并行与分布式作业

“ 黄炜钊”

第六次作业

姓名：黄炜钊

班级：行政三班

学号：18340066

1. 问题描述

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

1.Write the missing CUDAkernel

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 withdifferent grid and block sizes and compare the execution times.

1. 解决方案

1、

（1）block size太大会抢占计算资源，一个报错例子如下：

Error: vector\_add kernel at 0\_4323.cu(86): invalid configuration argument

yhrun: error: gn07: task 1: Exited with exit code 1

删除了cudaDeviceSynchronize() call之后，报错信息如下：

Error: vector\_add kernel at 0\_4323.cu(86): invalid configuration argument

yhrun: error: gn07: task 0: Exited with exit code 1

（2）当运行到解除引用的时候会发生报错，具体信息如下：

Error: vector\_add kernel at 0\_4323.cu(85): invalid configuration argument

yhrun: error: gn07: task 0: Exited with exit code 1

（3）在编译的时候就会报错，具体信息如下：

0\_4323.cu(48): error: identifier "hb" is undefined in device code

1 error detected in the compilation of "/tmp/tmpxft\_0000440e\_00000000-11\_0\_4323.cpp2.i".

slurmd[gn07]: execve(): 0\_4323.cu.out: No such file or directory

yhrun: error: gn07: task 0: Exited with exit code 2

rm: cannot remove ‘0\_4323.cu.out’: No such file or directory

代码如下：

#include <stdio.h>

#include <math.h>

//#include "error\_checks.h" // Macros CUDA\_CHECK and CHECK\_ERROR\_MSG

// This header provides two helper macros for error checking

// See the exercise skeletons and answers for usage examples.

#ifndef COURSE\_UTIL\_H\_

#define COURSE\_UTIL\_H\_

#include <stdio.h>

#include <stdlib.h>

#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));

    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;

}

2、代码如下：

//err\_checker.h

// This header provides two helper macros for error checking

// See the exercise skeletons and answers for usage examples.

#ifndef COURSE\_UTIL\_H\_

#define COURSE\_UTIL\_H\_

#include <stdio.h>

#include <stdlib.h>

#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

//jacbi.h

#ifndef EX3\_H\_

#define EX3\_H\_

#include <thrust/device\_vector.h>

#include <thrust/functional.h>

#include <thrust/transform\_reduce.h>

#include <thrust/iterator/zip\_iterator.h>

// Helper function prototypes

double compareArrays(const double \*a, const double \*b, int N);

double diffCPU(const double \*a, const double \*b, int N);

void sweepCPU(double \*phi, const double \*phiPrev,

              const double \*source, double h2, int N);

/\* -------------------------------------------------------------------------

   EXTRACURRICULAR ACTIVITIES

   This part provides the reduction operation (in this case summation of

   difference of two arrays) using thrust library. Thrust mimics the

   syntax and design of standard template library (STL) of C++. Thrust is

   also a part of CUDA 4 SDK.

   More information can be found from thrust home page

   http://code.google.com/p/thrust/

   ----------------------------------------------------------------------- \*/

template <typename T>

class square\_diff\_thr : public thrust::unary\_function<thrust::tuple<T, T>, T>

{

public:

    \_\_host\_\_ \_\_device\_\_

        T

        operator()(const thrust::tuple<T, T> &x) const

    {

        return (thrust::get<1>(x) - thrust::get<0>(x)) \*

               (thrust::get<1>(x) - thrust::get<0>(x));

    }

};

template <typename T>

class square\_thr : public thrust::unary\_function<T, T>

{

public:

    \_\_host\_\_ \_\_device\_\_

        T

        operator()(const T &x) const

    {

        return x \* x;

    }

};

template <typename T>

T diffGPU(T \*A\_d, T \*B\_d, int N)

{

    typedef thrust::device\_ptr<T> FloatIterator;

    typedef thrust::tuple<FloatIterator, FloatIterator> IteratorTuple;

    typedef thrust::zip\_iterator<IteratorTuple> ZipIterator;

    thrust::device\_ptr<T> A\_ptr(A\_d);

    thrust::device\_ptr<T> B\_ptr(B\_d);

    ZipIterator first =

        thrust::make\_zip\_iterator(thrust::make\_tuple(A\_ptr, B\_ptr));

    ZipIterator last =

        thrust::make\_zip\_iterator(thrust::make\_tuple(A\_ptr + N \* N,

                                                     B\_ptr + N \* N));

    T a1 = thrust::transform\_reduce(first, last, square\_diff\_thr<T>(),

                                    static\_cast<T>(0), thrust::plus<T>());

    T a2 = thrust::transform\_reduce(B\_ptr, B\_ptr + N \* N,

                                    square\_thr<T>(), static\_cast<T>(0),

                                    thrust::plus<T>());

    return sqrt(a1 / a2);

}

#endif // EX3\_H\_

//jacobi.cu

#include <time.h>

#include <stdio.h>

//#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;

    if (i > 0 && j > 0 && i < N - 1 && j < N - 1)

    { // be careful!

        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]);

    }

}

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()

{

    clock\_t 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(phi\_d, phi, size, cudaMemcpyHostToDevice));

    CUDA\_CHECK(cudaMemcpy(phiPrev\_d, phiPrev, size, cudaMemcpyHostToDevice));

    // CPU version

    if (COMPUTE\_CPU\_REFERENCE)

    {

        t1 = clock();

        // 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);

            }

        }

        t2 = clock();

        printf("CPU Jacobi: %g ms, %d iterations\n",

               t2 - t1,

               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;

    t1 = clock();

    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, h \* h, N);

        sweepGPU<<<dimGrid, dimBlock>>>(phi\_d, phiPrev\_d, source, 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, phi\_d, size, cudaMemcpyDeviceToHost));

    CUDA\_CHECK(cudaMemcpy(phiPrev, phiPrev\_d, size, cudaMemcpyDeviceToHost));

    //#error Copy back the results

    t2 = clock();

    printf("GPU Jacobi: %g ms, %d iterations\n",

           t2 - t1,

           iterations);

    //// Add here the clean up code for all allocated CUDA resources

    CUDA\_CHECK(cudaFree(phi\_d));

    CUDA\_CHECK(cudaFree(phiPrev\_d));

    CUDA\_CHECK(cudaFree(source\_d));

    //#error Add here the clean up code

    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;

}