实验报告

实验名称（用 **GPU** 加速 **FFT** 程序）

班级 智能 1602 学号 201608010605 姓名 金可欣

实验目标

用 GPU 加速 FFT 程序运行，测量加速前后的运行时间，确定加速比。

实验要求

- 采用 CUDA 或 OpenCL（视具体 GPU 而定）编写程序
- 根据自己的机器配置选择合适的输入数据大小 n
- 对测量结果进行分析，确定使用 GPU 加速 FFT 程序得到的加速比
- 回答思考题，答案加入到实验报告叙述中合适位置

思考题

1. 分析 GPU 加速 FFT 程序可能获得的加速比

2. 实际加速比相对于理想加速比差多少？原因是什么？

实验内容

FFT 算法代码

//fft.h

```
#ifndef __FFT_H__
#define __FFT_H__

typedef struct complex //复数类型
{
    float real;       //实部
    float imag;       //虚部
}complex;

#define PI 3.14159265358979323846264338327950288419 71


//////////////////////////////////////////
void conjugate_complex(int n, complex in[], complex out[]);
void c_plus(complex a, complex b, complex *c);//复数加
void c_mul(complex a, complex b, complex *c);//复数乘
void c_sub(complex a, complex b, complex *c);    //复数减法
void c_div(complex a, complex b, complex *c);    //复数除法
void fft(int N, complex f[]);//傅立叶变换 输出也存在数组f中
void ifft(int N, complex f[]); // 傅里叶逆变换
void c_abs(complex f[], float out[], float n);//复数数组取模
//////////////////////////////////////////
#endif#pragma once
```

//cuda

```
#include <iostream>
#include <time.h>
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include "../include/cufft.h"
#include "fft.h"
// 12   13   14     15    16    17
//4096 9192 18384 36748 73496  146992

#define NX 4096 // 有效数据个数
#define N  5335// 补0之后的数据长度
#define MAX 1<<12
#define BATCH 1
```

```
#define BLOCK_SIZE 1024
using std::cout;
using std::endl;

complex m[MAX], l[MAX];

/**
* 功能：判断两个 cufftComplex 数组的是否相等
* 输入：idataA 输入数组A的头指针
* 输入：idataB 输出数组B的头指针
* 输入：size 数组的元素个数
* 返回：true | false
*/
bool IsEqual(cufftComplex *idataA, cufftComplex *idataB, const long int size)
{
    for (int i = 0; i < size; i++)
    {
        if (abs(idataA[i].x - idataB[i].x) > 0.000001 || abs(idataA[i].y -
idataB[i].y) > 0.000001)
            return false;
    }

    return true;
}



/**
* 功能：实现 cufftComplex 数组的尺度缩放，也就是乘以一个数
* 输入：idata 输入数组的头指针
* 输出：odata 输出数组的头指针
* 输入：size 数组的元素个数
* 输入：scale 缩放尺度
*/
static __global__ void cufftComplexScale(cufftComplex *idata, cufftComplex *odata,
const long int size, float scale)
{
    const int threadID = blockIdx.x * blockDim.x + threadIdx.x;

    if (threadID < size)
    {
        odata[threadID].x = idata[threadID].x * scale;
        odata[threadID].y = idata[threadID].y * scale;
    }
```

```
}


void conjugate_complex(int n, complex in[], complex out[])
{
    int i = 0;
    for (i = 0; i < n; i++)
    {
        out[i].imag = -in[i].imag;
        out[i].real = in[i].real;
    }
}

void c_abs(complex f[], float out[], int n)
{
    int i = 0;
    float t;
    for (i = 0; i < n; i++)
    {
        t = f[i].real * f[i].real + f[i].imag * f[i].imag;
        out[i] = sqrt(t);
    }
}


void c_plus(complex a, complex b, complex *c)
{
    c->real = a.real + b.real;
    c->imag = a.imag + b.imag;
}

void c_sub(complex a, complex b, complex *c)
{
    c->real = a.real - b.real;
    c->imag = a.imag - b.imag;
}

void c_mul(complex a, complex b, complex *c)
{
    c->real = a.real * b.real - a.imag * b.imag;
    c->imag = a.real * b.imag + a.imag * b.real;
}

void c_div(complex a, complex b, complex *c)
```

```
{
    c->real = (a.real * b.real + a.imag * b.imag) / (b.real * b.real + b.imag *
b.imag);
    c->imag = (a.imag * b.real - a.real * b.imag) / (b.real * b.real + b.imag *
b.imag);
}

#define SWAP(a,b)  tempr=(a);(a)=(b);(b)=tempr

void Wn_i(int n, int i, complex *Wn, char flag)
{
    Wn->real = cos(2 * PI*i / n);
    if (flag == 1)
        Wn->imag = -sin(2 * PI*i / n);
    else if (flag == 0)
        Wn->imag = -sin(2 * PI*i / n);
}

//傅里叶变化
void fft(int NN, complex f[])
{
    complex t, wn;//中间变量
    int i, j, k, m, n, l, r, M;
    int la, lb, lc;
    /*----计算分解的级数M=log2(N)----*/
    for (i = NN, M = 1; (i = i / 2) != 1; M++);
    /*----按照倒位序重新排列原信号----*/
    for (i = 1, j = NN / 2; i <= NN - 2; i++)
    {
        if (i < j)
        {
            t = f[j];
            f[j] = f[i];
            f[i] = t;
        }
        k = NN / 2;
        while (k <= j)
        {
            j = j - k;
            k = k / 2;
        }
        j = j + k;
    }
```

```
    /*----FFT算法----*/
    for (m = 1; m <= M; m++)
    {
        la = pow(2, m); //la=2^m代表第m级每个分组所含节点数
        lb = la / 2;    //lb代表第m级每个分组所含碟形单元数
                        //同时它也表示每个碟形单元上下节点之间的距离
        /*----碟形运算----*/
        for (l = 1; l <= lb; l++)
        {
            r = (l - 1)*pow(2, M - m);
            for (n = l - 1; n < NN - 1; n = n + la) //遍历每个分组，分组总数为N/la
            {
                lc = n + lb;  //n,lc分别代表一个碟形单元的上、下节点编号
                Wn_i(NN, r, &wn, 1);//wn=Wnr
                c_mul(f[lc], wn, &t);//t = f[lc] * wn复数运算
                c_sub(f[n], t, &(f[lc]));//f[lc] = f[n] - f[lc] * Wnr
                c_plus(f[n], t, &(f[n]));//f[n] = f[n] + f[lc] * Wnr
            }
        }
    }
}

//傅里叶逆变换
void ifft(int NN, complex f[])
{
    int i = 0;
    conjugate_complex(NN, f, f);
    fft(NN, f);
    conjugate_complex(NN, f, f);
    for (i = 0; i < NN; i++)
    {
        f[i].imag = (f[i].imag) / NN;
        f[i].real = (f[i].real) / NN;
    }
}

bool IsEqual2(complex idataA[], complex idataB[], const int size)
{
    for (int i = 0; i < size; i++)
    {
        if (abs(idataA[i].real - idataB[i].real) > 0.000001 || abs(idataA[i].imag -
idataB[i].imag) > 0.000001)
            return false;
    }
```

```
    return true;
}


int main()
{
    cufftComplex *data_dev; // 设备端数据头指针
    cufftComplex *data_Host = (cufftComplex*)malloc(NX*BATCH * sizeof(cufftComplex));
// 主机端数据头指针
    cufftComplex *resultFFT = (cufftComplex*)malloc(N*BATCH * sizeof(cufftComplex)); //
正变换的结果
    cufftComplex *resultIFFT = (cufftComplex*)malloc(NX*BATCH * sizeof(cufftComplex));
// 先正变换后逆变换的结果

    // 初始数据
    for (int i = 0; i < NX; i++)
    {
        data_Host[i].x = float((rand() * rand()) % NX) / NX;
        data_Host[i].y = float((rand() * rand()) % NX) / NX;
    }



    dim3 dimBlock(BLOCK_SIZE); // 线程块
    dim3 dimGrid((NX + BLOCK_SIZE - 1) / dimBlock.x); // 线程格

    cufftHandle plan; // 创建cuFFT句柄
    cufftPlan1d(&plan, N, CUFFT_C2C, BATCH);

    // 计时
    clock_t start, stop;
    double duration;
    start = clock();

    cudaMalloc((void**)&data_dev, sizeof(cufftComplex)*N*BATCH); // 开辟设备内存
    cudaMemset(data_dev, 0, sizeof(cufftComplex)*N*BATCH); // 初始为0
    cudaMemcpy(data_dev, data_Host, NX * sizeof(cufftComplex), cudaMemcpyHostToDevice);
// 从主机内存拷贝到设备内存

    cufftExecC2C(plan, data_dev, data_dev, CUFFT_FORWARD); // 执行 cuFFT，正变换
    cudaMemcpy(resultFFT, data_dev, N * sizeof(cufftComplex), cudaMemcpyDeviceToHost);
// 从设备内存拷贝到主机内存

    cufftExecC2C(plan, data_dev, data_dev, CUFFT_INVERSE); // 执行 cuFFT，逆变换
```

```
    cufftComplexScale << <dimGrid, dimBlock >> > (data_dev, data_dev, N, 1.0f / N); //
乘以系数
    cudaMemcpy(resultIFFT, data_dev, NX * sizeof(cufftComplex),
cudaMemcpyDeviceToHost); // 从设备内存拷贝到主机内存

    stop = clock();
    duration = (double)(stop - start) * 1000 / CLOCKS_PER_SEC;
    cout << "时间为 " << duration << " ms" << endl;

    cufftDestroy(plan); // 销毁句柄
    cudaFree(data_dev); // 释放空间


    if (IsEqual(data_Host, resultIFFT, NX))
        cout << "逆变化后数据相同。" << endl;
    else
        cout << "逆变换后不同" << endl;



    //cpu fft
    for (long int i = 0; i < MAX; i++) {
        m[i].real = float((rand() * rand()) % 4096) / 4096;
        l[i].real = m[i].real;
        l[i].imag = m[i].imag = 0;
    }
    clock_t start2, end2;
    double duration1;
    start2 = clock();
    fft(MAX, m);
    ifft(MAX, m);
    end2 = clock();
    duration1 = (double)(end2 - start2) * 1000 / CLOCKS_PER_SEC;
    cout << "cpu时间为 " << duration1 << " ms" << endl;

    if (IsEqual2(m, l, MAX))
        cout << "逆变换后相同" << endl;
    else
        cout << "逆变换后不同" << endl;

    return 0;
}
```

GPU 加速 FFT 程序的可能加速比

通过分析 FFT 算法代码，可以得到该 FFT 算法的并行性体现在在每轮运算中都有多数线程参与运算，相比较于 cpu 的单线程运算，加速比是无法估量的。

如果使用 GPU 进行加速，可以采用线程块 dimGrid*一块线程块中线程数 dimBlock 个线程分别计算一次蝴蝶变换，这样真不好推测出加速比。

注意上述分析中未考虑初始化、数据传递等时间，实际上相比较于运算，开销时间主要用于加载数据和传输数据。

测试

测试平台

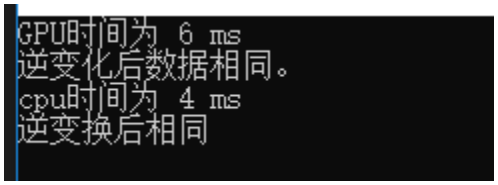
在如下机器上进行了测试：

部件	配置
CPU	core i5-6300HQ
内存	DDR4 8GB
GPU	Nvidia Geforce 960M
显存	DDR5 2GB

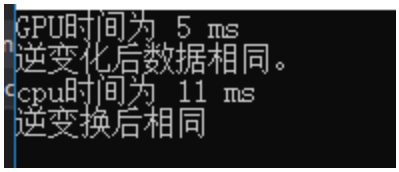
部件	配置
操作系统	Windows 10

测试记录

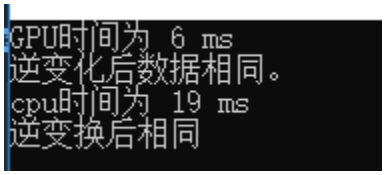
1<<12



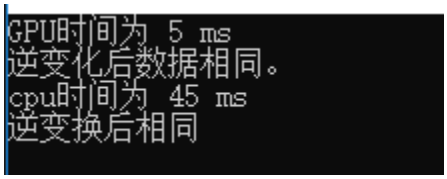
1<<13



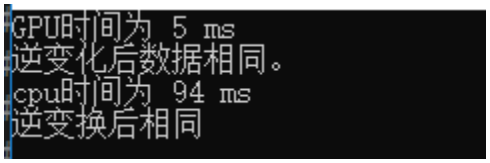
1<<14



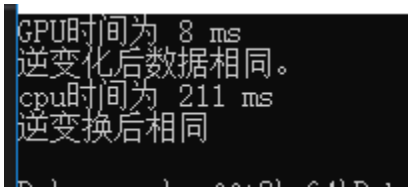
1<<15



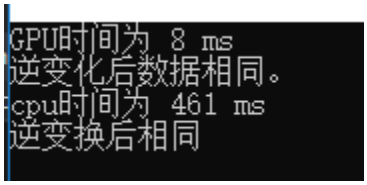
1<<16



1<<17



1<<18



分析和结论

从测试记录来看，使用 GPU 加速 FFT 程序获得的加速比仍判断不明。但肉眼可见的 gpu 处理 FFT 接近 O（1）的处理速度。

造成这种现象的原因为：

1. 数据通信所消耗的时间为预估在 3-4s 左右；
2. GPU 上线程调度开销也会造成影响，因为 GPU 上线程数是有限的，但随着数据规模扩大，一轮蝴蝶变换可能并不是所有用到的线程参与就可以完成，需要等待前一轮完成后再进行一次同阶层的蝴蝶变换，但是之后线程数又满足。
3. GPU 上线程之间访存竞争造成的影响，当同一阶层的蝴蝶变换在分配线程后哪怕只剩一次，也需要所有其他线程进行等待才能进行下一阶层。