17级并行与分布式计算

HW6 17341137 宋震鹏

$1 实验要求

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.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. 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:

    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.

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

$2 实验分析

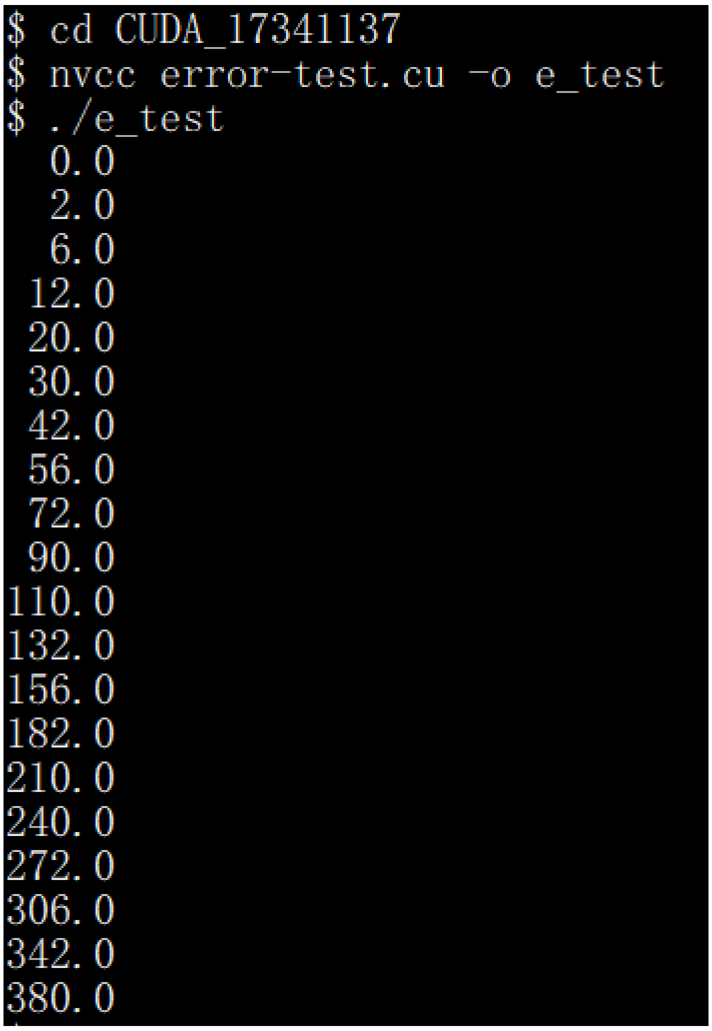
实验一

参考代码文件`error-test.cu`，补全内存分配、复制、启动内核的代码。

检查代码，回复题目的三个问题。

> 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?
>
>   > Segmentation fault. (Core Dumped.)
>
> 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.
>
>   > Error: vector_add kernel at error-test.cu(56): an illegal memory access was encountered.

正常运行截图



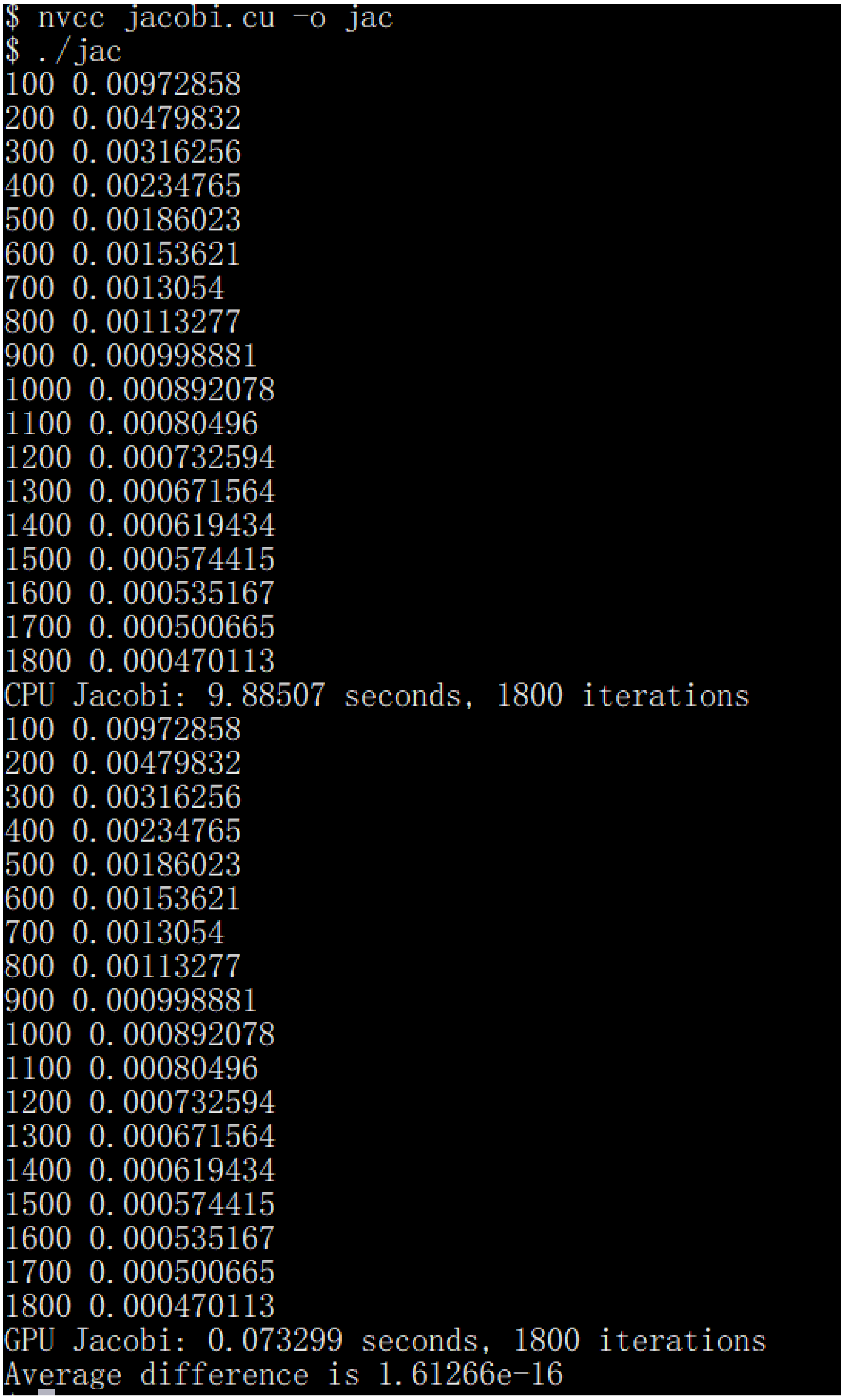
```
$ cd CUDA_17341137
$ nvcc error-test.cu -o e_test
$ ./e_test
  0.0
  2.0
  6.0
 12.0
 20.0
 30.0
 42.0
 56.0
 72.0
 90.0
110.0
132.0
156.0
182.0
210.0
240.0
272.0
306.0
342.0
380.0
```

实验二

实现一个有限差分的雅各比迭代。需要实现sweepGPU。

细节见下代码。

正常运行截图



```
$ nvcc jacobi.cu -o jac
$ ./jac
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: 9.88507 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: 0.073299 seconds, 1800 iterations
Average difference is 1.61266e-16
```

$3 实验代码

实验一

```
#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
    */
    CUDA_CHECK( cudaMalloc((void**)&dA, sizeof(double)*N) );
    // #error Add the remaining memory allocations and copies
    CUDA_CHECK( cudaMalloc((void**)&dB, sizeof(double)*N) );
    CUDA_CHECK( cudaMalloc((void**)&dC, sizeof(double)*N) );

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

    // Note the maximum size of threads in a block
    dim3 threads(ThreadsInBlock), grid((N + threads.x - 1) / threads.x);

    //// Add the kernel call here
    // #error Add the CUDA kernel call
    // vector_add(double *C, const double *A, const double *B, int N);
    // // dereference host pointer hA, hB.
    // vector_add <<<grid, threads>>> (dC, hA, hB, N);

    vector_add <<<grid, threads>>> (dC, dA, dB, N);

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

    // // dereference device pointer dC[i]
    // for (int i = 0; i < N; i++)
    //     printf("%5.1f\n", dC[i]);

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

    return 0;
}
```

实验二

```
#include <sys/time.h>
#include <cstdio>
#include "jacobi.h"
#include "error_checks.h"

// Change this to 0 if CPU reference result is not needed
#define COMPUTE_CPU_REFERENCE 1
#define MAX_ITERATIONS 3000

// CPU kernel
void sweepCPU(double* phi, const double *phiPrev, const double *source,
              double h2, int N)
{
    int i, j;
    int index, i1, i2, i3, i4;

    for (j = 1; j < N-1; j++) {
        for (i = 1; i < N-1; i++) {
            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]);
        }
    }
}

// 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;
    int index = i + j*N;
    int i1, i2, i3, i4;

    i1 = (i-1) +   j   * N;
    i2 = (i+1) +   j   * N;
    i3 =   i   + (j-1) * N;
    i4 =   i   + (j+1) * N;

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


double compareArrays(const double *a, const double *b, int N)
{
    double error = 0.0;
    int i;
    for (i = 0; i < N*N; i++) {
        error += fabs(a[i] - b[i]);
    }
    return error/(N*N);
}


double diffCPU(const double *phi, const double *phiPrev, int N)
{
    int i;
    double sum = 0;
    double diffsum = 0;

    for (i = 0; i < N*N; i++) {
        diffsum += (phi[i] - phiPrev[i]) * (phi[i] - phiPrev[i]);
        sum += phi[i] * phi[i];
    }

    return sqrt(diffsum/sum);
}
```

```
int main()
{
    timeval t1, t2; // Structs for timing
    const int N = 512;
    double h = 1.0 / (N - 1);
    int iterations;
    const double tolerance = 5e-4; // Stopping condition
    int i, j, index;

    const int blocksize = 16;

    double *phi      = new double[N*N];
    double *phiPrev  = new double[N*N];
    double *source   = new double[N*N];
    double *phi_cuda = new double[N*N];

    double *phi_d, *phiPrev_d, *source_d;
    // Size of the arrays in bytes
    const int size = N*N*sizeof(double);
    double diff;

    // Source initialization
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            double x, y;
            x = (i - N / 2) * h;
            y = (j - N / 2) * h;
            index = j + i * N;
            if (((x - 0.25) * (x - 0.25) + y * y) < 0.1 * 0.1)
                source[index] = 1e10*h*h;
            else if (((x + 0.25) * (x + 0.25) + y * y) < 0.1 * 0.1)
                source[index] = -1e10*h*h;
            else
                source[index] = 0.0;
        }
    }

    CUDA_CHECK( cudaMalloc( (void**)&source_d, size) );
    CUDA_CHECK( cudaMemcpy(source_d, source, size, cudaMemcpyHostToDevice) );

    // Reset values to zero
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            index = j + i * N;
            phi[index] = 0.0;
            phiPrev[index] = 0.0;
        }
    }

    CUDA_CHECK( cudaMalloc( (void**)&phi_d, size) );
    CUDA_CHECK( cudaMalloc( (void**)&phiPrev_d, size) );
    CUDA_CHECK( cudaMemcpy((void*)phi_d, (void*)phi, size, cudaMemcpyHostToDevice) );
    CUDA_CHECK( cudaMemcpy((void*)phiPrev_d, (void*)phiPrev, size, cudaMemcpyHostToDevice)
);

    // CPU version
    if(COMPUTE_CPU_REFERENCE) {
        gettimeofday(&t1, NULL);

        // Do sweeps untill difference is under the tolerance
        diff = tolerance * 2;
        iterations = 0;
        while (diff > tolerance && iterations < MAX_ITERATIONS) {
            sweepCPU(phiPrev, phi, source, h * h, N);
            sweepCPU(phi, phiPrev, source, h * h, N);

            iterations += 2;
            if (iterations % 100 == 0) {
                diff = diffCPU(phi, phiPrev, N);
                printf("%d %g\n", iterations, diff);
            }
        }
        gettimeofday(&t2, NULL);
        printf("CPU Jacobi: %g seconds, %d iterations\n",
               t2.tv_sec - t1.tv_sec +
               (t2.tv_usec - t1.tv_usec) / 1.0e6, iterations);
    }

    // GPU version

    dim3 dimBlock(blocksize, blocksize);
    dim3 dimGrid((N + blocksize - 1) / blocksize, (N + blocksize - 1) / blocksize);

    //do sweeps until diff under tolerance
    diff = tolerance * 2;
    iterations = 0;

    gettimeofday(&t1, NULL);

    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
        // #error Add GPU kernel calls here (see CPU version above)
        sweepGPU <<<dimGrid, dimBlock>>> (phiPrev_d, phi_d, source_d, h * h, N);
        sweepGPU <<<dimGrid, dimBlock>>> (phi_d, phiPrev_d, source_d, h * h, N);

        iterations += 2;
        // Test
        // printf("GPU iter: %d\n", iterations);

        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
    // #error Copy back the results
    CUDA_CHECK( cudaMemcpy(phi_cuda, phi_d, size, cudaMemcpyDeviceToHost) );

    gettimeofday(&t2, NULL);
    printf("GPU Jacobi: %g seconds, %d iterations\n",
           t2.tv_sec - t1.tv_sec +
           (t2.tv_usec - t1.tv_usec) / 1.0e6, iterations);

    //// Add here the clean up code for all allocated CUDA resources
    // #error Add here the clean up code
    CUDA_CHECK( cudaFree(phi_d) );
    CUDA_CHECK( cudaFree(phiPrev_d) );
    CUDA_CHECK( cudaFree(source_d) );


    if (COMPUTE_CPU_REFERENCE) {
        printf("Average difference is %g\n", compareArrays(phi, phi_cuda, N));
    }

    delete[] phi;
    delete[] phi_cuda;
    delete[] phiPrev;
    delete[] source;

    return EXIT_SUCCESS;
}
```

$5 实验总结

实验一

在实验一中，我首先学习了CUDA编程的基本要求和概念。

对照参考代码中#error部分，完成了剩余代码的实现。

实验二

在实验二中，我先学习了CUDA内存分布，然后参考CPUsweep函数实现了GPU的sweep函数。

最后对照参考代码中#error部分，完成了剩余代码的实现。

17341137 宋震鹏 19/6/13