实验报告

实验名称（用 **GPU** 加速 **FFT** 程序）

物联 1601　201608010628　曾彤芳

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.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 programint 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
```

```cpp
/* 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>
#include<ctime>
#include<iostream>

#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 () {
    clock_t start,finish;
    double totaltime;
    start=clock();
    int count=0;
while(count<3){
//while(count<31){
//while(count<312){
//while(count<3125){
    count++;
    const int nSamples = 16;
    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 );
    }
}
    finish=clock();
    totaltime=double (finish-start);
    cout<<"yunxingshijian: "<<totaltime/CLOCKS_PER_SEC<<endl;
}

// eof
```

GPU 加速 FFT 程序的可能加速比

通过分析 FFT 算法代码，FFT 的时间复杂度为 o(n log n)，FFT 卷积复杂度为 3 次 FFT+L 次乘法，3o(n log n)+o(n)=o(n log n)，及 o(n log n)。在 CUDA 编程模型中，要获得较高的加速效果，必须要有足够多的活跃线程才能提高计算效能，隐藏访存延时。因此需要充分挖掘算法的可并行性。根据 CUDA 多线程执行模式的特点，以及对 FFT 算法的分析，将算法按级划分，级内的 N/2 个蝶形完全

并行，级间是数据通信，前级蝶形运算的结果存回原位置 ，次级运算时按照蝶形路径索引的地址从新位置读取操作数据. 前一级产生的数据作为后一级的输入，数据在各级操作间流动起来设定 N/2 个线程，每个线程负责从第 1～log2N 级的一路蝶形运算，并在每一级完成后进行同步，如图 4 所示。每个蝶形的两个操作数的地址由线程号和当前级数的函数确定, 蝶形计算完毕后存入原地址，即采用同址方式。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

FFT 程序运行过程的截图如下：

N=50

CPU 上 FFT 程序的执行输出:

```
yunxingshijian: 0.00048
```

GPU 上 FFT 程序的执行输出:

```
total time :1.648000 s
```

N=500

CPU 上 FFT 程序的执行输出:

```
yunxingshijian: 0.02095
```

GPU 上 FFT 程序的执行输出:

```
total time :2.221000 s
```

N=5000

CPU 上 FFT 程序的执行输出:

```
yunxingshijian: 0.166304
```

GPU 上 FFT 程序的执行输出:

```
total time :9.403000 s
```

N=50000

CPU 上 FFT 程序的执行输出:

```
yunxingshijian: 1.73297
```

GPU 上 FFT 程序的执行输出:

```
total time :36.037000 s
```

数据规模	CPU 运行时间	GPU 运行时间	CPU/GPU
50	0.0005	1.648	0.03
500	0.0209	2.221	0.94
5000	0.1663	9.403	1.77
50000	1.7329	36.037	4.81

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 4.81，相对于理想情况，加速比降低，数据规模小时，GPU 更耗时，数据规模大时，GPU 下运行速度明显更快。

造成这种现象的原因为：

1. 初始化所消耗的时间；
2. 数据通信所消耗的时间；
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。