实验报告

实验名称（用 **GPU** 加速 **FFT** 程序）

智能 1501　　201508010513　　王鑫淼

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 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
```

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每次计算时，都会有多个线程同时计算，这回节省很多时间。

如果使用 GPU 进行加速，可以分别计算在 CPU 上运行的时间与在 GPU 上运行的时间，这样可能得到的加速比可能有一些误差，不会和理想情况下的加速比一样，这是由于未考虑初始化、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 i5-5200U	

部件	配置	备注
内存	DDR3 4GB	
GPU	AMD Radeon R5 M330	
显存	DDR5 2GB	
操作系统	Windows 8	

测试记录

当 n=1000 时，
CPU 上 FFT 程序的执行输出

```
time: 7.043000 ms
```

GPU 上 FFT 程序的执行输出

```
GPU time: 4.481000 ms
```

当 n=5000 时，
CPU 上 FFT 程序的执行输出

```
time: 43.434000 ms
```

GPU 上 FFT 程序的执行输出

```
GPU time: 23.937000 ms
```

当 n=10000 时，
CPU 上 FFT 程序的执行输出

```
time: 89.632000 ms
```

GPU 上 FFT 程序的执行输出

```
GPU time: 43.283000 ms
```

当 n=50000 时，
CPU 上 FFT 程序的执行输出

```
time: 420.378000 ms
```

GPU 上 FFT 程序的执行输出

```
GPU time: 208.887000 ms
```

当 n=100000 时，

CPU 上 FFT 程序的执行输出

```
time: 839.939000 ms
```

GPU 上 FFT 程序的执行输出

```
GPU time: 425.090000 ms
```

将所有结果综合起来得到

数据规模	CPU	GPU	加速比
1000	7.043	4.481	1.572
5000	43.43	23.937	1.865
10000	89.632	43.283	2.071
50000	420.378	208.887	2.012
100000	839.939	425.09	1.976

分析和结论

从测试记录来看，使用 GPU 加速 FFT 程序获得的加速比基本上保持在 2 倍左右，当数据规模较小时，加速比还会变小为，所以当规模量非常小时，CPU 上的运行速度反而更快。

造成这种现象的原因为：

1. 初始化要消耗的时间；
2. 数据通信要消耗的时间.；
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。