CUDA 编程简介: 基础与实践

李瑜 (liyu@tjufe.edu.cn)

February 10, 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`;
- 函数修饰符: `__host__`; `__global__`; `__device__`
- 变量修饰符: `__device__`; `__constant__`; `__shared__`
- `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`; `atomic...`
- `cudaDeviceSynchronize`; `__syncthreads`; `__syncwarp`

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) 动力学数值模拟的 CUDA 程序开发

 i 数值方法

 ii 程序实现

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表1.1所示:

Table 1.1: 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. 如图1.1所示, 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.

https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/

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

```
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 |
+-----------------------------------------------------------------------------+
```

Figure 1.1: NVIDIA 的系统管理界面

理.

- 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, 如表1.2.

Table 1.2: 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 grid 包含若干个 blocks, 每个 block 包含若干个 threads. 图1.2展示了 grid 内包含 $3 * 2$ 个 blocks, 每个 block 包含 $4 * 3$ 个 threads. 图1.2展示了 grid 内包含 $3 * 2$ 个 blocks, 每个 block 包含 $4 * 3$ 个 threads.

- 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.

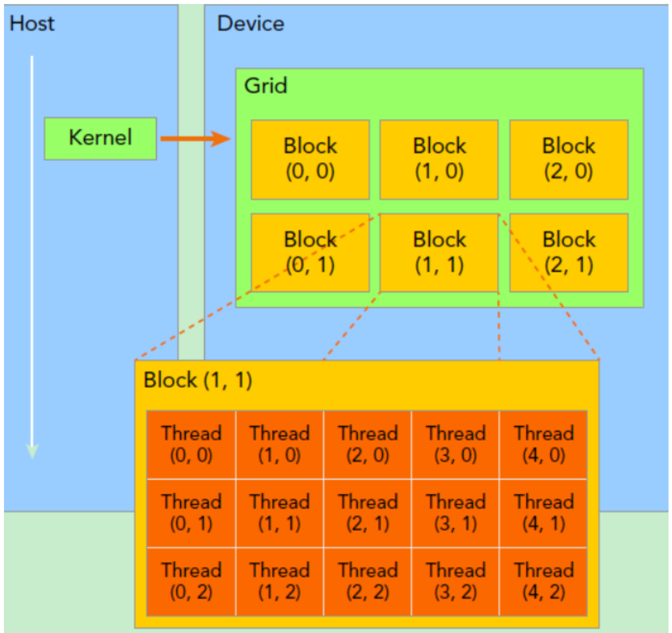

Figure 1.2: grid, block, thread

- 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*32 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. 图1.2给出某个 block 内的 thread 的具体结构.

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.

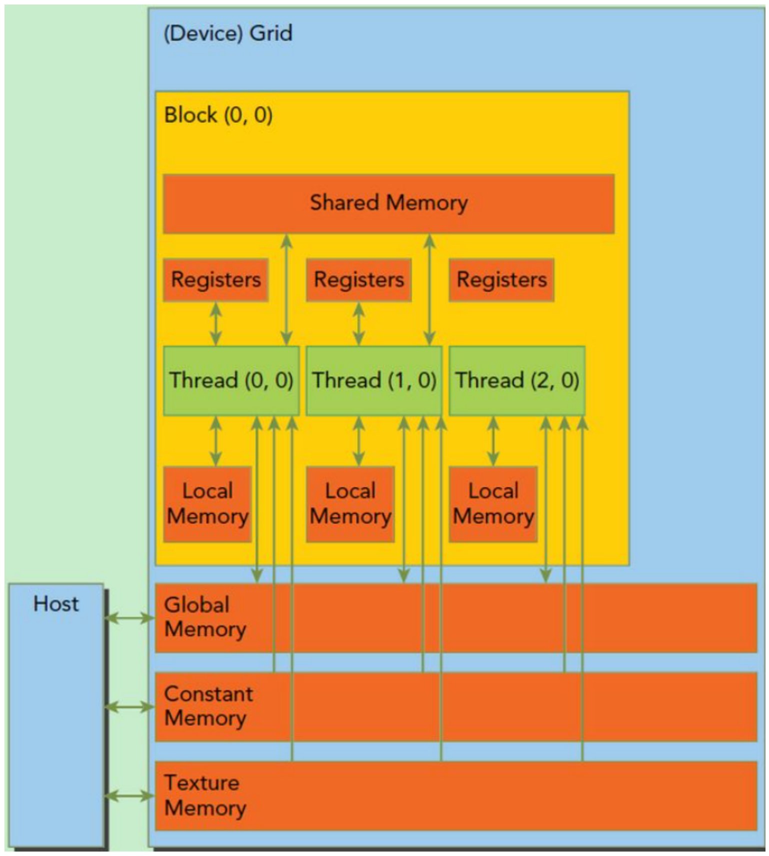

Figure 1.3: GPU 上的内存

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.

```
#include "cuda_runtime.h"
int dev = 0;
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp, dev);
```

```
---------------------------------------
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 1.4: RTX 2080Ti 设备属性

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.
- `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

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

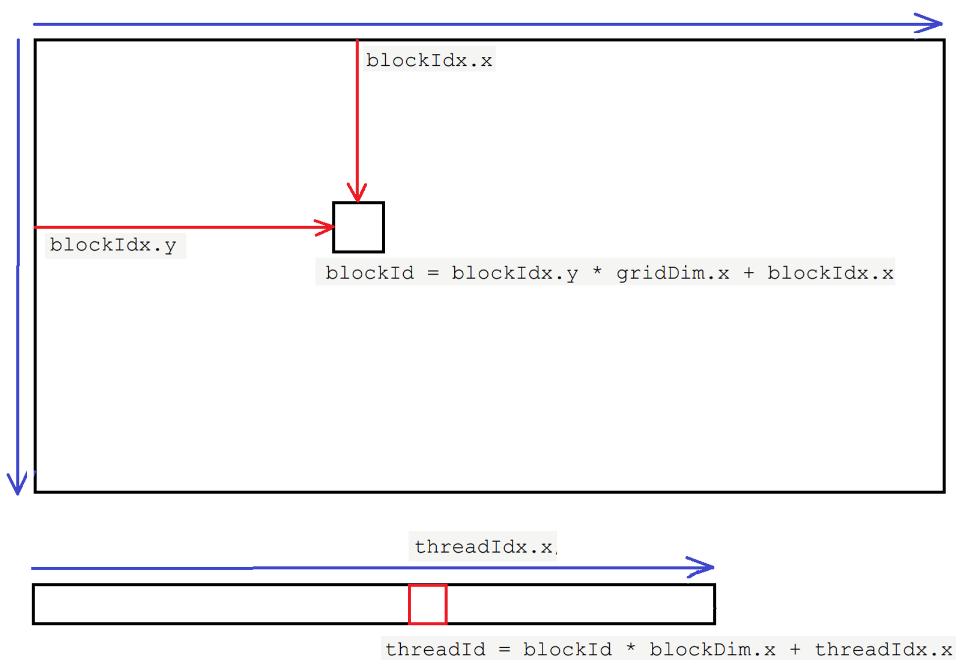

Figure 1.5: 2D grid of 1D blocks

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.

忽略块内/束内线程的同步, 我们可以将 CUDA 程序简单理解为如下形式:

```
warpSize = 32;
numWarp = blockDim / warpSize;
for(blockIdx = 0; blockIdx < gridDim; blockIdx++) {
    threadIdx = 0;
    for(warpIdx = 0; warpIdx < numWarp; warpIdx++) {
        for(laneIdx = 0; laneIdx < warpSize; laneIdx++) {
            //////////////
            //核函数部分//
            //////////////
            threadIdx++;
        }
    }
}
```

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;
        __syncthreads();
    }
    if (cacheIndex == 0) {
        c[blockIdx.x] = cache[0];
    }
}
int main(int argc, char *argv[])
{
    int i, N = 14;
    float *a, *b, c, *partial_c;
    float *dev_a, *dev_b, *dev_partial_c;
    //在CPU上面分配内存
```

```
    a = (float*)malloc(N*sizeof(float));
    b = (float*)malloc(N*sizeof(float));
    partial_c = (float*)malloc(blocksPerGrid*sizeof(float));
    //在GPU上分配内存
    cudaSetDevice(0);
    cudaMalloc((void**)&dev_a,N*sizeof(float));
    cudaMalloc((void**)&dev_b,N*sizeof(float));
    cudaMalloc((void**)&dev_partial_c,blocksPerGrid*sizeof(float));
    //填充主机内存
    for(i = 0; i < N; i++) {
        a[i] = 1; b[i] = -1;
    }
    //将数组 a 和 数组 b 复制到GPU
    cudaMemcpy(dev_a,a,N*sizeof(float),cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b,b,N*sizeof(float),cudaMemcpyHostToDevice);
    dot<<<blocksPerGrid, threadsPerBlock>>>(
        dev_a, dev_b, dev_partial_c);
    cudaDeviceSynchronize();
    //将数组 dev_partial_c 从 GPU 复制到 CPU
    cudaMemcpy(partial_c,dev_partial_c,
        blocksPerGrid*sizeof(float),cudaMemcpyDeviceToHost);
    //在CPU上完成最终的求和运算
    c = 0.0;
#pragma omp parallel for reduction(+:c) num_threads(2)
    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`; `finish`
- `list`; `focus`
- `info locals`; `info args`;
- `frame`; `backtrace`; `where`
- `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 4.1: 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,
```

```
    float *m, int ldm, int n,
    int p, int q, float alpha, float beta)
{
    cublasSscal(handle, n-q, &alpha, &m[IDX2C(p, q, ldm)], ldm);
    cublasSscal(handle, ldm-p, &beta, &m[IDX2C(p, q, ldm)], 1);
}
int main(int argc, char *argv[])
{
    cublasHandle_t handle;
    int i, j;
    float* devPtrA;
    float* a = 0;
    a = (float *)malloc(LDM*N*sizeof(*a));
    for (j = 0; j < N; j++) {
        for (i = 0; i < LDM; i++) {
            a[IDX2C(i, j, LDM)] = (float)(i * N + j + 1);
        }
    }
    printf("%s%dx%d\n", "==C version==================", N, LDM);
    for (j = 0; j < N; j++) {
        for (i = 0; i < LDM; i++) {
            printf("%7.2f(%2d)",
                a[IDX2C(i, j, LDM)], IDX2C(i, j, LDM));
        }
        printf("\n");
    }
    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");
    }
    cudaMalloc((void**)&devPtrA, LDM*N*sizeof(*a));
    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 $k_2 = 0, \ldots, n_2 - 1$,

$$w[k_1][k_2] = \sum_{j_1=0}^{n_1-1}\sum_{j_2=0}^{n_2-1} z[j_1][j_2]\exp\Big(\delta 2\pi\mathbf{i}\cdot(\frac{j_1k_1}{n_1}+\frac{j_2k_2}{n_2})\Big)$$

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:*

$$z[j_1][j_2] = \sum_{k_1=0}^{n_1-1}\sum_{k_2=0}^{n_2-1} w[k_1][k_2]\exp\Big(+2\pi\mathbf{i}\cdot(\frac{j_1k_1}{n_1}+\frac{j_2k_2}{n_2})\Big),$$

$$(n_1n_2)w[k_1][k_2] = \sum_{j_1=0}^{n_1-1}\sum_{j_2=0}^{n_2-1} z[j_1][j_2]\exp\Big(-2\pi\mathbf{i}\cdot(\frac{j_1k_1}{n_1}+\frac{j_2k_2}{n_2})\Big),$$

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

接下来我们针对不同的并行方式, 对 FFT 做一个简单的性能测试.

- 测试环境:

- Intel(R) Xeon(R) Gold 6248R CPU @ 3.00GHz, 逻辑核数 2*24*2, 内存 500GB
- GeForce RTX 2080 Ti, 流处理器簇 (SM) 数量 68, 显存 11016MB

• 测试例子: FFT 3D 512*512*512, 数据类型是复的双精度浮点型.

在表4.2中, 第一列是使用的线程数或进程数. 每个行块的第一行是 plan 的时间, 第二行是执行一次正变换和一次逆变换的总时间. 利用 cuFFT, plan 时间为 0.15 秒, 执行时间为 0.13 秒.

Table 4.2: 性能测试

	fftw+mpicc	fftw+gcc+OpenMP	fttw+icc+OpenMP	Intel MKL
2	97.58	51.60	54.00	0.06
	5.08	2.95	2.95	3.20
4	63.17	45.78	51.75	0.06
	2.52	2.76	3.21	1.69
8	44.78	40.85	43.11	0.06
	1.45	2.81	3.17	1.00
16	53.33	38.20	40.04	0.06
	1.28	3.02	2.93	0.72

5 BEC 动力学模拟

5.1 模型描述

对于 $(x,t)\in\mathbb{R}^3\times(0,T]$,

$$\mathbf{i}\frac{\partial\varphi}{\partial t}(x,t)=\big(-\varepsilon\Delta+V(x)+\beta|\varphi(x,t)|^2+\lambda\Phi(x,t)-\omega L_z\big)\varphi(x,t),$$

其中, $\varepsilon=0.5$, $\gamma=0.5$,

$$\begin{aligned}&x=(x_1,x_2,x_3),\ V(x)=\gamma(x_1^2+x_2^2+x_3^2),\\&\Phi(x,t)=U_{\text{dip}}*|\varphi|^2=\int_{\mathbb{R}^3}U_{\text{dip}}(x-\tilde{x})|\varphi(\tilde{x},t)|^2\,d\tilde{x},\\&L_z=-\mathbf{i}(x_1\partial_{x_2}-x_2\partial_{x_1}).\end{aligned}$$

5.2 解析结果

将初始波函数赋值为

$$\varphi(x,0)=\sqrt{\frac{1}{(\sqrt{2\pi})^3}\exp\big(-\frac{1}{2}(x_1^2+x_2^2+x_3^2)\big)}$$

那么

$$\begin{aligned}&\int_{\mathbb{R}^3}\frac{1}{2}|\nabla\varphi|^2\,dx=\frac{3}{8},\ \int_{\mathbb{R}^3}V|\varphi|^2\,dx=\frac{3}{2},\ \int_{\mathbb{R}^3}\frac{\beta}{2}|\varphi|^4\,dx=\frac{\beta}{2}\frac{1}{8\sqrt{\pi^3}},\\&\int_{\mathbb{R}^3}\varphi^*L_z\varphi\,dx=\int_{\mathbb{R}^3}\varphi^*(-\mathbf{i})(x_1\partial_{x_2}-x_2\partial_{x_1})\varphi\,dx=0.\end{aligned}$$

其中, $\dfrac{\partial\varphi}{\partial x_i}=-\dfrac{x_i}{2}\varphi$, $i=1,2,3$. 取 $U_{\text{dip}}(x)=1$, 则 $\Phi(x)=1$.

当 $\beta=-2,\lambda=2$ 时,

$$M(\varphi)=1,\ E(\varphi)\approx 2.852551609734354176595.$$

其中, 总能量表达式

$$E(\varphi)=\int_{\mathbb{R}^3}\Big[\frac{1}{2}|\nabla\varphi|^2+V|\varphi|^2+\frac{\beta}{2}|\varphi|^4+\frac{\lambda}{2}\Phi|\varphi|^2-\omega\varphi^*L_z\varphi\Big]\,dx,$$

总质量表达式

$$M(\varphi)=\int_{\mathbb{R}^3}\rho\,dx=\int_{\mathbb{R}^3}|\varphi|^2\,dx.$$

质量守恒, 能量守恒.

5.3 ADI 算法流程

每个时间步迭代:

1. 给定 φ_0 和 $\hat{U}_{\text{dip}}$
2. $\varphi_0 \to \varphi_{\frac{1}{2}}$: 求解 $\mathbf{i}\dfrac{\partial\varphi}{\partial t} = -\varepsilon\Delta\varphi$
3. $\rho_{\frac{1}{2}} = |\varphi_{\frac{1}{2}}|^2 \to \hat{\rho}_{\frac{1}{2}}$: 点乘并进行傅里叶正变换
4. $\hat{\Phi}_{\frac{1}{2}} = \hat{\rho}_{\frac{1}{2}}\hat{U}_{\text{dip}} \to \Phi_{\frac{1}{2}}$: 点乘并进行傅里叶逆变换
5. $\varphi_{\frac{1}{2}} \to \varphi_{\frac{3}{2}}$: 求解 $\mathbf{i}\dfrac{\partial\varphi}{\partial t} = (V + \beta|\varphi|^2 + \lambda\Phi)\varphi - \omega L_z\varphi$
6. $\varphi_{\frac{3}{2}} \to \varphi_2$: 求解 $\mathbf{i}\dfrac{\partial\varphi}{\partial t} = -\varepsilon\Delta\varphi$

这里, $\hat{U}_{\text{dip}}$ 和 $\hat{\rho}$ 分别表示 U_{dip} 和 ρ 的傅里叶变换.

5.3.1 STEP 2

对于 $t \in [t_s, t_e]$,

$$\mathbf{i}\frac{\partial\varphi}{\partial t}(x,t) = -\varepsilon\Delta\varphi(x,t),$$

即,

$$\mathbf{i}\frac{\partial\psi}{\partial t}(k,t) = \varepsilon|k|^2\psi(k,t),$$

得到

$$\psi(k,t_e) = \psi(k,t_s)\exp\Big(-\mathbf{i}\varepsilon|k|^2(t_e - t_s)\Big),$$

其中, $k = (k_1, k_2, k_3)$,

$$\begin{aligned}
\varphi(x,t) &= \int_{\mathbb{R}^3}\psi(k,t)\cdot\exp(+\mathbf{i}k\cdot x)\,dk,\\
\frac{\partial\varphi}{\partial t}(x,t) &= \int_{\mathbb{R}^3}\frac{\partial\psi}{\partial t}(k,t)\cdot\exp(+\mathbf{i}k\cdot x)\,dk,\\
\frac{\partial\varphi}{\partial x_j}(x,t) &= \int_{\mathbb{R}^3}\mathbf{i}k_j\cdot\psi(k,t)\cdot\exp(+\mathbf{i}k\cdot x)\,dk,\ j = 1,2,3,\\
\Delta\varphi(x,t) &= \int_{\mathbb{R}^3}-|k|^2\psi(k,t)\cdot\exp(+\mathbf{i}k\cdot x)\,dk.
\end{aligned}$$

Remark 5.1.

$$\int_{\mathbb{R}^3} \varphi^*(+\mathbf{i})\frac{\partial \varphi}{\partial t}(x,t)\,dx = \int_{\mathbb{R}^3} -\varepsilon\varphi^*\Delta\varphi\,dx,$$
$$\int_{\mathbb{R}^3} \varphi(-\mathbf{i})\frac{\partial \varphi^*}{\partial t}(x,t)\,dx = \int_{\mathbb{R}^3} -\varepsilon\varphi\Delta\varphi^*\,dx,$$

那么,

$$\frac{d}{dt}\int_{\mathbb{R}^3} |\varphi(x,t)|^2\,dx = 0.$$

5.3.2 STEP 4

$$\Phi(x,t) = \int_{\mathbb{R}^3} U_{\text{dip}}(x-\tilde{x})\rho(\tilde{x},t)\,d\tilde{x}$$
$$\hat{\Phi}(k,t) = \hat{U}_{\text{dip}}(k)\hat{\rho}(k,t)$$

Remark 5.2. U_{dip} 和 ρ 都是实值函数, 则对于

$$\rho(x,t) = \int_{\mathbb{R}^3} \hat{\rho}(k,t)\cdot \exp(+\mathbf{i}k\cdot x)\,dk,$$
$$U_{dip}(x) = \int_{\mathbb{R}^3} \hat{U}_{dip}(k)\cdot \exp(+\mathbf{i}k\cdot x)\,dk,$$

两边取共轭

$$\rho(x,t) = \int_{\mathbb{R}^3} \hat{\rho}^*(k,t)\cdot \exp(-\mathbf{i}k\cdot x)\,dk,$$
$$U_{dip}(x) = \int_{\mathbb{R}^3} \hat{U}^*_{dip}(k)\cdot \exp(-\mathbf{i}k\cdot x)\,dk,$$

得到 $\hat{\rho}(k,t) = \hat{\rho}^*(-k,t)$, $\hat{U}_{dip}(k) = \hat{U}^*_{dip}(-k)$.

5.3.3 STEP 5

当 $\omega = 0$ 时, 求解 $\mathbf{i}\dfrac{\partial\varphi}{\partial t} = (V+\beta|\varphi|^2+\lambda\Phi)\varphi$ 等价于求解 $\mathbf{i}\dfrac{\partial\varphi}{\partial t} = (V+\beta|\varphi_{\frac{1}{2}}|^2+\lambda\Phi_{\frac{1}{2}})\varphi$. 这是因为

$$\varphi^*(+\mathbf{i})\frac{\partial\varphi}{\partial t} = (V+\beta|\varphi|^2+\lambda\Phi)\varphi^*\varphi,\ \varphi(-\mathbf{i})\frac{\partial\varphi^*}{\partial t} = (V+\beta|\varphi|^2+\lambda\Phi)\varphi\varphi^*,$$

即

$$\frac{d}{dt}|\varphi(x,t)|^2 = 0.$$

对于一般情况, 可以考虑使用预估校正方法求解.

对于 $t \in [t_s, t_e]$,

$$\mathbf{i}\frac{\partial \varphi}{\partial t}(x,t) = (V(x) + \beta|\varphi(x,t)|^2 + \lambda\Phi(x,t))\varphi - \omega L_z \varphi(x,t),$$

首先计算

$$\mathbf{i}\frac{\partial \varphi}{\partial t} = (V(x) + \beta|\varphi(x,t_s)|^2 + \lambda\Phi(x,t_s))\varphi - \omega L_z \varphi(x,t_s),$$

得到解析表达式

$$\tilde{\varphi}(x,t_e) = (\varphi(x,t_s) - \frac{q(x)}{p(x)}) \exp\Big(-\mathbf{i}(t_e - t_s)p(x)\Big) + \frac{q(x)}{p(x)},$$

其中,

$$\begin{aligned} p(x) &= V(x) + \beta|\varphi(x,t_s)|^2 + \lambda\Phi(x,t_s). \\ q(x) &= \omega L_z \varphi(x,t_s). \end{aligned}$$

然后计算

$$\mathbf{i}\frac{\partial \varphi}{\partial t} = (V(x) + \beta|\tilde{\varphi}(x,t_e)|^2 + \lambda\tilde{\Phi}(x,t_e))\varphi - \omega L_z \tilde{\varphi}(x,t_e),$$

得到解析表达式

$$\bar{\varphi}(x,t_e) = (\varphi(x,t_s) - \frac{\tilde{q}(x)}{\tilde{p}(x)}) \exp\Big(-\mathbf{i}(t_e - t_s)\tilde{p}(x)\Big) + \frac{\tilde{q}(x)}{\tilde{p}(x)},$$

其中, $\tilde{\Phi}(x,t_e) = \int_{\mathbb{R}^3} U_{\text{dip}}(x-\tilde{x})|\tilde{\varphi}(\tilde{x},t_e)|^2\, d\tilde{x}$,

$$\begin{aligned} \tilde{p}(x) &= V(x) + \beta|\tilde{\varphi}(x,t_e)|^2 + \lambda\tilde{\Phi}(x,t_e). \\ \tilde{q}(x) &= \omega L_z \tilde{\varphi}(x,t_e). \end{aligned}$$

最后,

$$\varphi(x,t_e) \approx \frac{1}{2}(\tilde{\varphi}(x,t_e) + \bar{\varphi}(x,t_e)).$$

Remark 5.3.

$$\frac{\partial \varphi}{\partial t} + (\mathbf{i}p)\varphi = (\mathbf{i}q)$$

得到

$$\varphi = Ce^{-\int \mathbf{i}p\, dt} + e^{-\int \mathbf{i}p\, dt} \int \mathbf{i}q e^{\int \mathbf{i}p\, dt}\, dt$$

5.4 数值结果

$$\varepsilon = 0.5,\ \gamma = 0.5,\ \beta = -2,\ \lambda = 2,\ \omega = 1,\ [-8,8]^3 \times [0,1]$$

$$\varphi(x,0) = \sqrt{\frac{1}{(\sqrt{2\pi})^3}\exp\big(-\frac{1}{2}(x_1^2 + x_2^2 + x_3^2)\big)}$$

密度函数的切片图5.1

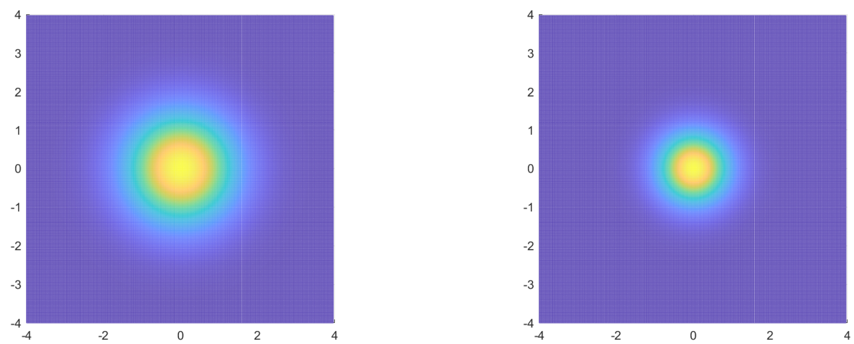

Figure 5.1: 左图: $t = 0$; 右图: $t = 1$

注意, 由于所取初值的特殊性, 得到 $L_z\varphi(x,t) = 0$. 一般地, 上述预估校正方法不保持数值解的质量守恒.

5.4.1 性能测试

由表5.1, 可知对于空间规模为 64^3 的算例, 每个时间步耗时约 0.00367 秒; 对于空间规模为 128^3 的算例, 每个时间步耗时约 0.027 秒; 对于空间规模为 256^3 的算例, 每个时间步耗时约 0.249 秒.

Table 5.1: 性能测试 with $[-8,8]^3 \times [0,1]$

空间规模	时间步长	时间 (sec)	显存 (Mb)	质量误差	能量误差
32^3	10^{-3}	0.90	237	5.5366×10^{-13}	1.654609×10^{-7}
64^3	10^{-3}	3.67	329	3.6837×10^{-13}	1.654654×10^{-7}
128^3	10^{-3}	27.77	1141	7.7182×10^{-13}	1.654661×10^{-7}
256^3	10^{-3}	249.18	7637	6.8826×10^{-12}	1.654668×10^{-7}

5.4.2 时间误差阶

由表5.2, 时间误差阶数为 2 阶.

Table 5.2: 时间误差阶 with $[-8,8]^3 \times [0,1]$

空间规模	时间步长	时间 (sec)	显存 (Mb)	质量误差	能量误差
128^3	1.00×10^{-3}	27.69	1141	7.7182×10^{-13}	1.654661×10^{-7}
128^3	5.00×10^{-4}	55.75	1141	9.6289×10^{-13}	4.136790×10^{-8}
128^3	2.50×10^{-4}	112.81	1141	1.3539×10^{-12}	1.034407×10^{-8}
128^3	1.25×10^{-4}	228.50	1141	2.2607×10^{-12}	2.589687×10^{-9}

5.4.3 长时间稳定性

$[-8,8]^3 \times [0,20]$ with 空间规模 128^3 and 时间步长 1.00×10^{-3}. 总时间 573.18 秒, 质量误差 5.0438×10^{-12}, 能量误差 1.275846×10^{-7}.

6 程序实现

6.1 基本操作

6.1.1 全局编号

$$
\begin{aligned}
\varphi(x) &= \sum_{k=0}^{n/2-1} \psi[k] \exp\Big(+2\pi\mathbf{i}\cdot(k\frac{x-a}{l})\Big) + \sum_{k=-n/2}^{-1} \psi[n+k] \exp\Big(+2\pi\mathbf{i}\cdot(k\frac{x-a}{l})\Big) \\
&= \sum_{k=0}^{n/2-1} \psi[k] \exp\Big(+2\pi\mathbf{i}\cdot(k\frac{x-a}{l})\Big) + \sum_{k=n/2}^{n-1} \psi[k] \exp\Big(+2\pi\mathbf{i}\cdot((k-n)\frac{x-a}{l})\Big) \\
\varphi[j] &= \sum_{k=0}^{n/2-1} \psi[k] \exp\Big(+\mathbf{i}\cdot(kj)\cdot(2\pi\Delta x/l)\Big) + \sum_{k=-n/2}^{-1} \psi[n+k] \exp\Big(+\mathbf{i}\cdot(kj)\cdot(2\pi\Delta x/l)\Big) \\
&= \sum_{k=0}^{n-1} \psi[k] \exp\Big(+\mathbf{i}\cdot(kj)\cdot(2\pi/n)\Big) \\
n\psi[k] &= \sum_{j=0}^{n-1} \varphi[j] \exp\Big(-\mathbf{i}\cdot(kj)\cdot(2\pi/n)\Big)
\end{aligned}
$$

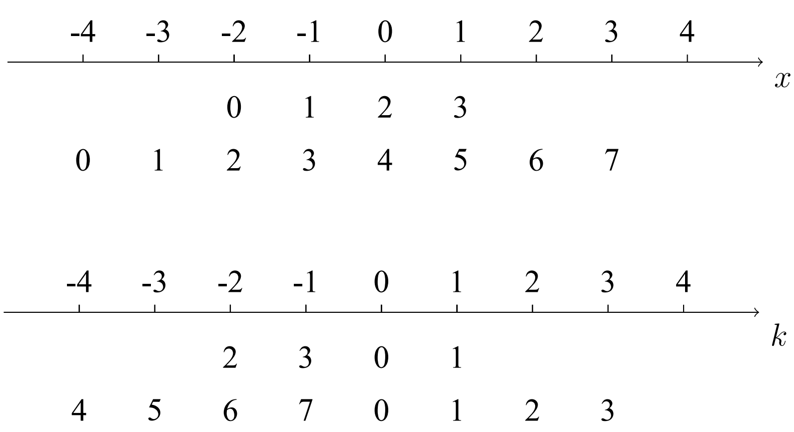

Figure 6.1: 一维时域对应 (上图), 一维频域对应 (下图)

```
int N[3] = {8,16,32}; /* dimensions of (x,y,z) */
dim3 dimGrid(N[0], N[1]), dimBlock(N[2]);
dim3 dimGridBig(2*N[0], 2*N[1]), dimBlockBig(2*N[2]);

__device__
```

3	7	11	15
2	6	10	14
1	5	9	13
0	4	8	12

(0,-1)	(1,-1)	(-2,-1)	(-1,-1)
(0,-2)	(1,-2)	(-2,-2)	(-1,-2)
(0,1)	(1,1)	(-2,1)	(-1,1)
(0,0)	(1,0)	(-2,0)	(-1,0)

Figure 6.2: 二维时域对应 (左图), 二维频域对应 (右图)

```
int GetGlobalIdx3D()
{
   return (blockIdx.x*gridDim.y+blockIdx.y)*blockDim.x+threadIdx.x;
}
__device__
void GetIdxFreq3D(int k[3])
{
    int N[3] = {gridDim.x, gridDim.y, blockDim.x};
    int j[3] = {blockIdx.x, blockIdx.y, threadIdx.x};
    for (int i = 0; i < 3; ++i) {
        k[i] = (j[i]>=N[i]/2)?(j[i]-N[i]):j[i];
    }
    return;
}
__device__
void GetCoordi3D(double x[3])
{
    int N[3] = {gridDim.x, gridDim.y, blockDim.x};
    int j[3] = {blockIdx.x, blockIdx.y, threadIdx.x};
    for (int i = 0; i < 3; ++i) {
        x[i] = LOWER[i] + j[i]*(UPPER[i]-LOWER[i])/N[i];
    }
    return;
}
```

6.1.2 代数运算

```
__device__
cufftDoubleReal _norm2(cufftDoubleComplex *X)
{
    return (X->x)*(X->x)+(X->y)*(X->y);
}
__global__
void ComputeDot(cufftDoubleComplex *Z, cufftDoubleReal alpha,
    cufftDoubleComplex *X, cufftDoubleComplex *Y)
{
    int globalIndex = GetGlobalIdx3D();
    Z[globalIndex] = cuCmul(X[globalIndex], Y[globalIndex]);
    Z[globalIndex].x *= alpha;
    Z[globalIndex].y *= alpha;
    return;
}
```

6.1.3 限制与延拓

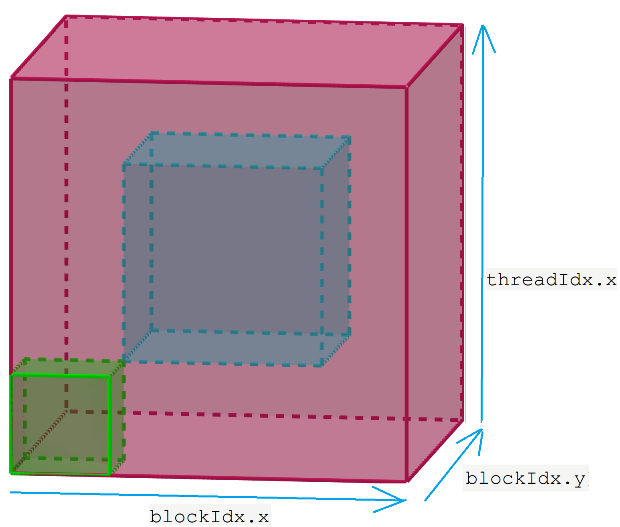

Figure 6.3: 在角 (绿色), 在核 (蓝色)

```
__device__
bool IsInCore()
```

```
{
    if (  ( blockIdx.x< gridDim.x/4|| blockIdx.x>=3* gridDim.x/4)
        ||( blockIdx.y< gridDim.y/4|| blockIdx.y>=3* gridDim.y/4)
        ||(threadIdx.x<blockDim.x/4||threadIdx.x>=3*blockDim.x/4)) {
        return false;
    }
    else {
        return true;
    }
}
__device__
bool IsAtCorner()
{
    if (  ( blockIdx.x< gridDim.x/4|| blockIdx.x>=3* gridDim.x/4)
        &&( blockIdx.y< gridDim.y/4|| blockIdx.y>=3* gridDim.y/4)
        &&(threadIdx.x<blockDim.x/4||threadIdx.x>=3*blockDim.x/4)) {
        return true;
    }
    else {
        return false;
    }
}
```

```
PaddingFullTime<<<dimGridBig, dimBlockBig>>>(full, core, flag);
PaddingCoreTime<<<dimGrid, dimBlock>>>(core, full, flag);
PaddingFullFreq<<<dimGridBig, dimBlockBig>>>(full, core);
PaddingCoreFreq<<<dimGrid, dimBlock>>>(core, full);

__global__
void PaddingFullTime(cufftDoubleComplex *full,
    cufftDoubleComplex *core, int flag)
{
    int globalIndex = GetGlobalIdx3D();
    if (!IsInCore()) {
        full[globalIndex].x = 0.0;
        full[globalIndex].y = 0.0;
```

```
    }
    else {
        int N[3] = {gridDim.x, gridDim.y, blockDim.x};
        int j[3] = {blockIdx.x, blockIdx.y, threadIdx.x};
        for (int i = 0; i < 3; ++i) {
            j[i] -= N[i]/4;
        }
        int idx = (j[0]*N[1]/2+j[1])*N[2]/2+j[2];
        if (flag == 2) {
            full[globalIndex].x = _norm2(core+idx);
            full[globalIndex].y = 0.0;
        } else if (flag == 1) {
            full[globalIndex].x = core[idx].x;
            full[globalIndex].y = 0.0;
        } else if (flag == -1) {
            full[globalIndex].x = 0.0;
            full[globalIndex].y = core[idx].y;
        }
        else {
            full[globalIndex].x = core[idx].x;
            full[globalIndex].y = core[idx].y;
        }
    }
    return;
}
__global__
void PaddingCoreTime(cufftDoubleComplex *core,
    cufftDoubleComplex *full, int flag)
{
    int globalIndex = GetGlobalIdx3D();
    int N[3] = {gridDim.x, gridDim.y, blockDim.x};
    int j[3] = {blockIdx.x, blockIdx.y, threadIdx.x};
    for (int i = 0; i < 3; ++i) {
        j[i] += N[i]/2;
    }
    int idx = (j[0]*2*N[1]+j[1])*2*N[2]+j[2];
```

```
    if (flag == 2) {
        core[globalIndex].x = _norm2(full+idx);
        core[globalIndex].y = 0.0;
    } else if (flag == 1) {
        core[globalIndex].x = full[idx].x;
        core[globalIndex].y = 0.0;
    } else if (flag == -1) {
        core[globalIndex].x = 0.0;
        core[globalIndex].y = full[idx].y;
    }
    else {
        core[globalIndex].x = full[idx].x;
        core[globalIndex].y = full[idx].y;
    }
    return;
}
__global__
void PaddingFullFreq(cufftDoubleComplex *full,
    cufftDoubleComplex *core)
{
    int globalIndex = GetGlobalIdx3D();
    if (IsAtCorner()) {
        int k[3], N[3] = {gridDim.x/2, gridDim.y/2, blockDim.x/2};
        k[0] = ( blockIdx.x>=N[0])?( blockIdx.x-N[0]): blockIdx.x;
        k[1] = ( blockIdx.y>=N[1])?( blockIdx.y-N[1]): blockIdx.y;
        k[2] = (threadIdx.x>=N[2])?(threadIdx.x-N[2]):threadIdx.x;
        int idx = (k[0]*N[1]+k[1])*N[2]+k[2];
        full[globalIndex].x = core[idx].x;
        full[globalIndex].y = core[idx].y;
    }
    else {
        full[globalIndex].x = 0.0;
        full[globalIndex].y = 0.0;
    }
    return;
}
```

```
__global__
void PaddingCoreFreq(cufftDoubleComplex *core,
    cufftDoubleComplex *full)
{
    int globalIndex = GetGlobalIdx3D();
    int k[3], N[3] = {gridDim.x, gridDim.y, blockDim.x};
    k[0] = ( blockIdx.x>=N[0]/2)?( blockIdx.x+N[0]): blockIdx.x;
    k[1] = ( blockIdx.y>=N[1]/2)?( blockIdx.y+N[1]): blockIdx.y;
    k[2] = (threadIdx.x>=N[2]/2)?(threadIdx.x+N[2]):threadIdx.x;
    int idx = (k[0]*2*N[1]+k[1])*2*N[2]+k[2];
    core[globalIndex].x = full[idx].x;
    core[globalIndex].y = full[idx].y;
    return;
}
```

6.2 cuFFT 创建与销毁

```
cufftHandle planZ2Z, planBigD2Z, planBigZ2D;
void *fft_workspace = NULL;
size_t ws_size = sizeof(cufftDoubleComplex)*4*N[0]*N[1]*(N[2]+1);
cudaMalloc(&fft_workspace, ws_size);
cufftCreate(&planZ2Z);
cufftCreate(&planBigD2Z);
cufftCreate(&planBigZ2D);
cufftSetAutoAllocation(planZ2Z   , 0);
cufftSetAutoAllocation(planBigD2Z, 0);
cufftSetAutoAllocation(planBigZ2D, 0);
size_t workSize = 0;
cufftMakePlan3d(planZ2Z   ,   N[0],   N[1],   N[2],
    CUFFT_Z2Z, &workSize);
cufftMakePlan3d(planBigD2Z, 2*N[0], 2*N[1], 2*N[2],
    CUFFT_D2Z, &workSize);
cufftMakePlan3d(planBigZ2D, 2*N[0], 2*N[1], 2*N[2],
    CUFFT_Z2D, &workSize);
cufftSetWorkArea(planZ2Z   , fft_workspace);
```

```
cufftSetWorkArea(planBigD2Z, fft_workspace);
cufftSetWorkArea(planBigZ2D, fft_workspace);
```

```
cufftDestroy(planZ2Z);
cufftDestroy(planBigD2Z);
cufftDestroy(planBigZ2D);
cudaFree(fft_workspace);
```

6.3 微分算子

```
/* data_out -> D phi, data_in -> phi */
__host__
void DifferentialOperator(cufftDoubleComplex *data_out,
    cufftDoubleComplex *data_in, int diff_order[3],
    cufftHandle plan, int N[3])
{
    int total_size = N[0]*N[1]*N[2];
    int dox = diff_order[0], doy = diff_order[1], doz = diff_order[2];
    dim3 dimGrid(N[0],N[1]), dimBlock(N[2]);
    cufftExecZ2Z(plan, data_in, data_out, CUFFT_FORWARD);
    cudaDeviceSynchronize();
    _DifferentialOperator<<<dimGrid, dimBlock>>>(data_out,
        data_out, dox,doy,doz, 1./total_size);
    cudaDeviceSynchronize();
    cufftExecZ2Z(plan, data_out, data_out, CUFFT_INVERSE);
    cudaDeviceSynchronize();
    return;
}
__global__
void _DifferentialOperator(cufftDoubleComplex *data_out,
    cufftDoubleComplex *data_in, int dox, int doy, int doz,
    cufftDoubleReal alpha)
{
    int globalIndex = GetGlobalIdx3D();
    int k[3], diff_order[3] = {dox, doy, doz};
```

```
    double length;
    GetIdxFreq3D(k);
    cufftDoubleComplex v[3];
    for (int i = 0; i < 3; ++i) {
        length = UPPER[i]-LOWER[i];
        v[i] = make_cuDoubleComplex(0, 2*M_PI*k[i]/length);
    }
    data_out[globalIndex].x = data_in[globalIndex].x;
    data_out[globalIndex].y = data_in[globalIndex].y;
    for (int i = 0; i < 3; ++i) {
        for (int d = 0; d < diff_order[i]; ++d) {
            data_out[globalIndex] = cuCmul(data_out[globalIndex], v[i]);
        }
    }
    data_out[globalIndex].x *= alpha;
    data_out[globalIndex].y *= alpha;
    return;
}
```

6.3.1 Laplace 项

```
/* data_out -> -Delta phi, data_in -> phi */
__host__
void LaplaceOperator(cufftDoubleComplex *data_out,
    cufftDoubleComplex *data_in, cufftHandle plan, int N[3])
{
    int total_size = N[0]*N[1]*N[2];
    dim3 dimGrid(N[0], N[1]), dimBlock(N[2]);
    cufftExecZ2Z(plan, data_in, data_out, CUFFT_FORWARD);
    cudaDeviceSynchronize();
    _LaplaceOperator<<<dimGrid, dimBlock>>>(data_out,
        data_out, 1./total_size);
    cudaDeviceSynchronize();
    cufftExecZ2Z(plan, data_out, data_out, CUFFT_INVERSE);
    cudaDeviceSynchronize();
```

```
}
__global__
void _LaplaceOperator(cufftDoubleComplex *data_out,
    cufftDoubleComplex *data_in, cufftDoubleReal alpha)
{
    int globalIndex = GetGlobalIdx3D();
    int k[3];
    GetIdxFreq3D(k);
    double length;
    double v = 0;
    for (int i = 0;  i < 3; ++i) {
        length = UPPER[i]-LOWER[i];
        v += (k[i]/length)*(k[i]/length);
    }
    v *= -4*M_PI*M_PI;
    data_out[globalIndex].x = data_in[globalIndex].x * v;
    data_out[globalIndex].y = data_in[globalIndex].y * v;
    data_out[globalIndex].x *= alpha;
    data_out[globalIndex].y *= alpha;
    return;
}
```

6.3.2 Rotation 项

```
/* data_out -> Lz phi, data_in -> phi
 * size of workspace = N[0]*N[1]*N[2] */
__host__
void RotationOperator(cufftDoubleComplex *data_out,
    cufftDoubleComplex *data_in, cufftHandle plan, int N[3],
    cufftDoubleComplex *workspace)
{
    int total_size = N[0]*N[1]*N[2];
    dim3 dimGrid(N[0],N[1]), dimBlock(N[2]);
    cufftExecZ2Z(plan, data_in, data_out, CUFFT_FORWARD);
    cudaDeviceSynchronize();
```

```
    _DifferentialOperator<<<dimGrid, dimBlock>>>(workspace,
        data_out, 1,0,0, 1./total_size);
    cudaDeviceSynchronize();
    cufftExecZ2Z(plan, workspace, workspace, CUFFT_INVERSE);
    _DifferentialOperator<<<dimGrid, dimBlock>>>(data_out,
        data_out, 0,1,0, 1./total_size);
    cudaDeviceSynchronize();
    cufftExecZ2Z(plan, data_out, data_out, CUFFT_INVERSE);
    cudaDeviceSynchronize();
    _RotationOperator<<<dimGrid, dimBlock>>>(data_out,
        workspace, data_out);
    cudaDeviceSynchronize();
    return;
}
/* Lz phi = -i ( x Py - y Px ) phi
 * Py may be have same address as Lz */
__global__
void _RotationOperator(cufftDoubleComplex *Lz,
    cufftDoubleComplex *Px, cufftDoubleComplex *Py)
{
    int globalIndex = GetGlobalIdx3D();
    double coordi[3];
    GetCoordi3D(coordi);
    Lz[globalIndex].x  = Py[globalIndex].x*coordi[0];
    Lz[globalIndex].y  = Py[globalIndex].y*coordi[0];
    Lz[globalIndex].x -= Px[globalIndex].x*coordi[1];
    Lz[globalIndex].y -= Px[globalIndex].y*coordi[1];
    Lz[globalIndex] = cuCmul(Lz[globalIndex],
        make_cuDoubleComplex(0, -1));
    return;
}
```

6.4 Dipolar 项

```
/* data_out -> Phi, data_in -> rho = |phi|^2
```

```
 * size of ws_cplx = 4*N[0]*N[1]*(N[2]+1)
 * size of ws_real = 8*N[0]*N[1]*N[2] */
__host__
void DipolarOperator(cufftDoubleReal *data_out,
    cufftDoubleReal *data_in, cufftDoubleComplex *hatU,
    cufftHandle planBigD2Z, cufftHandle planBigZ2D,
    int N[3], cufftDoubleReal dx[3],
    cufftDoubleComplex *ws_cplx, cufftDoubleReal *ws_real)
{
    int total_size = N[0]*N[1]*N[2];
    cufftDoubleReal h3 = dx[0]*dx[1]*dx[2];
    dim3 dimGrid(N[0], N[1]), dimBlock(N[2]);
    dim3 dimGridBig(2*N[0], 2*N[1]), dimBlockBig(2*N[2]);
    dim3 dimBlockBigHalf(N[2]+1);
    PaddingFullTime<<<dimGridBig, dimBlockBig>>>(ws_real, data_in, 1);
    cudaDeviceSynchronize();
    cufftExecD2Z(planBigD2Z, ws_real, ws_cplx);
    cudaDeviceSynchronize();
    ComputeDot<<<dimGridBig, dimBlockBigHalf>>>(ws_cplx,
        h3/(8.*total_size), ws_cplx, hatU);
    cudaDeviceSynchronize();
    cufftExecZ2D(planBigZ2D, ws_cplx, ws_real);
    cudaDeviceSynchronize();
    PaddingCoreTime<<<dimGrid, dimBlock>>>(data_out, ws_real, 1);
    cudaDeviceSynchronize();
    return;
}
```

A 三维离散傅里叶变换的具体计算形式

标准的离散傅里叶变换是对定义在 $[0,2\pi]^3$ 上的函数进行的. 所以, 首先进行坐标变换

$$(x-a_1)\frac{2\pi}{l_1},\ (y-a_2)\frac{2\pi}{l_2},\ (z-a_3)\frac{2\pi}{l_3},$$

使得 $(x,y,z)\in[a_1,b_1]\times[a_2,b_2]\times[a_3,b_3]$, 其中, $l_i=b_i-a_i$, $i=1,2,3$. 那么

$$\begin{aligned}\varphi(x,y,z,t)=&\sum_{k_1=0}^{n_1/2-1}\sum_{k_2=0}^{n_2/2-1}\sum_{k_3=0}^{n_3/2-1}\psi[k_1][k_2][k_3](t)\exp\Big(+2\pi\mathbf{i}\cdot(k_1\frac{x-a_1}{l_1}+k_2\frac{y-a_2}{l_2}+k_3\frac{z-a_3}{l_3})\Big)\\&+\cdots\\&+\sum_{k_1=-n_1/2}^{-1}\sum_{k_2=-n_2/2}^{-1}\sum_{k_3=-n_3/2}^{-1}\psi[n_1+k_1][n_2+k_2][n_3+k_3](t)\cdot\\&\exp\Big(+2\pi\mathbf{i}\cdot(k_1\frac{x-a_1}{l_1}+k_2\frac{y-a_2}{l_2}+k_3\frac{z-a_3}{l_3})\Big)\end{aligned}$$

等号右端共八项, 其中等式右端第八个求和式为

$$\begin{aligned}&\sum_{k_1=n_1/2}^{n_1-1}\sum_{k_2=n_2/2}^{n_2-1}\sum_{k_3=n_3/2}^{n_3-1}\psi[k_1][k_2][k_3](t)\cdot\\&\exp\Big(+2\pi\mathbf{i}\cdot((k_1-n_1)\frac{x-a_1}{l_1}+(k_2-n_2)\frac{y-a_2}{l_2}+(k_3-n_3)\frac{z-a_3}{l_3})\Big)\\=&\sum_{k_1=n_1/2}^{n_1-1}\sum_{k_2=n_2/2}^{n_2-1}\sum_{k_3=n_3/2}^{n_3-1}\psi[k_1][k_2][k_3](t)\exp\Big(+2\pi\mathbf{i}\cdot(k_1\frac{x-a_1}{l_1}+k_2\frac{y-a_2}{l_2}+k_3\frac{z-a_3}{l_3})\Big)\cdot\\&\exp\Big(+2\pi\mathbf{i}\cdot(-n_1\frac{x-a_1}{l_1}-n_2\frac{y-a_2}{l_2}-n_3\frac{z-a_3}{l_3})\Big)\end{aligned}$$

取

$$\begin{aligned}x_{j_1}&=a_1+j_1\Delta x,\ \frac{x_{j_1}-a_1}{l_1}=\frac{j_1}{n_1}\\y_{j_2}&=a_2+j_2\Delta y,\ \frac{y_{j_2}-a_2}{l_2}=\frac{j_2}{n_2}\\z_{j_3}&=a_3+j_3\Delta z,\ \frac{z_{j_3}-a_3}{l_3}=\frac{j_3}{n_3}\end{aligned}$$

其中, $j_i=0,1,\ldots,n_i-1$, $\Delta x=l_1/n_1$, $\Delta y=l_2/n_2$, $\Delta z=l_3/n_3$, $i=1,2,3$.

$$\varphi(x_{j_1},y_{j_2},z_{j_3},t)=\sum_{k_1=0}^{n_1-1}\sum_{k_2=0}^{n_2-1}\sum_{k_3=0}^{n_3-1}\psi[k_1][k_2][k_3](t)\exp\Big(+2\pi\mathbf{i}\cdot(\frac{j_1k_1}{n_1}+\frac{j_2k_2}{n_2}+\frac{j_3k_3}{n_3})\Big)$$

$$\frac{\partial \varphi}{\partial t}(x_{j_1}, y_{j_2}, z_{j_3}, t) = \sum_{k_1=0}^{n_1-1} \sum_{k_2=0}^{n_2-1} \sum_{k_3=0}^{n_3-1} \frac{\partial \psi[k_1][k_2][k_3]}{\partial t}(t) \exp\Big(+ 2\pi \mathbf{i} \cdot (\frac{j_1 k_1}{n_1} + \frac{j_2 k_2}{n_2} + \frac{j_3 k_3}{n_3})\Big)$$

$$\Delta \varphi(x_{j_1}, y_{j_2}, z_{j_3}, t) = \sum_{k_1=0}^{n_1/2-1} \sum_{k_2=0}^{n_2/2-1} \sum_{k_3=0}^{n_3/2-1} -4\pi^2 \Big((\frac{k_1}{l_1})^2 + (\frac{k_2}{l_2})^2 + (\frac{k_3}{l_3})^2 \Big) \cdot$$

$$\psi[k_1][k_2][k_3](t) \exp\Big(+ 2\pi \mathbf{i} \cdot (\frac{j_1 k_1}{n_1} + \frac{j_2 k_2}{n_2} + \frac{j_3 k_3}{n_3})\Big)$$

$$+ \cdots$$

$$+ \sum_{k_1=n_1/2}^{n_1-1} \sum_{k_2=n_2/2}^{n_2-1} \sum_{k_3=n_3/2}^{n_3-1} -4\pi^2 \Big((\frac{k_1 - n_1}{l_1})^2 + (\frac{k_2 - n_2}{l_2})^2 + (\frac{k_3 - n_3}{l_3})^2 \Big) \cdot$$

$$\psi[k_1][k_2][k_3](t) \exp\Big(+ 2\pi \mathbf{i} \cdot (\frac{j_1 k_1}{n_1} + \frac{j_2 k_2}{n_2} + \frac{j_3 k_3}{n_3})\Big)$$

$$\partial_x \varphi(x_{j_1}, y_{j_2}, z_{j_3}, t) = \sum_{k_1=0}^{n_1/2-1} \sum_{k_2=0}^{n_2/2-1} \sum_{k_3=0}^{n_3/2-1} +2\pi \mathbf{i} \Big(\frac{k_1}{l_1} \Big) \cdot$$

$$\psi[k_1][k_2][k_3](t) \exp\Big(+ 2\pi \mathbf{i} \cdot (\frac{j_1 k_1}{n_1} + \frac{j_2 k_2}{n_2} + \frac{j_3 k_3}{n_3})\Big)$$

$$+ \cdots$$

$$+ \sum_{k_1=n_1/2}^{n_1-1} \sum_{k_2=n_2/2}^{n_2-1} \sum_{k_3=n_3/2}^{n_3-1} +2\pi \mathbf{i} \Big(\frac{k_1 - n_1}{l_1} \Big) \cdot$$

$$\psi[k_1][k_2][k_3](t) \exp\Big(+ 2\pi \mathbf{i} \cdot (\frac{j_1 k_1}{n_1} + \frac{j_2 k_2}{n_2} + \frac{j_3 k_3}{n_3})\Big)$$

$$\partial_y \varphi(x_{j_1}, y_{j_2}, z_{j_3}, t) = \sum_{k_1=0}^{n_1/2-1} \sum_{k_2=0}^{n_2/2-1} \sum_{k_3=0}^{n_3/2-1} +2\pi \mathbf{i} \Big(\frac{k_2}{l_2} \Big) \cdot$$

$$\psi[k_1][k_2][k_3](t) \exp\Big(+ 2\pi \mathbf{i} \cdot (\frac{j_1 k_1}{n_1} + \frac{j_2 k_2}{n_2} + \frac{j_3 k_3}{n_3})\Big)$$

$$+ \cdots$$

$$+ \sum_{k_1=n_1/2}^{n_1-1} \sum_{k_2=n_2/2}^{n_2-1} \sum_{k_3=n_3/2}^{n_3-1} +2\pi \mathbf{i} \Big(\frac{k_2 - n_2}{l_2} \Big) \cdot$$

$$\psi[k_1][k_2][k_3](t) \exp\Big(+ 2\pi \mathbf{i} \cdot (\frac{j_1 k_1}{n_1} + \frac{j_2 k_2}{n_2} + \frac{j_3 k_3}{n_3})\Big)$$

$$\partial_z \varphi(x_{j_1}, y_{j_2}, z_{j_3}, t) = \sum_{k_1=0}^{n_1/2-1} \sum_{k_2=0}^{n_2/2-1} \sum_{k_3=0}^{n_3/2-1} +2\pi \mathbf{i} \Big(\frac{k_3}{l_3} \Big) \cdot$$

$$\psi[k_1][k_2][k_3](t) \exp\Big(+ 2\pi \mathbf{i} \cdot (\frac{j_1 k_1}{n_1} + \frac{j_2 k_2}{n_2} + \frac{j_3 k_3}{n_3})\Big)$$

$$
\begin{aligned}
&+\cdots \\
&+\sum_{k_1=n_1/2}^{n_1-1}\sum_{k_2=n_2/2}^{n_2-1}\sum_{k_3=n_3/2}^{n_3-1}+2\pi\mathbf{i}\Big(\frac{k_3-n_3}{l_3}\Big)\cdot \\
&\psi[k_1][k_2][k_3](t)\exp\Big(+2\pi\mathbf{i}\cdot(\frac{j_1k_1}{n_1}+\frac{j_2k_2}{n_2}+\frac{j_3k_3}{n_3})\Big)
\end{aligned}
$$

那么, $k_i = 0, 1, \ldots, n_i/2-1$, $i = 1, 2, 3$,

$$
\mathbf{i}\frac{\partial\psi[k_1][k_2][k_3]}{\partial t}(t) = 4\pi^2\varepsilon\Big((\frac{k_1}{l_1})^2+(\frac{k_2}{l_2})^2+(\frac{k_3}{l_3})^2\Big)\psi[k_1][k_2][k_3](t)
$$

且 $k_i = n_i/2, \ldots, n_i-1$, $i = 1, 2, 3$,

$$
\mathbf{i}\frac{\partial\psi[k_1][k_2][k_3]}{\partial t}(t) = 4\pi^2\varepsilon\Big((\frac{k_1-n_1}{l_1})^2+(\frac{k_2-n_2}{l_2})^2+(\frac{k_3-n_3}{l_3})^2\Big)\psi[k_1][k_2][k_3](t)
$$

Remark A.1. 卷积的计算

$$
\hat{\Phi}(k,t) = \hat{U}_{dip}(k)\cdot\hat{\rho}(k,t)\cdot\frac{l_1l_2l_3}{n_1n_2n_3}.
$$

Remark A.2. 若 $\varphi(x,y,z,t)$ 是实值函数, 则利用两边取共轭

$$\begin{aligned}
\varphi(x,y,z,t) = &\sum_{k_1=0}^{n_1/2-1}\sum_{k_2=0}^{n_2/2-1}\sum_{k_3=0}^{n_3/2-1}\psi^*[k_1][k_2][k_3](t)\\
&\exp\Big(+2\pi\mathbf{i}\cdot((-k_1)\frac{x-a_1}{l_1}+(-k_2)\frac{y-a_2}{l_2}+(-k_3)\frac{z-a_3}{l_3})\Big)\\
&+\cdots\\
&+\sum_{k_1=-n_1/2}^{-1}\sum_{k_2=-n_2/2}^{-1}\sum_{k_3=-n_3/2}^{-1}\psi^*[n_1+k_1][n_2+k_2][n_3+k_3](t)\cdot\\
&\exp\Big(+2\pi\mathbf{i}\cdot((-k_1)\frac{x-a_1}{l_1}+(-k_2)\frac{y-a_2}{l_2}+(-k_3)\frac{z-a_3}{l_3})\Big)\\
=&\sum_{k_1=-n_1/2+1}^{0}\sum_{k_2=-n_2/2+1}^{0}\sum_{k_3=-n_3/2+1}^{0}\psi^*[-k_1][-k_2][-k_3](t)\\
&\exp\Big(+2\pi\mathbf{i}\cdot(k_1\frac{x-a_1}{l_1}+k_2\frac{y-a_2}{l_2}+k_3\frac{z-a_3}{l_3})\Big)\\
&+\cdots\\
&+\sum_{k_1=1}^{n_1/2}\sum_{k_2=1}^{n_2/2}\sum_{k_3=1}^{n_3/2}\psi^*[n_1-k_1][n_2-k_2][n_3-k_3](t)\cdot\\
&\exp\Big(+2\pi\mathbf{i}\cdot(k_1\frac{x-a_1}{l_1}+k_2\frac{y-a_2}{l_2}+k_3\frac{z-a_3}{l_3})\Big)\\
=&\sum_{k_1=n_1/2+1}^{n_1}\sum_{k_2=n_2/2+1}^{n_2}\sum_{k_3=n_3/2+1}^{n_3}\psi^*[n_1-k_1][n_2-k_2][n_3-k_3](t)\\
&\exp\Big(+2\pi\mathbf{i}\cdot((k_1-n_1)\frac{x-a_1}{l_1}+(k_2-n_2)\frac{y-a_2}{l_2}+(k_3-n_3)\frac{z-a_3}{l_3})\Big)\\
&+\cdots\\
&+\sum_{k_1=1}^{n_1/2}\sum_{k_2=1}^{n_2/2}\sum_{k_3=1}^{n_3/2}\psi^*[n_1-k_1][n_2-k_2][n_3-k_3](t)\cdot\\
&\exp\Big(+2\pi\mathbf{i}\cdot(k_1\frac{x-a_1}{l_1}+k_2\frac{y-a_2}{l_2}+k_3\frac{z-a_3}{l_3})\Big)
\end{aligned}$$

得到

$$\psi[k_1][k_2][k_3](t)=\psi^*[n_1-k_1][n_2-k_2][n_3-k_3](t)$$

for $k_i=1,\ldots,n_i-1$, $i=1,2,3$.