实验报告

实验名称（用 GPU 加速 FFT 程序）

智能 1602 201608010609 李鹏飞

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 的代码如下：

```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 );
}
}

// eof
```

利用 cuda 进行 FFT 代码：

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
// Include CUDA runtime and CUFFT
#include <cuda_runtime.h>
#include <cufft.h>
#include<iostream>
// Helper functions for CUDA
//#include <helper_functions.h>
//#include <helper_cuda.h>
//#include "device_launch_parameters.h"

#define pi 3.1415926535
#define LENGTH 100000 //signal sampling points
using namespace std;
int main()
{
clock_t start,finish;
   start = clock();
      int i;
 //   for( i = 0 ;i<30 ;i++){
    // data gen
    float Data[LENGTH] = {1,2,3,4};
    float fs = 1000000.000;//sampling frequency
    float f0 = 200000.00;// signal frequency
    for( 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

    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));// 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

  for(i=0;i<LENGTH/2;i++)
  {
   //if(CompData[i].x != 0)
   //{
     printf("i=%d\tf= %6.1fHz\tRealAmp=%3.1f\t",i,fs*i/LENGTH,CompData[i].x*2.0/LENGTH);//print the result:
```

```
    //}
    //if(CompData[i].y != 0 )
    //{
      printf("ImagAmp=+%3.1fi",CompData[i].y*2.0/LENGTH);
   // }
    printf("\n");
  }
     cufftDestroy(plan);
     free(CompData);
     cudaFree(d_fftData);
finish = clock();
   cout<<(finish - start) <<"/"<<CLOCKS_PER_SEC << "(s)"<<endl;
//    }
}
```

GPU 加速 FFT 程序的可能加速比

通过分析 FFT 算法代码，我们预期的优化部分为变换过程中 x 轴与 y 轴的计算过程。

但是因为直接调用 cufft 库，无法看到内部加速过程，最终加速比由实验测试结果进行推断。

注意理论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

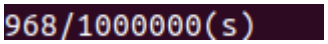
在如下机器上进行了测试：

| 部件 | 配置 | 备注 |

| :--------|:----------------:| :-----:|
CPU	core i5-6500U	
内存	DDR4 16GB	
GPU	Nvidia Geforce 950M	
显存	DDR5 6GB	
操作系统	Ubuntu 18.04 LTS	中文版

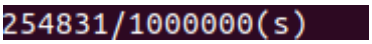
测试记录

数据集 64
CPU:



图一　64 数据 CPU 结果

GPU:

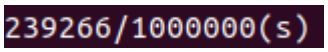


图二　64 数据 GPU 结果

数据集 1000
CPU:

图三　1000 数据 CPU 结果

GPU:

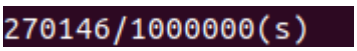


图四　1000 数据 GPU 结果

数据集 10000
CPU:

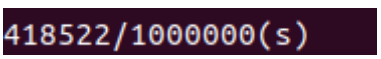
图五　10000 数据 CPU 结果

GPU:



图六　10000 数据 GPU 结果

数据集 100000
CPU:



图七　100000 数据 CPU 结果

GPU:

图八　100000 数据 GPU 结果

数据集 1000000
CPU:

图九　1000000 数据 CPU 结果

GPU:

图十　1000000 数据 CPU 结果

我们列一个表格进行进一步说明

数据集大小	CPU 运行时间/1000000	GPU 运行时间/1000000	比值(CPU/GPU)
64	968	254831	0.00379
1000	5468	239266	0.0228
10000	45255	270146	0.167
100000	418522	439724	0.951
1000000	4301728	2197316	1.957

分析和结论

从测试记录来看，使用 GPU 加速 FFT 程序获得的加速比在理想的情况下为 1.975，因为 GPU 加速的优势只有在数据量大时才能够体现，当数据量小的时候，更多的时间用于数据拷贝和传输，所以反而性能比 CPU 要差很多.。

造成实际加速比与理想加速比不同的原因为：

1. 程序的初始化需要消耗时间；
2. 数据之间的通信需要消耗时间；
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，也会影响最终结果。