CUDA 编程简介: 基础与实践

李瑜 (liyu@tjufe.edu.cn)

February 2, 2023

Abstract

课程主要内容:

1 GPU 硬件与 CUDA 程序开发简介

- `nvidia-smi`; `nvcc`
- SP, SM; grid, block, thread, warp

2 CUDA 中的 Hello World 程序

- `dim3`; `gridDim`, `blockDim`; `blockIdx`, `threadIdx`
- `cudaMalloc`, `cudaFree`; `cudaMemcpy`; `cudaDeviceSynchronize()`
- 函数修饰符: `__host__`; `__global__`; `__device__`
- 变量修饰符: `__device__`; `__constant__`; `__shared__`
- `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`; `__syncthreads()`; `atomic...`

3 CUDA 程序的错误与性能检测

- `cudaError_t`; `cuda-memcheck`
- `cudaEvent...`; Nsight System

4 CUDA 标准库的使用

- cuBLAS, cuSPARSE; cuSolver
- cuRAND; cuFFT
- Thrust

5 三维玻色–爱因斯坦凝聚 (BEC) 数值模拟的 CUDA 程序开发

i BEC 动力学模拟

ii BEC 基态解计算

课程时间:

- 第一次课 (2023.02.01, 15:00-16:30): 1, 2
- 第二次课 (2023.02.03, 15:00-16:30): 3, 4
- 第三次课 (2023.02.06, 15:00-16:30): 上机实践
- 第四次课 (2023.02.08, 15:00-16:30): 5.i
- 第五次课 (2023.02.10, 15:00-16:30): 5.ii

参考资料:

- NVIDIA 官方文档. `https://docs.nvidia.com/cuda/`
- Cook, Shane. CUDA 并行程序设计 – GPU 编程指南, 机械工业出版社, 2014.
- 樊哲勇. CUDA 编程: 基础与实践, 清华大学出版社, 2020.
- ...

Contents

1 GPU 硬件与 CUDA 程序开发简介

Graphic Processing Unit (GPU), Compute Unified Device Architecture (CUDA)

显卡的架构对显卡的性能有很大的影响. 目前, 显卡架构性能的排序如下:

Table 1: NVIDIA Architecture

Tesla	Fermi	Kepler	Maxwell	Pascal	Volta	Turing	Ampere	Hopper
2006	2010	2012	2014	2016	2018	2018	2020	2022

https://docs.nvidia.com/cuda/

1.1 安装

Ubuntu 18.04 安装显卡驱动与 CUDA:

```
# 卸载原有驱动, 再更新驱动
sudo apt-get remove --purge nvidia*
ubuntu-drivers devices
sudo apt-get install nvidia-driver-xxx
sudo reboot
nvidia-smi
sudo apt-get install nvidia-cuda-toolkit
nvcc
sudo reboot
```

`nvidia-smi`是 NVIDIA 的系统管理界面, 其中 smi 是 System Management Interface 的缩写. 它可以收集各种级别的信息, 查看显存使用情况以及启用和禁用 GPU 配置选项.

`nvcc`是编译 CUDA 程序的编译器, 其源文件以 cu 为后缀.

1.2 基本概念

- SP (streaming processor): 最基本的处理单元, 也称为 CUDA core. 具体的指令和任务都是在 SP 上处理的. GPU 进行并行计算, 也就是很多个 SP 同时做处理.

```
Wed Feb  1 11:12:15 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.161.03   Driver Version: 470.161.03   CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA GeForce ...  Off  | 00000000:73:00.0 Off |                  N/A |
| 38%   74C    P2   234W / 250W |   6998MiB / 11016MiB |    100%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A     65250      C   ./a.out                         6995MiB |
+-----------------------------------------------------------------------------+
```

- SM (streaming multiprocessor): 多个 SP 加上其它的一些资源组成一个 SM, 也叫 GPU 大核. 其它资源如: warp scheduler, register, shared memory 等.

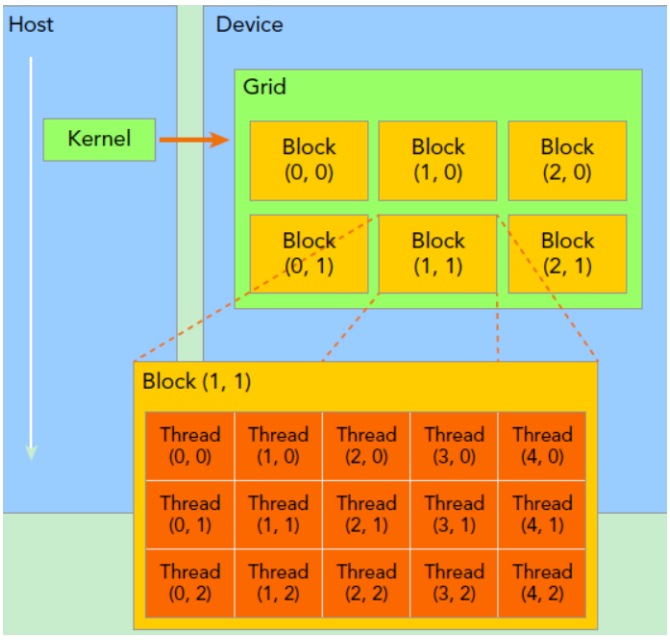
SP (streaming Process), SM (streaming multiprocessor) 是硬件 (GPU) 概念, 一个 SM 可以包含多个 SP, 每个 SM 包含的 SP 数量依据 GPU 架构而不同.

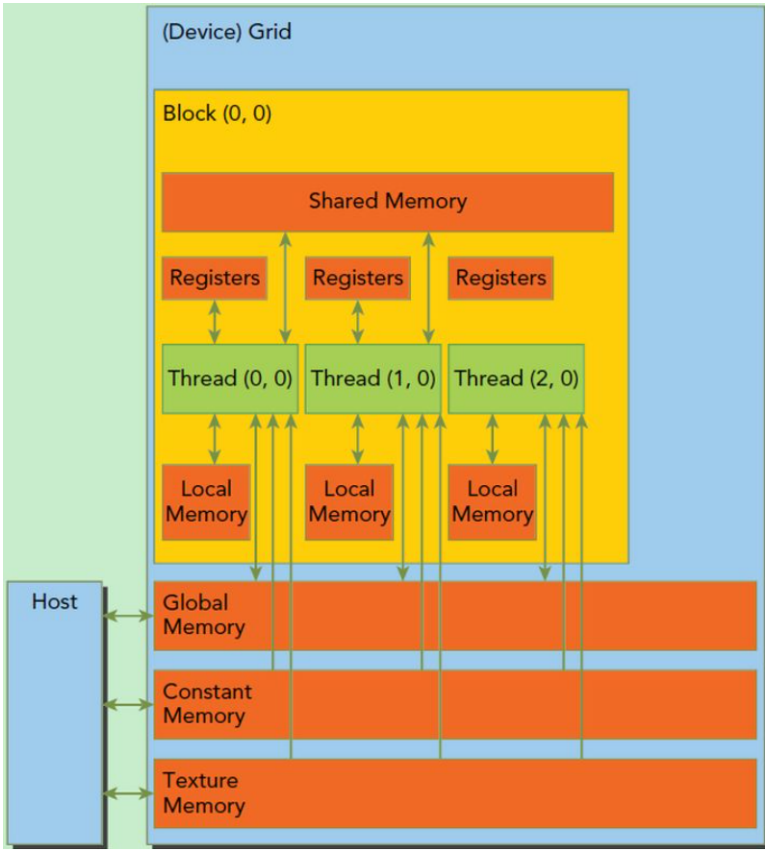
Table 2: NVIDIA Architecture

Tesla	Fermi	Kepler	Maxwell	Pascal	Volta	Turing	Ampere	Hopper
8	32	192	128	64	64	64	64	128

thread, block, grid, warp 是软件上的 (CUDA) 概念. 为了方便程序员软件设计、组织线程, CUDA 的软件架构由网格 (Grid)、线程块 (Block) 和线程 (Thread) 组成. 相当于把 GPU 上的计算单元分为若干 (2 或 3) 个网格, 每个网格内包含若干个线程块, 每个线程块包含若干个线程.

- thread: 一个 CUDA 的并行程序会被以许多个 threads 来执行.
- block: 数个 threads 会被组成一个 block, 同一个 block 中的 threads 可以同步, 也可以通过 shared memory 通信.
- grid: 多个 blocks 则会再构成 grid.
- warp: 把 32 个 threads 组成一个 warp. warp 是调度和运行的基本单元. warp 中所有 threads 并行的执行相同的指令. 一个 warp 需要占用一个 SM 运行. 所以, 一个 GPU 上 resident thread 最多只有 SM*warp 个.





在 GPU 上调用的函数成为 CUDA 核函数 (Kernel Function), 核函数会被 GPU 上的多个线程执行. 在实际执行 kernal 时, 会以 block 为单位, 把一个个 block 分配

给 SM 进行运算. block 中的 thread 又会以 warp 为单位, 对 thread 进行分组计算. 也就是说 32 个 thread 会被组成一个 warp 来一起执行. 同一个 warp 中的 thread 执行的指令是相同的, 只是处理的数据不同.

CUDA 通过 block 这个概念, 提供了细粒度的通信手段, 因为 block 是加载在 SM 上运行的, 所以可以利用 SM 提供的 shared memory 和`__syncthreads()` 功能实现线程同步和通信. 而 block 之间, 除了结束 kernel 之外是无法同步的, 一般也不保证运行先后顺序, 这是因为 CUDA 程序要保证在不同规模 (不同 SM 数量) 的 GPU 上都可以运行, 必须具备规模的可扩展性, 因此 block 之间不能有依赖. 这就是 CUDA 的两级并行结构.

一个 block 只会由一个 SM 调度, block 一旦被分配到某个 SM, 该 block 就会一直驻留在该 SM 中, 直到执行结束. 一个 SM 可以同时拥有多个 blocks, 但需要序列执行.

大部分 threads 只是逻辑上并行, 并不是所有的 thread 可以在物理上同时执行. 这就导致同一个 block 中的线程可能会有不同步调. 另外, 并行 thread 之间的共享数据会导致竞态, 即多个线程请求同一个数据会导致未定义行为.

同一个 warp 中的 thread 可以以任意顺序执行, active warps 被 SM 资源限制. 当一个 warp 空闲时, SM 就可以调度驻留在该 SM 中另一个可用 warp. 在并发的 warp 之间切换是没什么消耗的, 因为硬件资源早就被分配到所有 thread 和 block, 所以该新调度的 warp 的状态已经存储在 SM 中了. CPU 切换线程需要保存/读取线程上下文 (register 内容), 这是非常耗时的, 而 GPU 为每个 threads 提供物理 register, 无需保存/读取上下文.

当一个 warp 中的线程顺序执行判断语句中的不同分支时, 称发生了分支分散. 我们应该尽量避免这种情形的发生.

```
1 #include "cuda_runtime.h"
2 int dev;
3 cudaDeviceProp deviceProp;
4 cudaGetDeviceProperties(&deviceProp, dev);
```

1.3 CUDA 内置变量

- `dim3`: 仅在 host 端可见, 是基于 uint3 的整数矢量类型.
- `gridDim`: dim3 类型的 build-in 变量, 表示 grid 的大小, 以 block 为单位.
- `blockDim`: dim3 类型的 build-in 变量, 表示 block 的大小, 以 thread 为单位.

```
---------------------------------------
Devices Information
There is 1 device supporting CUDA.

 Device 0:"NVIDIA GeForce RTX 2080 Ti"
Major revision number: 7
Minor revision number: 5
Total amount of global memory: 11544035328 bytes
Number of multiprocessors: 68
Number of cores: 64*68
Total amount of constant memory: 65536 bytes
Total amount of shared memory per block: 49152 bytes
Total number of registers available per block: 65536
Warp size: 32
Maximum number of threads per block: 1024
Maximum sizes of each dimension of a block: 1024 x 1024 x 64
Maximum sizes of each dimension of a grid: 2147483647 x 65535 x 65535
Maximum memory pitch: 2147483647 bytes
Texture alignment: 512 bytes
Clock rate: 1.54 GHz
Concurrent copy and execution: Yes
---------------------------------------
```

Figure 1: RTX 2080Ti 设备属性

- `blockIdx`: uint3 类型的 build-in 变量, 表示在 grid 内的一个 block 的索引.
- `threadIdx`: uint3 类型的 build-in 变量, 表示在 block 内一个 thread 的索引.

1D grid of 1D blocks

```
__device__
int getGlobalIdx_1D_1D()
{
    return blockIdx.x *blockDim.x + threadIdx.x;
}
```

1D grid of 2D blocks

```
__device__
int getGlobalIdx_1D_2D()
{
    return blockIdx.x * blockDim.x * blockDim.y
     + threadIdx.y * blockDim.x + threadIdx.x;
}
```

2D grid of 1D blocks

```cpp
__device__
int getGlobalIdx_2D_1D()
{
    int blockId = blockIdx.y * gridDim.x + blockIdx.x;
    int threadId = blockId * blockDim.x + threadIdx.x;
    return threadId;
}
```

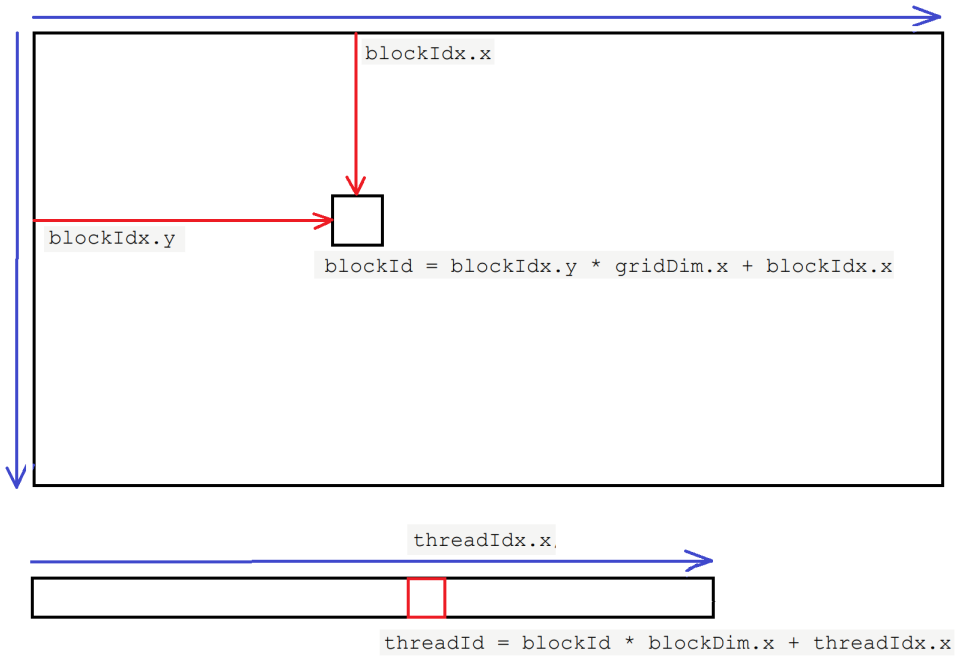


Figure 2: 2D grid of 1D blocks

1.4 CUDA 编程

CUDA 的操作概括来说包含 6 个步骤:

- CPU 在 GPU 上分配内存: `cudaMalloc`
- CPU 把数据发送到 GPU: `cudaMemcpy`
- CPU 在 GPU 上启动 kernel, 它是自己写的一段程序, 在每个线程上运行

- CPU 等待 GPU 端完成之前 CUDA 的任务: `cudaDeviceSynchronize`
- CPU 把数据从 GPU 取回: `cudaMemcpy`
- CPU 释放 GPU 上的内存: `cudaFree`

一个 kernel 的调用为:

```
Kernel<<<dimGrid, dimBlock>>>(param1, param2, ...);
```

并通过线程编号进行编程.

三种前缀分别用于在定义函数时, 限定该函数的调用和执行方式:

- `__host__ int foo(int a){}`

  是由 CPU 调用, 由 CPU 执行的函数. 它与 C/C++ 中的`int foo(int a){}`相同.

- `__global__ int foo(int a){}`

  是由`__host__`函数以`foo<<<dimGrid,dimBlock>>>(a)`的形式或者 driver API 的形式调用, 在 GPU 上执行的核函数.

- `__device__ int foo(int a){}`

  是由`__global__`函数或`__device__`函数调用, 在 GPU 中一个线程上执行的函数. 实际上, `__device__`函数是以`__inline`形式展开后直接编译到二进制代码中实现的, 并不是真正的函数.

三种前缀分别用于在定义变量时, 限定该变量的声明和访问方式:

- `__device__`: 设备端的全局变量. `__device__`函数和`__global__`函数可对其进行读写访问.
- `__constant__`: 设备端的常量. `__device__`函数和`__global__`函数可对其进行只读访问.
- `__shared__`: 块内共享变量. 只能在`__device__`函数或者`__global__`函数内被声明. 同一个线程块中的不同线程可对其进行读写访问.

对于`__device__`变量和`__constant__`变量, 主机端可以通过`cudaMemcpyToSymbol`以及`cudaMemcpyFromSymbol`函数传递或者获取它们的值. 对于`__shared__`变量, `__global__`函数通过`__syncthreads()`对线程块中的每个线程进行同步.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

const int threadsPerBlock = 4;
const int blocksPerGrid   = 2;
__device__
int getGlobalIdx_1D_1D()
{
    return blockIdx.x * blockDim.x + threadIdx.x;
}
__global__
void dot(float *a, float *b, float *c, int N)
{
    __shared__ float cache[threadsPerBlock];
    int tid = getGlobalIdx_1D_1D();
    int cacheIndex = threadIdx.x;
    float temp = 0;
    while(tid < N) {
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;
    }
    cache[cacheIndex] = temp;
    //对线程块中的线程进行同步
    __syncthreads();
    int flag = blockDim.x%2;
    int i = blockDim.x/2;
    while(i != 0) {
        if(cacheIndex < i) {
            cache[cacheIndex] += cache[cacheIndex + i];
        }
        if (flag == 1 && cacheIndex == 0) {
            cache[cacheIndex] += cache[2*i];
        }
        flag = i%2; i /= 2;
```

```
36          __syncthreads();
37      }
38      if (cacheIndex == 0) {
39          c[blockIdx.x] = cache[0];
40      }
41  }
42  int main(int argc, char *argv[])
43  {
44      int i, N = 14;
45      float *a, *b, c, *partial_c;
46      float *dev_a, *dev_b, *dev_partial_c;
47      //在CPU上面分配内存
48      a = (float*)malloc(N*sizeof(float));
49      b = (float*)malloc(N*sizeof(float));
50      partial_c = (float*)malloc(blocksPerGrid*sizeof(float));
51      //在GPU上分配内存
52      cudaMalloc((void**)&dev_a,N*sizeof(float));
53      cudaMalloc((void**)&dev_b,N*sizeof(float));
54      cudaMalloc((void**)&dev_partial_c,blocksPerGrid*sizeof(float));
55      //填充主机内存
56      for(i = 0; i < N; i++) {
57          a[i] = 1; b[i] = -1;
58      }
59      //将数组 a 和 数组 b 复制到GPU
60      cudaMemcpy(dev_a,a,N*sizeof(float),cudaMemcpyHostToDevice);
61      cudaMemcpy(dev_b,b,N*sizeof(float),cudaMemcpyHostToDevice);
62      dot<<<blocksPerGrid, threadsPerBlock>>>(
63          dev_a, dev_b, dev_partial_c);
64      cudaDeviceSynchronize();
65      //将数组 dev_partial_c 从 GPU 复制到 CPU
66      cudaMemcpy(partial_c,dev_partial_c,
67          blocksPerGrid*sizeof(float),cudaMemcpyDeviceToHost);
68      //在CPU上完成最终的求和运算
69      c = 0.0;
70  #pragma omp parallel for reduction(+:c) num_threads(2)
71      for(i = 0; i < blocksPerGrid; i++) {
```

```
        c += partial_c[i];
    }
    printf("%s\n", "==========================================");
    for(i = 0; i < N; i++) {
        printf("%f  %f  \n", a[i], b[i]);
    }
    printf("c = %f  \n", c);
    printf("blocksPerGrid   %d  \n", blocksPerGrid);
    printf("threadsPerBlock %d  \n", threadsPerBlock);
    // 释放 GPU 上的内存
    cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_partial_c);
    // 释放 CPU 上的内存
    free(a); free(b); free(partial_c);
    return 0;
}
```

```
nvcc -Xcompiler -fopenmp main.cu （切勿复制）
```

多文件时, 可以考虑利用nvcc编译 cu 文件, gcc编译 c 文件再使用gcc以及-lcudart将二者链接在一起.

Remark 1.1. 本小节中示例程序配有动画展示, 请查看 *bilibi: OpenMP, CUDA, MPI* 并行架构.

2 CUDA 程序的错误检测

2.1 检查 API 函数

```
#define CHECK(call)                                   \
do                                                    \
{                                                     \
    const cudaError_t error = call;                   \
    if (error != cudaSuccess)                         \
    {                                                 \
        printf("CUDA Error\n");                       \
        printf("\t File       : %s\n", __FILE__);     \
        printf("\t Line       : %d\n", __LINE__);     \
        printf("\t Error Code: %d\n", error);         \
        printf("\t Error Text: %s\n",                  \
               cudaGetErrorString(error));            \
        exit(1);                                      \
    }                                                 \
} while(0)
```

2.2 检查核函数

在调用核函数之后加上如下两条语句:

```
CHECK(cudaGetLastError());
CHECK(cudaDeviceSynchronize());
```

2.3 检查内存错误

```
cuda-memcheck ./a.out
```

https://docs.nvidia.com/pdf/CUDA_Memcheck.pdf

2.4 程序调试

- `nvcc -g -G -arch=compute_70 -code=sm_70 main.cu -o a.out`
- `cuda-gdb ./a.out`
- `set cuda memcheck on`
- `break main;break main.cu:185;delete`
- `run;next;contimue`
- `list;focus`
- `info locals;info args;`
- `backtrace;where;frame`
- `cuda device block thread;cuda thread (15);cuda block 1 thread 3`
- `print array[0]@4`
- `info cuda warps lanes blocks threads breakpoint all`

`https://docs.nvidia.com/cuda/cuda-gdb/`

3 CUDA 程序的性能检测

3.1 用 CUDA 事件计时

```
cudaEvent_t start, stop;

cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);
cudaEventQuery(start);

// 需要技时的代码块

cudaEventRecord(stop);
cudaEventSynchronize(stop);
float elapsed_time;
cudaEventElapsedTime(&elapsed_time, start, stop);
printf("Time = %g ms.\n", elapsed_time);
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

3.2 用 Nsight 分析性能

```
nsys profile <application> [application-arguments]
nsys stats report1.nsys-rep
```

https://docs.nvidia.com/nsight-systems/

4 CUDA 标准库的使用

Table 3: 一些 CUDA 库

库名	简介
Thrust	类似于 C++ 的标准模板库
cuRAND	随机数生成器
cuBLAS	基本线性代数子程序
cuSPARSE	稀疏矩阵
cuSOLVER	稠密矩阵核稀疏矩阵线性代数库
cuFFT	快速傅里叶变换

4.1 cuBLAS

https://docs.nvidia.com/cuda/cublas/

```
cublasStatus_t cublasSscal(cublasHandle_t handle, int n,
    const float *alpha, float *x, int incx);
cublasStatus_t cublasSetMatrix(int rows, int cols, int elemSize,
    const void *A, int lda, void *B, int ldb);
cublasStatus_t cublasGetMatrix(int rows, int cols, int elemSize,
    const void *A, int lda, void *B, int ldb);
```

```
//Application Using C and cuBLAS: 0-based indexing
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "cublas_v2.h"

#define LDM 6
#define N 8
#define IDX2C(i, j, ld) (((j)*(ld))+(i))

static __inline__
void modify(cublasHandle_t handle,
```

```
13      float *m, int ldm, int n,
14      int p, int q, float alpha, float beta)
15  {
16      cublasSscal(handle, n-q, &alpha, &m[IDX2C(p, q, ldm)], ldm);
17      cublasSscal(handle, ldm-p, &beta, &m[IDX2C(p, q, ldm)], 1);
18  }
19  int main(int argc, char *argv[])
20  {
21      cublasHandle_t handle;
22      int i, j;
23      float* devPtrA;
24      float* a = 0;
25      a = (float *)malloc(LDM*N*sizeof(*a));
26      for (j = 0; j < N; j++) {
27          for (i = 0; i < LDM; i++) {
28              a[IDX2C(i, j, LDM)] = (float)(i * N + j + 1);
29          }
30      }
31      printf("%s%dx%d\n", "==C version==================", N, LDM);
32      for (j = 0; j < N; j++) {
33          for (i = 0; i < LDM; i++) {
34              printf("%7.2f(%2d)",
35                  a[IDX2C(i, j, LDM)], IDX2C(i, j, LDM));
36          }
37          printf("\n");
38      }
39      printf("%s%dx%d\n", "==F version==================", LDM, N);
40      for (i = 0; i < LDM; i++) {
41          for (j = 0; j < N; j++) {
42              printf ("%7.2f(%2d)",
43                  a[IDX2C(i, j, LDM)], IDX2C(i, j, LDM)+1);
44          }
45          printf("\n");
46      }
47      cudaMalloc((void**)&devPtrA, LDM*N*sizeof(*a));
48      cublasCreate(&handle);
```

```
    cublasSetMatrix(LDM, N, sizeof(*a), a, LDM, devPtrA, LDM);
    modify(handle, devPtrA, LDM, N, 1, 2, 2.0f, 0.5f);
    cublasGetMatrix(LDM, N, sizeof(*a), devPtrA, LDM, a, LDM);
    cudaFree(devPtrA);
    cublasDestroy(handle);
    printf("%s%dx%d\n", "==F version==========", LDM, N);
    for (i = 0; i < LDM; i++) {
        for (j = 0; j < N; j++) {
            printf("%7.2f(%2d)",
                a[IDX2C(i, j, LDM)], IDX2C(i, j, LDM)+1);
        }
        printf("\n");
    }
    printf("%s\n", "==================================");
    free(a);
    return 0;
}
```

4.2 cuFFT

The fast Fourier transform (FFT) is a discrete Fourier transform algorithm which reduces the number of computations needed for N points from $2N^2$ to $2N \log_2 N$.

For $k_1 = 0, \ldots, n_1 - 1$ and $k_2 = 0, \ldots, n_2 - 1$,

$$z[k_1][k_2] = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} w[j_1][j_2] \exp\left(\delta 2\pi i \cdot \left(\frac{j_1 k_1}{n_1} + \frac{j_2 k_2}{n_2}\right)\right)$$

where $\delta = -1$ for the forward transform, and $\delta = +1$ for the inverse (backward) transform.

Remark 4.1. *FFTW 和 cuFFT 都是没有乘以标准化系数. 下述两个式子等价:*

$$z[k_1][k_2] = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} w[j_1][j_2] \exp\left(+2\pi i \cdot \left(\frac{j_1 k_1}{n_1} + \frac{j_2 k_2}{n_2}\right)\right),$$

$$(n_1 n_2) w[j_1][j_2] = \sum_{k_1=0}^{n_1-1} \sum_{k_2=0}^{n_2-1} z[k_1][k_2] \exp\left(-2\pi i \cdot \left(\frac{j_1 k_1}{n_1} + \frac{j_2 k_2}{n_2}\right)\right),$$

其中 w 是频域, z 是时域.

https://docs.nvidia.com/cuda/cufft/

```
#define NX 64
#define NY 128
#define NZ 128
#define BATCH 10
#define NRANK 3

cufftHandle plan;
cufftComplex *data;
int n[NRANK] = {NX, NY, NZ};
cudaMalloc((void**)&data, sizeof(cufftComplex)*NX*NY*NZ*BATCH);
/* Create a 3D FFT plan. */
cufftPlanMany(&plan, NRANK, n,
    NULL, 1, NX*NY*NZ, // *inembed, istride, idist
    NULL, 1, NX*NY*NZ, // *onembed, ostride, odist
    CUFFT_C2C, BATCH);
/* Use the CUFFT plan to transform the signal in place. */
cufftExecC2C(plan, data, data, CUFFT_FORWARD);
cudaDeviceSynchronize();
...
cufftExecC2C(plan, data, data, CUFFT_INVERSE);
cudaDeviceSynchronize();
...
cufftDestroy(plan);
cudaFree(data);
```

```
cufftResult cufftMakePlan3d(cufftHandle plan,
    int nx, int ny, int nz, cufftType type, size_t *workSize);
cufftResult cufftSetWorkArea(cufftHandle plan, void *workArea);
```