实验报告

实验名称（用GPU加速FFT程序）

物联1601 201601110208 方缙

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

CPU FFT的算法。

```
/* 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 );
    }
}

// eof
```

使用CUDA的FFT算法

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include<Windows.h>
#include <cuda_runtime.h>
#include <cufft.h>

#define pi 3.1415926535
#define LENGTH 100000
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;
        }
        cufftComplex *d_fftData;
        cudaMalloc((void**)&d_fftData, LENGTH * sizeof(cufftComplex));
        cudaMemcpy(d_fftData, CompData, LENGTH * sizeof(cufftComplex),
cudaMemcpyHostToDevice);
        cufftHandle plan;
        cufftPlan1d(&plan, LENGTH, CUFFT_C2C, 1);
        cufftExecC2C(plan, (cufftComplex*)d_fftData,
(cufftComplex*)d_fftData,CUFFT_FORWARD);
        cudaDeviceSynchronize();//wait to be done
        cudaMemcpy(CompData, d_fftData, LENGTH * sizeof(cufftComplex),
cudaMemcpyDeviceToHost);
                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

由于直接使用CUDA自带的cufft库，所以不知道内部具体细节，从官方文档可以看到cufft库针对基为2、3、57的数据规模有优化，其中基为2的数据优化最大，算法最快。

- Algorithms highly optimized for input sizes that can be written in the form 2 a × 3 b × 5 c × 7 d . In general the smaller the prime factor, the better the performance, i.e., powers of two are fastest.

算法复杂度为O(nlogn)。

- An O (n log n) algorithm for every input data size

从CUDA的文档中可以看到cufft针对两种数据规模使用了不同的算法。以2-127为底的数据规模使用Cooley-Tukey algorithm ，其他的数据规模则使用Bluestein's algorithm。由于我们的测试数据均以2或者10为底所以只需讨论Cooley-Turkey algorithm。

通过测试由于我的显卡只有三个处理器核心，所以相对于单线程FFT最大加速比应该是3.

```
Device 0: "GeForce GPU"
  CUDA Driver Version / Runtime Version          10.0 / 10.0
  CUDA Capability Major/Minor version number:    5.0
  Total amount of global memory:                 1024 MBytes (1073741824 bytes)
  ( 3) Multiprocessors, (128) CUDA Cores/MP:     384 CUDA Cores
```

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 i7-6600U	
内存	DDR4 16GB	
GPU	Nvidia Geforce GPU	surface book特制的，具体差不多是gtx 940m
显存	DDR5 1GB	
操作系统	windows 10	1809版本

测试记录

cufft程序运行过程的截图如下：

```
cpu time: 1297 ms
gpu time = 424.9 ms
```

其他数据规模测试结果如下

数据规模	CPU time(ms)	GPU time(ms)
8	1032	494.9
10	844	440.3
100	1953	639.4
128	1297	424.9
1000	1797	699.7
1024	1265	493.0
10000	1953	550.3
100000	1500	811.8

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2.88，相对于理想情况，加速比有所降低。

数据规模为128的时候GPU时间比100的时候要少大约一半，数据规模为1000和1024的时候同样，说明算法对底为2的数据规模进行了一部分优化。而数据规模为8和10的时候差距则不大，可能原因是数据规模太小，导致cache会有一些冲突。

而数据规模越大，可以看出GPU的加速比越大，但是在数据规模达到100000的时候加速比又减小了，可能原因是数据规模太大，显卡存储单元不够用。

造成这种现象的原因为：

1. 虽然显卡并行度比较高，但是由于显卡本身的频率较低，所以没有达到理想情况
2. 在测试过程中还有其他进程，所以操作系统的调度也会影响测试。
3. GPU上线程调度开销也会造成影响。
4. GPU上线程之间访存竞争造成的影响。