实验报告

实验名称（用 GPU 加速 FFT 程序）

班级智能 1602 学号 201608010708 姓名田宗杭

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 的算法可以参考[这
里](https://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm)。

```
c++
/* fft.cpp
 *
 * This is a KISS implementation of
 * the Cooley-Tukey recursive FFT algorithm.
 * This works, and is visibly clear about what is happening where.
 *
 * To compile this with the GNU/GCC compiler:
 * g++ -o fft fft.cpp -lm
 *
 * To run the compiled version from a *nix command line:
 * ./fft
 *
 */
#include <complex>
#include <cstdio>
```

```
#define M_PI 3.14159265358979323846 // Pi constant with double precision

using namespace std;

// separate even/odd elements to lower/upper halves of array respectively.
// Due to Butterfly combinations, this turns out to be the simplest way
// to get the job done without clobbering the wrong elements.
void separate (complex<double>* a, int n) {
    complex<double>* b = new complex<double>[n/2];  // get temp heap storage
    for(int i=0; i<n/2; i++)    // copy all odd elements to heap storage
        b[i] = a[i*2+1];
    for(int i=0; i<n/2; i++)    // copy all even elements to lower-half of a[]
        a[i] = a[i*2];
    for(int i=0; i<n/2; i++)    // copy all odd (from heap) to upper-half of a[]
        a[i+n/2] = b[i];
    delete[] b;                 // delete heap storage
}

// N must be a power-of-2, or bad things will happen.
// Currently no check for this condition.
//
// N input samples in X[] are FFT'd and results left in X[].
// Because of Nyquist theorem, N samples means
// only first N/2 FFT results in X[] are the answer.
// (upper half of X[] is a reflection with no new information).
void fft2 (complex<double>* X, int N) {
    if(N < 2) {
        // bottom of recursion.
        // Do nothing here, because already X[0] = x[0]
    } else {
        separate(X,N);      // all evens to lower half, all odds to upper half
        fft2(X,     N/2);   // recurse even items
        fft2(X+N/2, N/2);   // recurse odd  items
        // combine results of two half recursions
        for(int k=0; k<N/2; k++) {
            complex<double> e = X[k    ];   // even
            complex<double> o = X[k+N/2];   // odd
                             // w is the "twiddle-factor"
            complex<double> w = exp( complex<double>(0,-2.*M_PI*k/N) );
            X[k    ] = e + w * o;
            X[k+N/2] = e - w * o;
        }
    }
}

// simple test program
int main () {
```

```
    const int nSamples = 64;
    double nSeconds = 1.0;                      // total time for sampling
    double sampleRate = nSamples / nSeconds;    // n Hz = n / second
    double freqResolution = sampleRate / nSamples; // freq step in FFT result
    complex<double> x[nSamples];                // storage for sample data
    complex<double> X[nSamples];                // storage for FFT answer
    const int nFreqs = 5;
    double freq[nFreqs] = { 2, 5, 11, 17, 29 }; // known freqs for testing

    // generate samples for testing
    for(int i=0; i<nSamples; i++) {
        x[i] = complex<double>(0.,0.);
        // sum several known sinusoids into x[]
        for(int j=0; j<nFreqs; j++)
            x[i] += sin( 2*M_PI*freq[j]*i/nSamples );
        X[i] = x[i];        // copy into X[] for FFT work & result
    }
    // compute fft for this data
    fft2(X,nSamples);

    printf("  n\tx[]\tX[]\tf\n");       // header line
    // loop to print values
    for(int i=0; i<nSamples; i++) {
        printf("% 3d\t%+.3f\t%+.3f\t%g\n",
            i, x[i].real(), abs(X[i]), i*freqResolution );
    }
}

// 利用 cuda 进行的 fft 算法代码
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include<Windows.h>
// Include CUDA runtime and CUFFT
#include <cuda_runtime.h>
#include <cufft.h>
// Helper functions for CUDA
//#include "device_launch_parameters.h"

#define pi 3.1415926535
#define LENGTH 100000 //signal sampling points
int main()
{
    // data gen
    float Data[LENGTH] = { 2,5,11,17,29};
    float fs = 1000000.000;//sampling frequency
    float f0 = 200000.00;// signal frequency
```

```
    DWORD start2, end2;
     start2 = GetTickCount();
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
     cudaEventRecord(start, 0);
    for (int i = 0; i < LENGTH; i++)
    {
        Data[i] = 1.35*cos(2 * pi*f0*i / fs);//signal gen,
    }

    cufftComplex *CompData = (cufftComplex*)malloc(LENGTH * sizeof(cufftComplex));//allocate
memory for the data in host
    int i;
    for (i = 0; i < LENGTH; i++)
    {
        CompData[i].x = Data[i];
        CompData[i].y = 0;
    }
     //end2 = GetTickCount();
   //cudaEventRecord(stop, 0);
   //cudaEventSynchronize(stop);
   //float time;
    //cudaEventElapsedTime(&time, start, stop);
    cufftComplex *d_fftData;
    cudaMalloc((void**)&d_fftData, LENGTH * sizeof(cufftComplex));// allocate memory for the
data in device
    cudaMemcpy(d_fftData, CompData, LENGTH * sizeof(cufftComplex), cudaMemcpyHostToDevice);//
copy data from host to device

    cufftHandle plan;// cuda library function handle
    cufftPlan1d(&plan, LENGTH, CUFFT_C2C, 1);//declaration
    cufftExecC2C(plan, (cufftComplex*)d_fftData, (cufftComplex*)d_fftData,
CUFFT_FORWARD);//execute
    cudaDeviceSynchronize();//wait to be done
    cudaMemcpy(CompData, d_fftData, LENGTH * sizeof(cufftComplex), cudaMemcpyDeviceToHost);//
copy the result from device to host
   end2 = GetTickCount();
   cudaEventRecord(stop, 0);
   cudaEventSynchronize(stop);
   float time;
    cudaEventElapsedTime(&time, start, stop);
    for (i = 0; i < LENGTH / 2; i++)
    {
        printf("i=%d\tf= %6.1fHz\tRealAmp=%3.1f\t", i, fs*i / LENGTH, CompData[i].x*2.0 /
LENGTH);
        printf("ImagAmp=+%3.1fi", CompData[i].y*2.0 / LENGTH);
        printf("\n");
    }

    printf("cpu time: %d ms\n", end2 - start2);
    printf("gpu time = %3.1f ms\n", time);
```

```
    cufftDestroy(plan);
    free(CompData);
    cudaFree(d_fftData);
     getchar();
}
```

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节点可以接受直线边传来的数据与交叉边传来的数据，通过两点 FFT 蝶形计算方法可求出下一列向量，以此类推，知道第 logN 列才变为用原始向量进行计算，所以，经过 logN 步并行计算后，可以计算出一个 N 点一维 FFT。如果使用 GPU 进行加速运算，可能得到的加速比为 4

注意上述分析中未考虑初始化、数据传递等时间，实际加速比可能要比理想情况低。

测试

测试平台

在如下机器上进行了测试：

部件	配置	备注
CPU	core i7-6700U	
内存	DDR3 16GB	
GPU	NVIDIA GeForce GTX 960M	
显存	DDR5 4GB	
操作系统	Windows10 x64	中文版

测试记录

FFT 程序的测试输入文件请见[这里](./test.input)。

FFT 程序运行过程的截图如下：

CPU 上 FFT 程序的执行输出

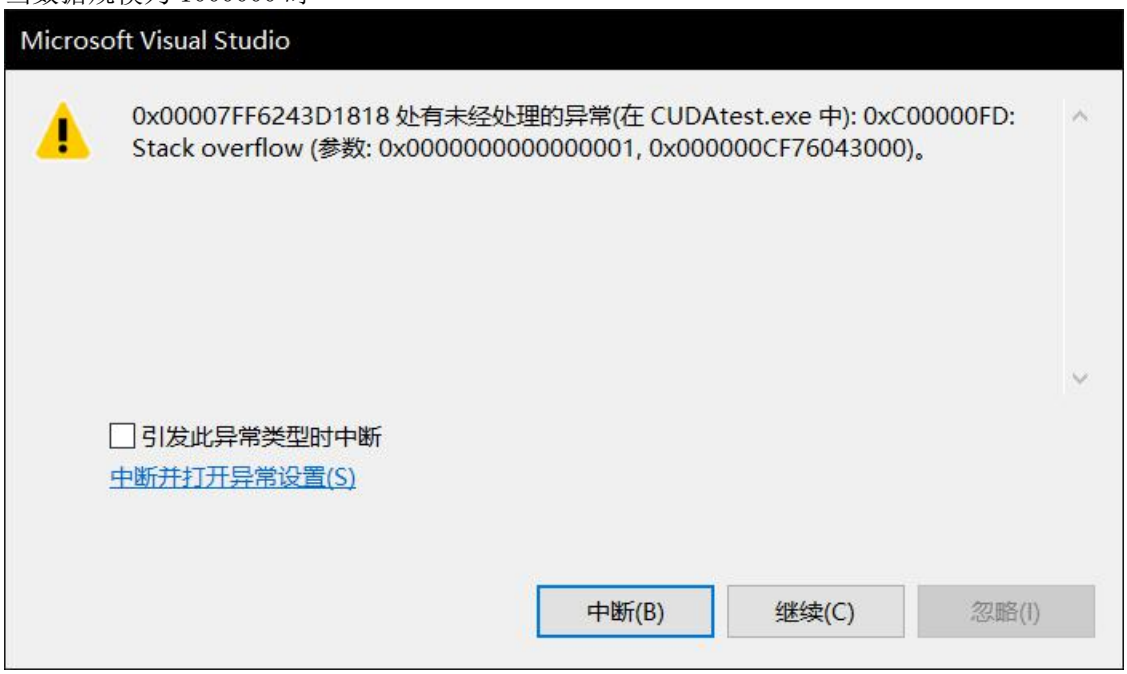
![图 1 CPU 执行时间](./perf_ls.png)
GPU 上 FFT 程序的执行输出

![图 2 GPU 执行时间](./perf_ls.png)
当数据规模为 10 时

当数据规模大小为 100 时

当数据规模为 1000 时

```
cpu time: 1703 ms
gpu time = 467.8 ms
```
当数据规模为 10000 时
```
cpu time: 1687 ms
gpu time = 468.3 ms
```
当数据规模为 100000 时
```
cpu time: 1688 ms
gpu time = 457.7 ms
```
当数据规模为 1000000 时



数据规模过大引发中断，程序停止运行。

数据规模	CPU 运行时间	GPU 运行时间	加速比 GPU/CPU
10	1656ms	451.4ms	3.6686
100	1672ms	473.5ms	3.5312
1000	1703ms	467.8ms	3.6404
10000	1687ms	468.3ms	3.6024
100000	1688ms	457.7ms	3.6880

分析和结论

从测试记录来看，使用 GPU 加速 FFT 程序获得的加速比为 3.62212，因为 CPU 做的是串行计算，所有的程序都要能够很好的处理，不能搞特殊化，所以在设计上使用了大量的晶体管用于片上缓存和控制与判断的逻辑电路。实际上只有大约 20%的晶体管用于运算单元。其次，GPU 是属于并行处理器，控制和缓存电路相对少很多，所以 80%的晶体管数量用于运算单元。并且同时期的 GPU 晶体管数量远远高于 CPU。。

相对于理想加速比为 4 的情况，造成这种现象的原因为：

1. 初始化数据所消耗的时间为 cpu 消耗时间较大，gpu 消耗时间几乎为 0.

数据规模为 10

```
cpu time: 1219 ms
gpu time = 0.0 ms
```

数据规模为 100

```
cpu time: 1219 ms
gpu time = 0.0 ms
```

数据规模为 1000

```
cpu time: 1266 ms
gpu time = 0.0 ms
```

数据规模为 10000

```
cpu time: 1219 ms
gpu time = 0.0 ms
```

数据规模为 100000

```
cpu time: 1234 ms
gpu time = 0.0 ms
```

2. 数据通信所消耗的时间为；

数据规模为 10 时

```
cpu time: 0 ms
gpu time = 0.1 ms
```

数据规模为 100

```
cpu time: 0 ms
gpu time = 0.1 ms
```

数据规模为 1000

```
cpu time: 0 ms
gpu time = 0.0 ms
```

数据规模为 10000

```
cpu time: 0 ms
gpu time = 0.4 ms
```

数据规模为 100000

```
cpu time: 0 ms
gpu time = 0.3 ms
```

3. GPU 上线程调度开销也会对 gpu 加速造成影响；
4. GPU 上线程之间访存竞争也会对 gpu 加速造成的影响；