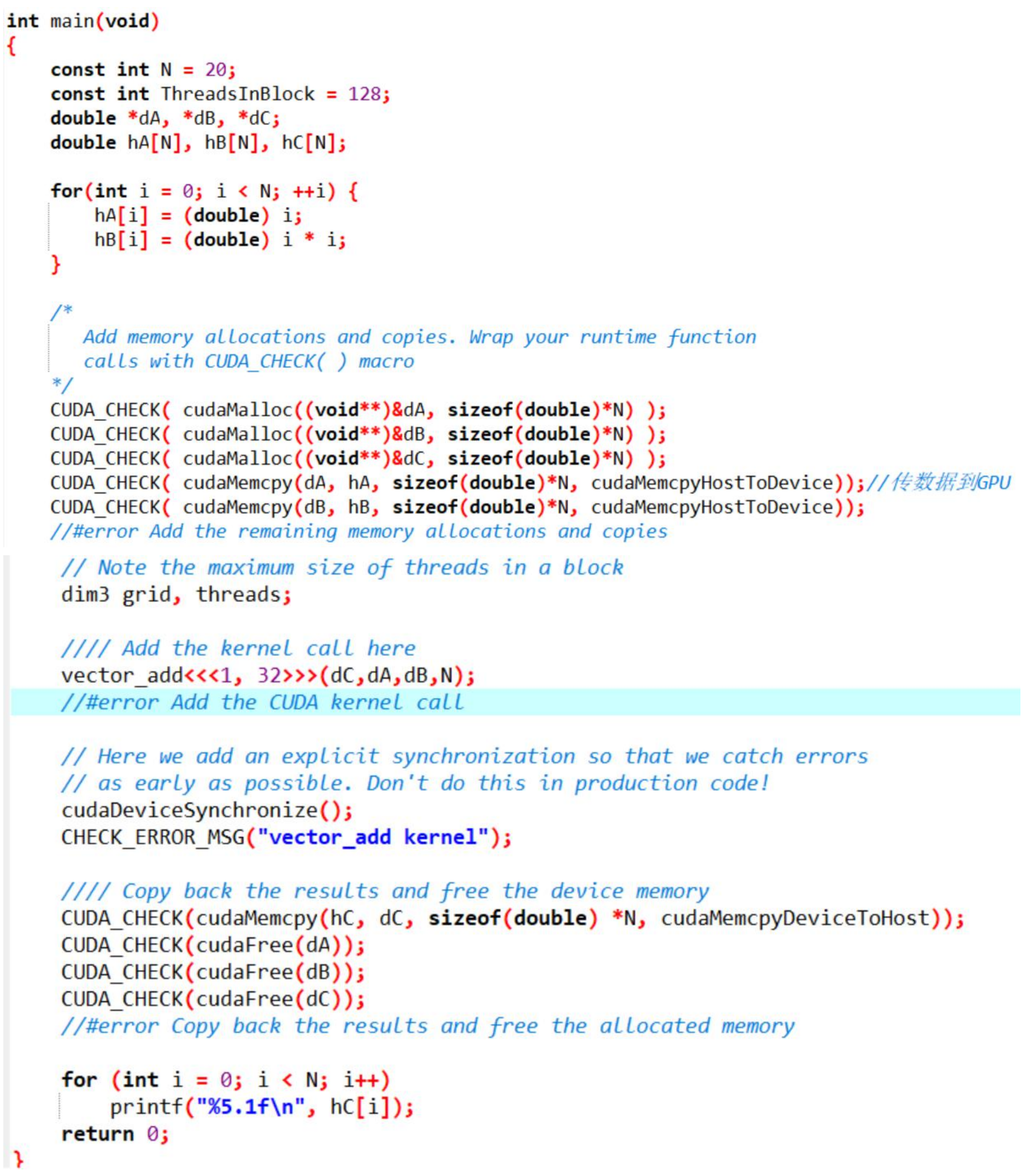
（注：本次实验，使用超算习堂进行测试。）

实验一：基于 error-test.cu 的测试：

代码如下：



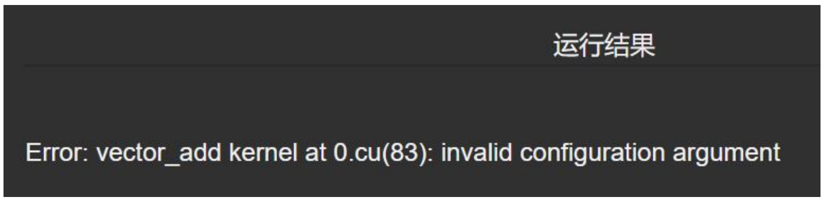
```
#include <cstdio>
#include <cmath>
//#include "error_checks.h" // Macros CUDA_CHECK and CHECK_ERROR_MSG

#ifndef COURSE_UTIL_H_
#define COURSE_UTIL_H_

#include <cstdio>
#include <cstdlib>

#define CUDA_CHECK(errarg)   __checkErrorFunc(errarg, __FILE__, __LINE__)
#define CHECK_ERROR_MSG(errstr) __checkErrMsgFunc(errstr, __FILE__, __LINE__)

inline void __checkErrorFunc(cudaError_t errarg, const char* file,
                 const int line)
{
    if(errarg) {
    fprintf(stderr, "Error at %s(%i)\n", file, line);
    exit(EXIT_FAILURE);
    }
}
inline void __checkErrMsgFunc(const char* errstr, const char* file,
                 const int line)
{
    cudaError_t err = cudaGetLastError();
    if(err != cudaSuccess) {
    fprintf(stderr, "Error: %s at %s(%i): %s\n",
        errstr, file, line, cudaGetErrorString(err));
    exit(EXIT_FAILURE);
    }
}

#endif

__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
    */
    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));//传数据到GPU
    CUDA_CHECK( cudaMemcpy(dB, hB, sizeof(double)*N, cudaMemcpyHostToDevice));
    //#error Add the remaining memory allocations and copies
    // Note the maximum size of threads in a block
    dim3 grid, threads;

    //// Add the kernel call here
    vector_add<<<1, 32>>>(dC,dA,dB,N);
    //#error Add the CUDA kernel call

    // Here we add an explicit synchronization so that we catch errors
    // as early as possible. Don't do this in production code!
    cudaDeviceSynchronize();
    CHECK_ERROR_MSG("vector_add kernel");

    //// Copy back the results and free the device memory
    CUDA_CHECK(cudaMemcpy(hC, dC, sizeof(double) *N, cudaMemcpyDeviceToHost));
    CUDA_CHECK(cudaFree(dA));
    CUDA_CHECK(cudaFree(dB));
    CUDA_CHECK(cudaFree(dC));
    //#error Copy back the results and free the allocated memory

    for (int i = 0; i < N; i++)
        printf("%5.1f\n", hC[i]);
    return 0;
}
```

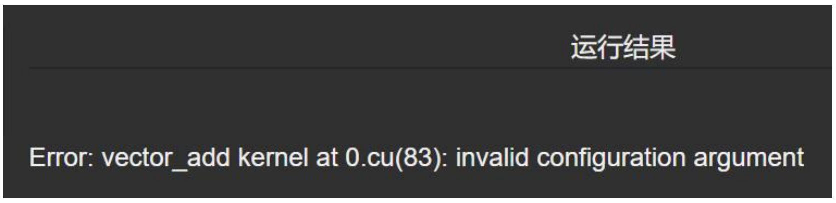
1. ”What happens if you try to launch kernel with too large block size? When do you catch the error if you remove the cudaDeviceSynchronize() call?”

由于过大 block size 会抢占计算资源。在超算习堂上面交了(vector_add<<<1, 1025>>>(dC, dA, dB, N);)之后，出现了这样的错误：



```
Error: vector_add kernel at 0.cu(83): invalid configuration argument
```

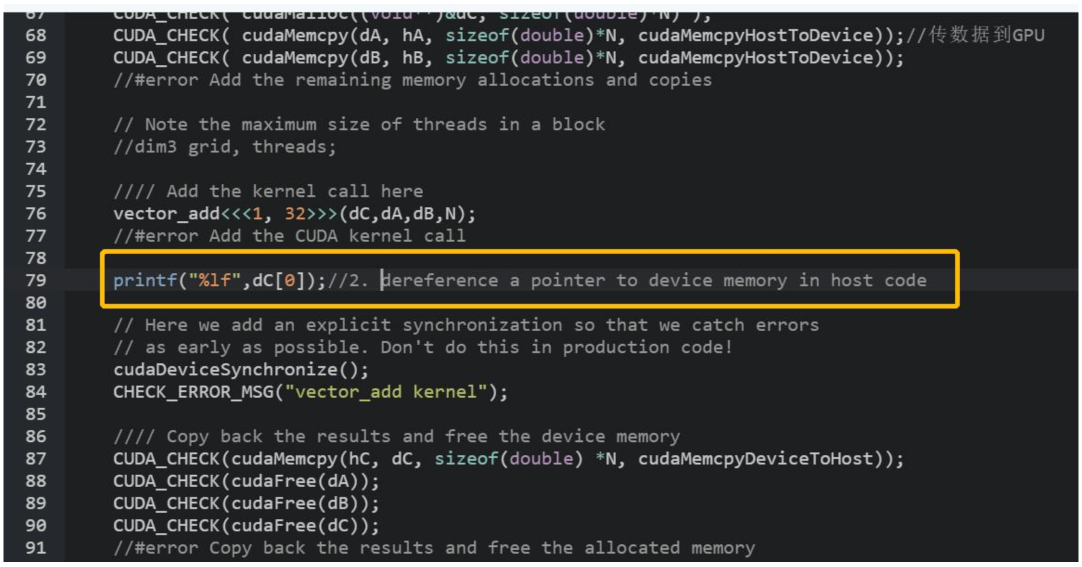
删去设备同步后，报错如下：



```
运行结果

Error: vector_add kernel at 0.cu(83): invalid configuration argument
```

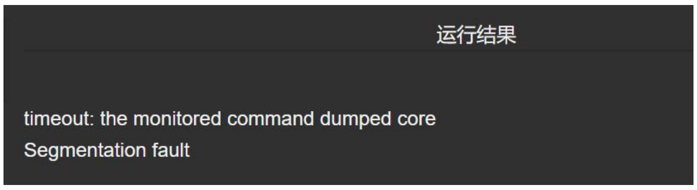
2. What happens if you try to dereference a pointer to device memory in host code?

Decive memory 指的是 GPU 内存，host 指的是 CPU。
“dereference 解引用”可以相当于在 host 代码中输出 device 变量的值。



```
67      CUDA_CHECK( cudaMalloc((void**)&dC, sizeof(double)*N) );
68      CUDA_CHECK( cudaMemcpy(dA, hA, sizeof(double)*N, cudaMemcpyHostToDevice));//传数据到GPU
69      CUDA_CHECK( cudaMemcpy(dB, hB, sizeof(double)*N, cudaMemcpyHostToDevice));
70      //#error Add the remaining memory allocations and copies
71
72      // Note the maximum size of threads in a block
73      //dim3 grid, threads;
74
75      //// Add the kernel call here
76      vector_add<<<1, 32>>>(dC,dA,dB,N);
77      //#error Add the CUDA kernel call
78
79      printf("%lf",dC[0]);//2. dereference a pointer to device memory in host code
80
81      // Here we add an explicit synchronization so that we catch errors
82      // as early as possible. Don't do this in production code!
83      cudaDeviceSynchronize();
84      CHECK_ERROR_MSG("vector_add kernel");
85
86      //// Copy back the results and free the device memory
87      CUDA_CHECK(cudaMemcpy(hC, dC, sizeof(double) *N, cudaMemcpyDeviceToHost));
88      CUDA_CHECK(cudaFree(dA));
89      CUDA_CHECK(cudaFree(dB));
90      CUDA_CHECK(cudaFree(dC));
91      //#error Copy back the results and free the allocated memory
```

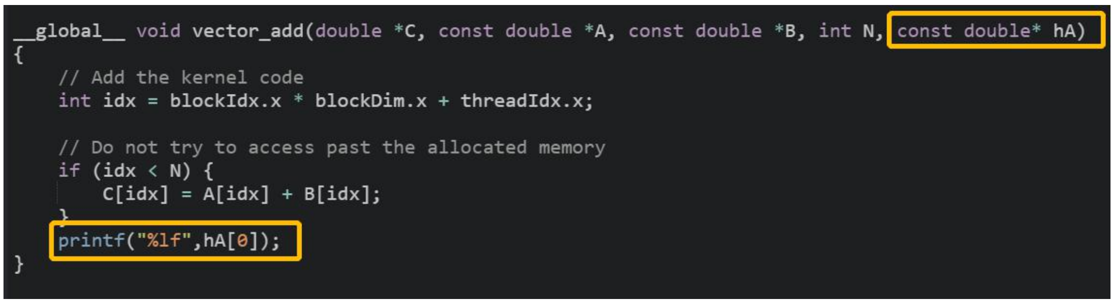
报错结果如下：



```
运行结果

timeout: the monitored command dumped core
Segmentation fault
```

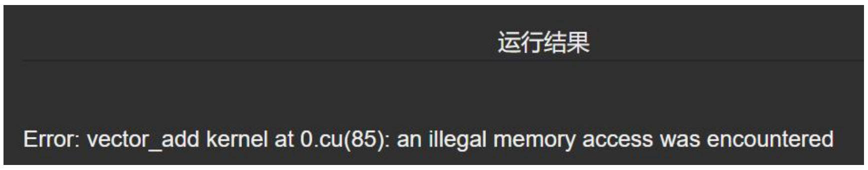
3. What if you try to access host memory from the kernel?

根据题目意思，试图在 kernel 函数中获取 host memory。故因此在 kernel 函数中，多传入一个参数 hA，并在 kernel 函数中试图输出这个*hA 的值。



```
__global__ void vector_add(double *C, const double *A, const double *B, int N, const double* hA)
{
    // 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];
    }
    printf("%lf",hA[0]);
}
```

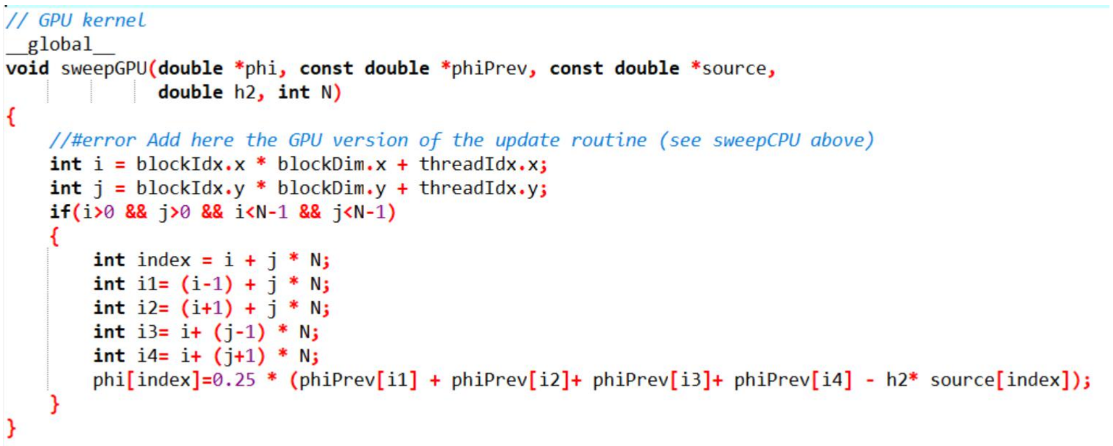
出现“内存访问”错误：



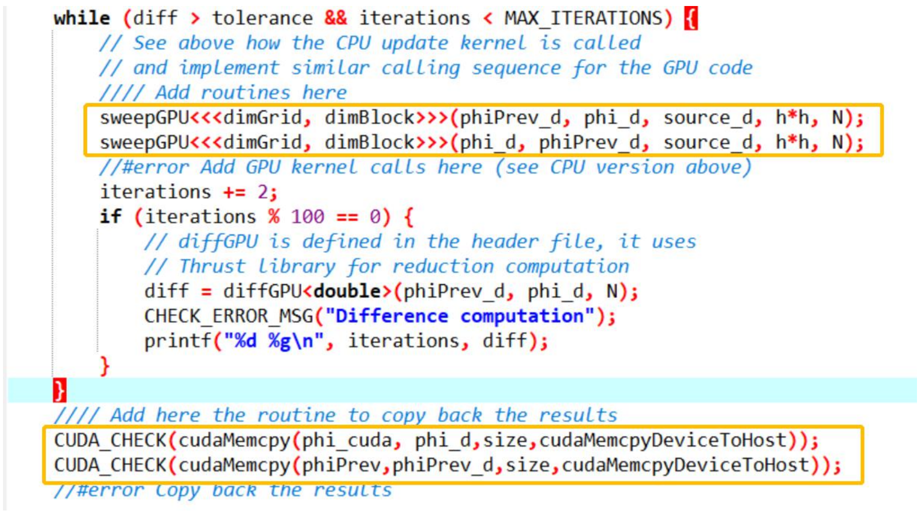
```
Error: vector_add kernel at 0.cu(85): an illegal memory access was encountered
```

实验二：Jacobi 迭代的实现

根据 CPU 的 sweepCPU 版本，可以写出 GPU 的版本：

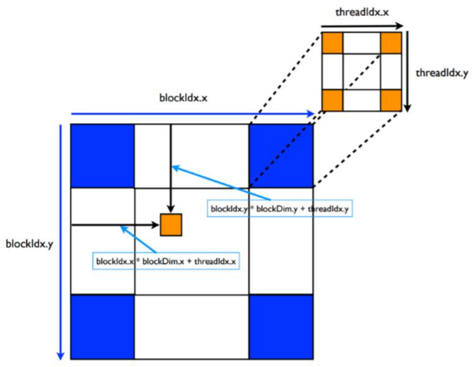


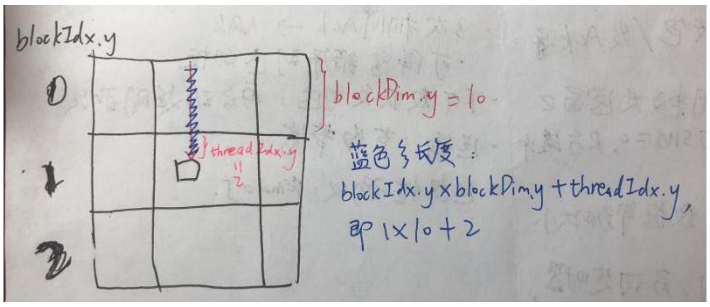
```
// GPU kernel
__global__
void sweepGPU(double *phi, const double *phiPrev, const double *source,
              double h2, int N)
{
    //#error Add here the GPU version of the update routine (see sweepCPU above)
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if(i>0 && j>0 && i<N-1 && j<N-1)
    {
        int index = i + j * N;
        int i1= (i-1) + j * N;
        int i2= (i+1) + j * N;
        int i3= i+ (j-1) * N;
        int i4= i+ (j+1) * N;
        phi[index]=0.25 * (phiPrev[i1] + phiPrev[i2]+ phiPrev[i3]+ phiPrev[i4] - h2* source[index]);
    }
}
```



```
    while (diff > tolerance && iterations < MAX_ITERATIONS) {
        // See above how the CPU update kernel is called
        // and implement similar calling sequence for the GPU code
        //// Add routines here
        sweepGPU<<<dimGrid, dimBlock>>>(phiPrev_d, phi_d, source_d, h*h, N);
        sweepGPU<<<dimGrid, dimBlock>>>(phi_d, phiPrev_d, source_d, h*h, N);
        //#error Add GPU kernel calls here (see CPU version above)
        iterations += 2;
        if (iterations % 100 == 0) {
            // diffGPU is defined in the header file, it uses
            // Thrust library for reduction computation
            diff = diffGPU<double>(phiPrev_d, phi_d, N);
            CHECK_ERROR_MSG("Difference computation");
            printf("%d %g\n", iterations, diff);
        }
    }
    //// Add here the routine to copy back the results
    CUDA_CHECK(cudaMemcpy(phi_cuda, phi_d,size,cudaMemcpyDeviceToHost));
    CUDA_CHECK(cudaMemcpy(phiPrev,phiPrev_d,size,cudaMemcpyDeviceToHost));
    //#error Copy back the results
```

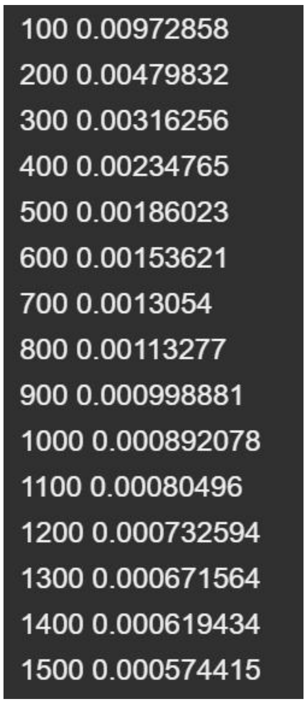
其中，值得注意的是 sweepCPU 函数中 i 和 j 的计算。可以这样理解：


threadIdx.x
threadIdx.y
blockIdx.x
blockIdx.y
blockIdx.y * blockDim.y + threadIdx.y
blockIdx.x * blockDim.x + threadIdx.x

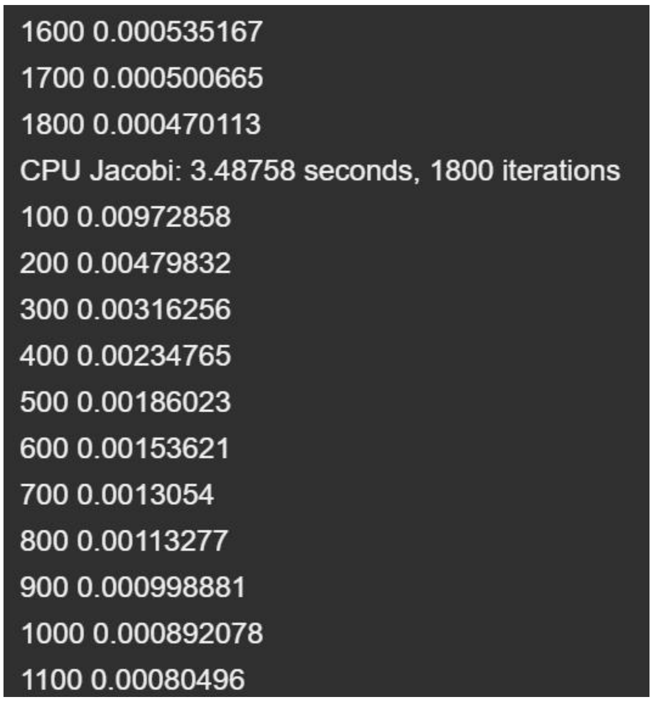


1. **Write the missing CUDA kernel *sweepGPU* that implements the same algorithm as the *sweepCPU* function. Check that the reported average difference is in the order of the numerical accuracy.**

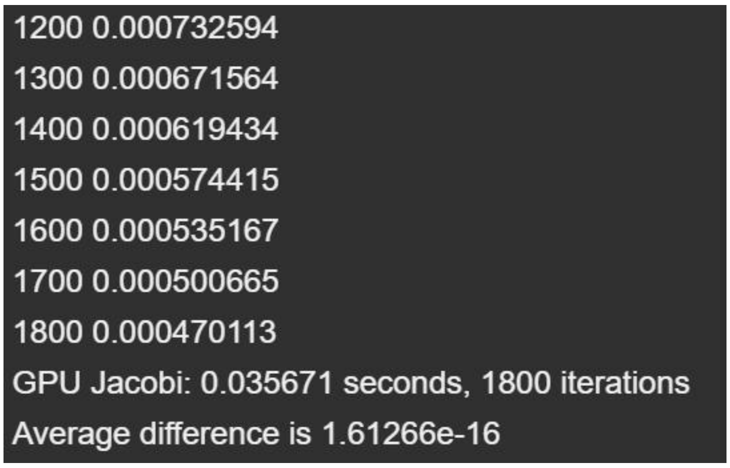
运行以上代码，结果如下：



```
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: 3.48758 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: 0.035671 seconds, 1800 iterations
Average difference is 1.61266e-16
```

发现 CPU 和 GPU 计算出来的平均误差。

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

题目要求我们修改 blocksize 和 gridsize，而有代码可知，问题规模为 512*512，而 1 个 block 中有是 16*16 个 thread，因此相当于有 32*32 个 block。

即原始 block size=16, grid size=32。修改后比较结果在“三、实验结果”中呈现。

三、 实验结果

（注：实验一的结果已在上面呈现。）

1. 实验二第一题的结果：

Numerical accuracy	Average difference
5e-1	9.4701e-17
5e-2	9.4701e-17
5e-3	9.78223e-17
5e-4	1.61266e-16

对于题目中的“Check that the reported average difference is in the order of the numerical accuracy.”，笔者理解为看平均误差是不是符合误差数量级，也就是在不在规定的误差数量级内。按照以上的执行结果，可以看出是在误差的数量级内，满足条件。（在数字精度在小于 5e-4 之后，就可运行但没有结果输出了）

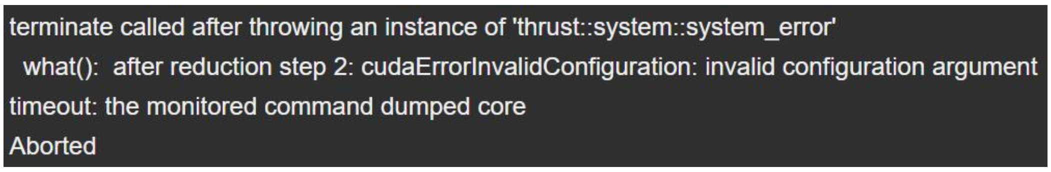
因此，平均误差符合误差数量级。

2. 实验二第二题的结果：

Grid size	Block size	Execution time(s)
128	4	0.065769
64	8	0.039449

32	16	0.035671
16	32	0.036349

实验结果：经过比较可知，当问题规模为 512 时，在 grid size=32,block size=16 时 GPU 计算性能最好；且当 block size>32 时，程序会报以下错误：



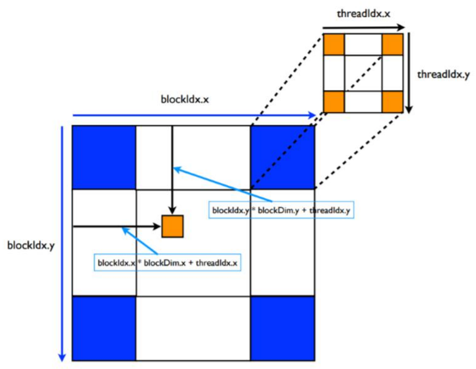
```
terminate called after throwing an instance of 'thrust::system::system_error'
  what():  after reduction step 2: cudaErrorInvalidConfiguration: invalid configuration argument
timeout: the monitored command dumped core
Aborted
```

四、 遇到的问题及解决方法

本次实验考察的是 CUDA 的编程，我认为难度中等，关键在于对 CUDA 语法和知识点的理解与掌握。

在实验一中，遇到了问题是：删去同步语句之后没有报错。后来，经过仔细阅读题目和与同学讨论，发现删去同步的操作是在修改了 block size 为很大的值的情况下去做的。

第二个问题在于如何理解 CUDA 的内存组织形式。后来，在与同学讨论后，发现了老师给予的参考代码是用“一维数组来模拟二维数组的储存”，加深了对 GPU 内存组织的理解（如下图），掌握了下标的计算方法。


threadIdx.x
threadIdx.y
blockIdx.x
blockIdx.y * blockDim.y + threadIdx.y
blockIdx.y
blockIdx.x * blockDim.x + threadIdx.x