CUDA编程入门：向量加法和矩阵乘法

本篇博客总结自我《并行计算》CUDA编程实验

在《并行计算》课程中我们学习了CUDA编程模型，在这里我实现了用CUDA实现简单的向量加法和矩阵乘法，并在USTC 联想深腾7000G GPU集群上完成了验证。

1.CUDA编程模型简介

CUDA（Compute Unified Device Architecture）是显卡厂商NVIDIA推出的运算平台。 CUDA是一种由NVIDIA推出的通用并行计算架构，该架构使GPU能够解决复杂的计算问题。 它包含了CUDA指令集架构（ISA）以及GPU内部的并行计算引擎。 开发人员现在可以使用C语言来为CUDA架构编写程序，C语言是应用最广泛的一种高级编程语言。所编写出的程序于是就可以在支持CUDA的处理器上以超高性能运行。

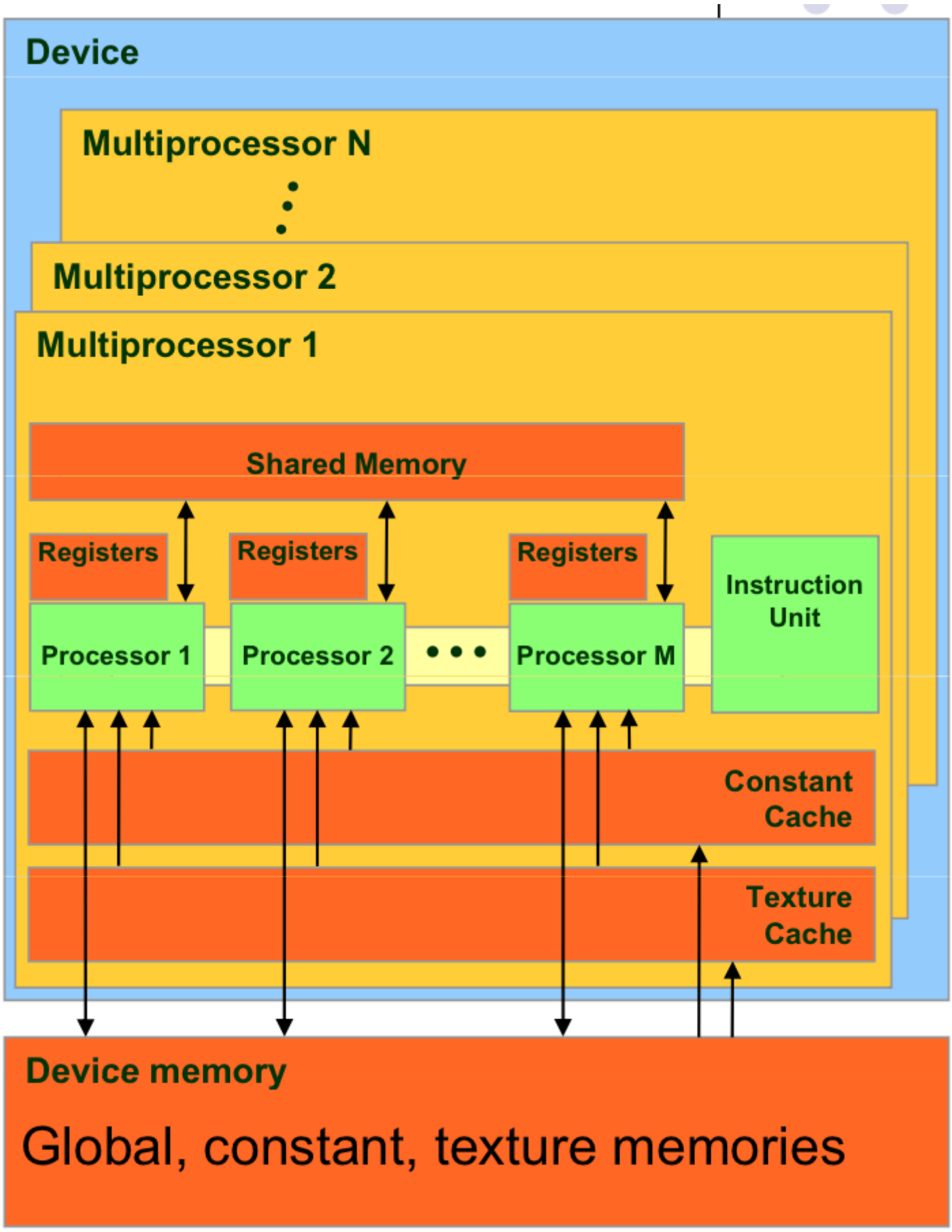
（1）GPU架构

主流的GPU架构是以NVIDIA GeForce系列为代表流处理器阵列架构，GPU上集成了上百个流式多处理器（Stream Multiprocessor，SM），如GeForce 8800GTX包含了128个流处理器。

每个SM包含若干独立受控于指令的可同步执行的纯量处理器（Scalar Processor，SP）、Cache（指令、常数、纹理）、多线程指令发射单元、特殊功能单元（Special Function Unit，SFU）以及供线程块共享数据的Shared Memory。

（2）存储器层次结构
CUDA编程模型下内存可以分为三个层次：流式多处理器片上内存、GPU设备内存以及主机内存。

- 流式多处理器片上内存包括每个线程独立拥有的寄存器以及局部内存（Local Memory）、片上共享内存（On-chip Shared Memory，供线程块内共享数据）。每个流式多处理器有自己常量Cache和纹理Cache。
- GPU设备内存（Device Memory）包括全局内存（Global Memory）以及只读的常量内存（Constant Memory）和纹理内存（Texture Memory）。所有流式多处理器均可访问GPU设备内存。
- 主机内存（Host Memory）是一般意义的内存，GPU设备不能直接访问主机内存，需要通过拷贝API传递数据（见后）。



详见：http://blog.csdn.net/augusdi/article/details/12186939

（3）线程组织结构

CUDA中线程也可以分成三个层次：线程、线程块和线程网络。

- 线程是CUDA中基本执行单元，由硬件支持、开销很小，每个线程执行相同代码
- 线程块（Block）是若干线程的分组，Block内一个块至多512个线程，线程块可以是一维、二维或者三维的
- 线程网络（Grid）是若干线程块的网格，Grid是一维和二维的。

线程用ID索引，线程块内用局部ID标记threadID，配合blockDim和blockID可以计算出全局ID，用于SIMD分配任务。

（4）CUDA程序结构
CUDA程序的结构大体是：`{主机串行->GPU并行}+ -> 主机串行`，这样的串并交叉结构。主机串行过渡到GPU并行时需要将数据从主机内存上拷贝到GPU设备内存上，GPU执行完毕时也需要把数据拷贝回来。

（5）CUDA内核函数
主机调用设备代码的唯一接口就是Kernel函数，使用限定符: `__global__`(见后)。

调用内核函数需要在内核函数名后添加 `<<<>>>` 指定内核函数配置，如 `<<<DimGrid, DimBlock>>>` 指定线程网络和线程块维度。若当前硬件无法满足用户配置，则内核函数不会被执行，直接返回错误。

（6）CUDA限定符
函数限定符：（默认host、global异步、主机不能调device；设备上执行的函数参数数目固定、不能声明静态变量且不支持递归调用）

函数限定符	在何处执行	从何处调用	特性
device	设备	设备	函数的地址无法获取
global	设备	主机	返回类型必须为空
host	主机	主机	等同于不使用任何限定符

变量限定符：（shared共享一致性必须由显式线程同步保证）

限定符	位于何处	可以访问的线程	主机访问
device	全局存储器	线程网格内的 所有线程	通过运行时库访问
constant	固定存储器	线程网格内的 所有线程	通过运行时库访问
shared	共享存储器	线程块内的 所有线程	不可从主机访问

（7）同步

- CPU启动kernel函数是异步的，它并不会阻塞等到GPU执行完kernel函数才执行后面的CPU部分，因此如果后续程序立即需要用到上一个kernel函数的结果我们需要显式设置同步障来阻塞CPU程序。
- 一个线程块内需要同步共享存储器的共享变量（`__shared__`）时，需要在使用前显式调用__syncthreads()`同步块内所有线程。
- 同一个Grid中不同Block之间无法设置同步。

（8）CUDA运行时API
只介绍一下我程序中用到的最基础的内存管理函数，其他详见官网：http://docs.nvidia.com/cuda/cuda-runtime-api/index.html

cudaMemcpy

```
__host__ cudaError_t cudaMemcpy( void* dst, const void* src, size_t count, cudaMemcpyKind kind )
    1
```

用于在主机和设备之间拷贝数据，其中cudaMemcpyKind枚举类型常用有cudaMemcpyHostToDevice表示把主机数据拷贝到内存以及逆向的cudaMemcpyDeviceToHost。

cudaMalloc

```
__host__ __device__ cudaError_t cudaMalloc( void** devPtr, size_t size )
    1
```

在设备上分配动态内存，两个限定符表示可以在主机或设备上调用。

cudaFree

```
__host__ __device__ cudaError_t cudaFree( void* devPtr )
    1
```

释放回收在设备上分配动态内存，两个限定符表示可以在主机或设备上调用。

2.向量加法程序简介

我实现的是长度为n的两个向量的加法，n由命令行运行参数确定。

程序流程为

这里主要讲一下Kernel函数配置设置和Kernel函数：
配置如下：

```
#define BLOCKSIZE 16

int gridsize = (int)ceil(sqrt(ceil(n / (BLOCKSIZE * BLOCKSIZE))));

dim3 dimBlock(BLOCKSIZE, BLOCKSIZE, 1);
dim3 dimGrid(gridsize, gridsize, 1);

add<<<dimGrid, dimBlock>>>(device_a, device_b, device_c, n);
```

初始化dimBlock为16*16*1的dim3类型，指定线程块的三个维度，这里第三维是1，即退化为16*16的二维线程块。为了最大化并行，安排每一个线程负责一次向量加法，那么则需要 `ceil(n / (16 * 16))` 个线程块，设置线程网络也为二维方阵的话，其gridsize取 `ceil(sqrt(ceil(n / (16 * 16))))` 。同样初始化维度变量dimGrid，注意Grid只能是二维以下的，第三个维度设置被自动忽略。

设置中用了上取整，是为了保证至少有一个线程完成向量每对元素的相加，那么这样设置可能会导致线程数目多于向量长度，因此在Kernel函数中需要让这些线程直接退出，而不是引起数组超界的错误。

下面来推导块号为blockIdx、线程号为threadIdx的线程完成向量计算的数组下标:

块内地址为 `threadIdx.x * blockDim.x + threadIdx.y` ，把线程号映射到 `[0, blockDim.x * blockDim.y - 1]` 。线程块地址为 `blockIdx.x * gridDim.x + blockIdx.y` ，把块号映射到 `[0, gridDim.x * gridDim.y - 1]` 。
所以该线程的地址为 `i = (blockIdx.x * gridDim.x + blockIdx.y) * blockDim.x * blockDim.y + threadIdx.x * blockDim.x + threadIdx.y`

向量加法Kernel函数定义如下：

```
__global__ void add(const int *a, const int *b, int *c, int n) {
    int i = (blockIdx.x * gridDim.x + blockIdx.y) * blockDim.x * blockDim.y + threadIdx.x *
blockDim.x + threadIdx.y;
    if (i < n) {
        c[i] = a[i] + b[i];
    }
}
```

- 1
- 2
- 3
- 4
- 5
- 6

注：先计算出线程地址i，如果i < n才计算否则该线程直接退出。

源代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <unistd.h>
#include <cuda.h>

#define RANDOM(x) (rand() % x)

#define MAX 100000

#define BLOCKSIZE 16


__global__ void add(const int *a, const int *b, int *c, int n) {
    int i = (blockIdx.x * gridDim.x + blockIdx.y) * blockDim.x * blockDim.y + threadIdx.x *
blockDim.x + threadIdx.y;
    if (i < n) {
        c[i] = a[i] + b[i];
    }
}


int main(int argc, char **argv) {
    int n = 512;
    int i;
    timeval start, finish;

    if (argc == 2) {
        n = atoi(argv[1]);
    }

    int *host_a = (int *)malloc(sizeof(int) * n);
    int *host_b = (int *)malloc(sizeof(int) * n);
    int *host_c = (int *)malloc(sizeof(int) * n);
    int *host_c2 = (int *)malloc(sizeof(int) * n);

    srand(time(NULL));

    for (i = 0; i < n; i++) {
        host_a[i] = RANDOM(MAX);
        host_b[i] = RANDOM(MAX);
    }

    cudaError_t error = cudaSuccess;

    int *device_a, *device_b, *device_c;
    error = cudaMalloc((void **)&device_a, sizeof(int) * n);
    error = cudaMalloc((void **)&device_b, sizeof(int) * n);
    error = cudaMalloc((void **)&device_c, sizeof(int) * n);

    if (error != cudaSuccess) {
        printf("Fail to cudaMalloc on GPU");
        return 1;
    }

//GPU parallel start
    gettimeofday(&start, 0);
```

```
    cudaMemcpy(device_a, host_a, sizeof(int) * n, cudaMemcpyHostToDevice);
    cudaMemcpy(device_b, host_b, sizeof(int) * n, cudaMemcpyHostToDevice);

    int gridsize = (int)ceil(sqrt(ceil(n / (BLOCKSIZE * BLOCKSIZE))));

    dim3 dimBlock(BLOCKSIZE, BLOCKSIZE, 1);
    dim3 dimGrid(gridsize, gridsize, 1);

    add<<<dimGrid, dimBlock>>>(device_a, device_b, device_c, n);
    cudaThreadSynchronize();

    cudaMemcpy(host_c, device_c, sizeof(int) * n, cudaMemcpyDeviceToHost);

    gettimeofday(&finish, 0);

    double t = 1000000 * (finish.tv_sec - start.tv_sec) + finish.tv_usec - start.tv_usec;
    printf("%lf ms\n", t / 1000);
//GPU parallel finish


//CPU serial start
    gettimeofday(&start, 0);

    for (i = 0; i < n; i++) {
        host_c2[i] = host_a[i] + host_b[i];
    }

    gettimeofday(&finish, 0);

    t = 1000000 * (finish.tv_sec - start.tv_sec) + finish.tv_usec - start.tv_usec;
    printf("%lf ms\n", t / 1000);
//CPU serial start

//check
    int errorNum = 0;
    for (int i = 0; i < n; i++) {
        if (host_c[i] != host_c2[i]) {
            errorNum ++;
            printf("Error occurs at index: %d: a + b = %d + %d = %d, but c = %d, c2 = %d\n", i,
host_a[i], host_b[i], host_a[i] + host_b[i], host_c[i], host_c2[i]);
        }
    }
    if (errorNum == 0) {
        printf("Successfully run on GPU and CPU!\n");
    } else {
        printf("%d error(s) occurs!\n", errorNum);
    }

    free(host_a);
    free(host_b);
    free(host_c);
    free(host_c2);

    cudaFree(device_a);
    cudaFree(device_b);
    cudaFree(device_c);
```

```
    return 0;
}
```

- 57
- 58
- 59
- 60
- 61
- 62
- 63
- 64
- 65
- 66
- 67
- 68
- 69
- 70
- 71
- 72
- 73
- 74
- 75
- 76
- 77
- 78
- 79
- 80
- 81
- 82
- 83
- 84
- 85
- 86
- 87
- 88
- 89
- 90
- 91
- 92
- 93
- 94
- 95
- 96
- 97
- 98
- 99
- 100
- 101
- 102
- 103
- 104
- 105
- 106
- 107
- 108
- 109
- 110
- 111
- 112
- 113
- 114

- 115
- 116

3.矩阵乘法程序简介

我实现的是维度为n*n的两个矩阵（方阵）的乘法，n由命令行运行参数确定。所有的二维矩阵都用一维数组表示，索引时计算对应下标。程序流程和向量加法几乎一模一样，不再重复说明。这里简单推导一下参数配置以及Kernel函数任务安排。
先观察矩阵乘法串行O(n3)的算法如下：

```
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        c[i * n + j] = 0;
        for (k = 0; k < n; k++) {
            c[i * n + j] += a[i * n + k] * b[k * n + j];
        }
    }
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8

对于每一个乘积矩阵c中任意元素 `c[i * n + j]` 需要做n次循环求和，那么我们很容易想到一种简单的并行思路：安排一个线程去完成这n次循环求和，那么可以用n2个线程同时完成这个任务那么最后的复杂度可以下降到O(n)。（我还想到一种策略是使用 `__shared__` 共享变量localsum，n3个线程每个线程同时只做1次加法并把和加在localsum中，挑选其中一个线程将结果写入c数组。但是由于线程块内同步 `__shared__` 共享变量需要时间开销，并且可能一个Block最多512个线程也不能完成一轮n循环而Block间又不能直接使用共享变量，所以最终没有实现这个想法。）
一个线程块仍设置有 `BLOCKSIZE * BLOCKSIZE = 16 * 16` 个线程，那么共需要 `ceil((n*n)/(16*16))` 个Block，即 `gridsize = ceil(sqrt(ceil((n*n)/(16*16))))` ，那么Kernel函数配置如下：

```
double num = ceil(pow((double)n,2) / pow((double)BLOCKSIZE, 2));
int gridsize = (int)ceil(sqrt(num));

dim3 dimBlock(BLOCKSIZE, BLOCKSIZE, 1);
dim3 dimGrid(gridsize, gridsize, 1);

multiply<<<dimGrid, dimBlock>>>(device_a, device_b, device_c, n);
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7

我们用一种和向量加法不同的任务分配方式来分配，把整个Grid的所有Block中的线程看作一个二维阵列，由于我们设置保证至少有n*n个线程，那么我们就取整个二维阵列阵列左上角一块n*n方阵作为计算节点。

`row = blockIdx.x * blockDim.x + threadIdx.x`，`col = blockIdx.y * blockDim.y + threadIdx.y`算出线程在二维阵列中的坐标，如果在n*n方阵中，则如前所述做一轮循环n次求和，刚好row和col和乘积矩阵的行列坐标对应。

源代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <unistd.h>
#include <cuda.h>

#define RANDOM(x) (rand() % x)

#define MAX 100000

#define BLOCKSIZE 16

__global__ void multiply(const int *a, const int *b, int *c, int n) {
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    int col = blockIdx.y * blockDim.y + threadIdx.y;

    int k;
    int sum = 0;

    if (row < n && col < n) {
        for (k = 0; k < n; k++) {
            sum += a[row * n + k] * b[k * n + col];
        }

        c[row * n + col] = sum;
    }
}

int main(int argc, char **argv) {
    int n = 512;
    int i, j, k;
    timeval start, finish;

    if (argc == 2) {
        n = atoi(argv[1]);
    }

    int *host_a = (int *)malloc(sizeof(int) * n * n);
    int *host_b = (int *)malloc(sizeof(int) * n * n);
    int *host_c = (int *)malloc(sizeof(int) * n * n);
    int *host_c2 = (int *)malloc(sizeof(int) * n * n);

    srand(time(NULL));

    for (i = 0; i < n * n; i++) {
        host_a[i] = RANDOM(MAX);
        host_b[i] = RANDOM(MAX);
    }

    cudaError_t error = cudaSuccess;

    int *device_a, *device_b, *device_c;
    error = cudaMalloc((void **)&device_a, sizeof(int) * n * n);
    error = cudaMalloc((void **)&device_b, sizeof(int) * n * n);
    error = cudaMalloc((void **)&device_c, sizeof(int) * n * n);

    if (error != cudaSuccess) {
```

```
        printf("Fail to cudaMalloc on GPU");
        return 1;
    }

//GPU parallel start
    gettimeofday(&start, 0);

    cudaMemcpy(device_a, host_a, sizeof(int) * n * n, cudaMemcpyHostToDevice);
    cudaMemcpy(device_b, host_b, sizeof(int) * n * n, cudaMemcpyHostToDevice);

    double num = ceil(pow((double)n,2) / pow((double)BLOCKSIZE, 2));
    int gridsize = (int)ceil(sqrt(num));

    dim3 dimBlock(BLOCKSIZE, BLOCKSIZE, 1);
    dim3 dimGrid(gridsize, gridsize, 1);

    multiply<<<dimGrid, dimBlock>>>(device_a, device_b, device_c, n);
    cudaThreadSynchronize();

    cudaMemcpy(host_c, device_c, sizeof(int) * n * n, cudaMemcpyDeviceToHost);

    gettimeofday(&finish, 0);

    double t = 1000000 * (finish.tv_sec - start.tv_sec) + finish.tv_usec - start.tv_usec;
    printf("%lf ms\n", t / 1000);
//GPU parallel finish


//CPU serial start
    gettimeofday(&start, 0);

    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            host_c2[i * n + j] = 0;
            for (k = 0; k < n; k++) {
                host_c2[i * n + j] += host_a[i * n + k] * host_b[k * n + j];
            }
        }
    }

    gettimeofday(&finish, 0);

    t = 1000000 * (finish.tv_sec - start.tv_sec) + finish.tv_usec - start.tv_usec;
    printf("%lf ms\n", t / 1000);
//CPU serial start

//check
    int errorNum = 0;
    for (int i = 0; i < n * n; i++) {
        if (host_c[i] != host_c2[i]) {
            errorNum ++;
            printf("Error occurs at index: %d: c = %d, c2 = %d\n", i, host_c[i], host_c2[i]);
        }
    }
    if (errorNum == 0) {
        printf("Successfully run on GPU and CPU!\n");
    } else {
        printf("%d error(s) occurs!\n", errorNum);
```

```
    }

    free(host_a);
    free(host_b);
    free(host_c);
    free(host_c2);

    cudaFree(device_a);
    cudaFree(device_b);
    cudaFree(device_c);

    return 0;
}
```

- 46
- 47
- 48
- 49
- 50
- 51
- 52
- 53
- 54
- 55
- 56
- 57
- 58
- 59
- 60
- 61
- 62
- 63
- 64
- 65
- 66
- 67
- 68
- 69
- 70
- 71
- 72
- 73
- 74
- 75
- 76
- 77
- 78
- 79
- 80
- 81
- 82
- 83
- 84
- 85
- 86
- 87
- 88
- 89
- 90
- 91
- 92
- 93
- 94
- 95
- 96
- 97
- 98
- 99
- 100
- 101
- 102
- 103

- 104
- 105
- 106
- 107
- 108
- 109
- 110
- 111
- 112
- 113
- 114
- 115
- 116
- 117
- 118
- 119
- 120
- 121
- 122
- 123
- 124
- 125
- 126
- 127
- 128
- 129