Discrete Fast Fourier Transform

Severin Davis

Teja Dharma

High Performance Computing with Graphic Cards

University of Stuttgart

Contents

Motivation

Fast Fourier Transforms are useful in a variety of applications, most notably signal processing. If signal processing must occur as fast as possible or with a very large input size, parallelization of a Fast Fourier Transform algorithm can reap large benefits in required processing time.
In this project, a Fast Fourier Transform algorithm is implemented. Initially it is implemented as single threaded, serially-executing program. This implementation is analyzed for candidates for parallelization which are then moved to an OpenCL kernel.
These two implementations are compared both in theoretical complexity as a function of input size, as well as real world execution speed on multiple platforms.

How-to

Import the FFT project into eclipse.
After running the program, the results of the algorithm should be printed.

The number of elements in the input and the input is printed.
The FFT CPU results are printed.
The FFT GPU results are printed.
These results are automatically compared and checked for errors.
The speed test for the algorithm is run and the results are printed.

The input file is included in the deliverable. This file is named “input.txt”. This can be modified to provide different inputs, as long as the format of the example is maintained.

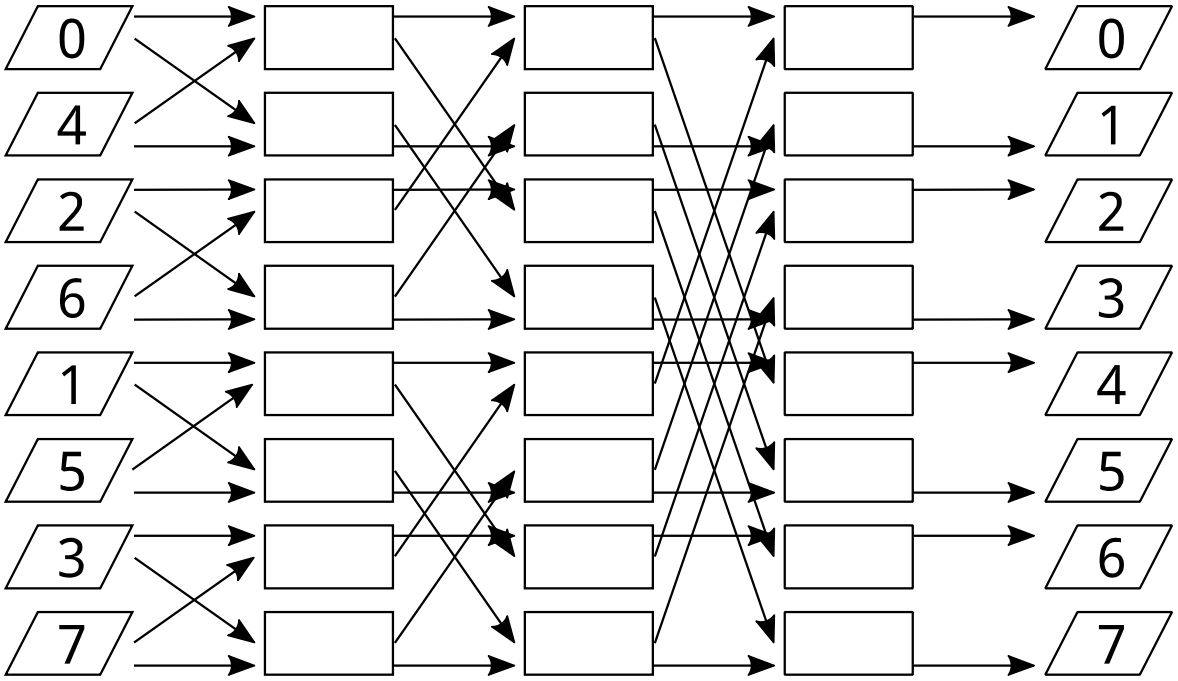
The speed test input cannot be changed, it’s purpose is solely to test the performance of the implementation with various input sizes. The results of these FFT executions are not returned. For the speed test, only the execution times are printed, both to the consoles and to “output.txt”

Program structure

For this Fast Fourier Transform implementation, the Tukey-Cooley Algorithm was chosen. This algorithm works in stages in which two independent indices of the input array interact. The results of these interactions are placed back into the array and the algorithm is run in a chained manner, in which the results of previous stages are used in the current stage. Each interaction involves two indices and a computed root of unity. Additionally, the input must be placed in a specific order. As such, our implementation could be broken down into three sub problems, each with their unique challenges.

1. Reordering of the input
2. Mapping and implementation of interactions
3. Computation and mapping of roots of unity

Each of these is visible in the diagram of an example input size of 8 below.
The reordering of the input is visible on the input nodes at the left of the diagram. The interacting index pairs, or stages, are shown as arrows. The boxes shown indicate the computations performed involving the roots of unity and the two inputs.



Reordering of the input

Inputs need to be reordered before the algorithm can be run. Unfortunately, because this reordering does not operate on independent indices, this portion of the FFT implementation was not a candidate for parallelization. Instead, this was implemented as a recursive function that executes prior to any FFT execution.

```
std::vector<doublec> reorder_input(std::vector<doublec> input)
{
	//check base case
	if (input.size() <= 2)
		return input;

	//create even and odd index reorder vectors
	std::vector<doublec> even(input.size()/2);
	std::vector<doublec> odd(input.size()/2);

	//split even and odd indices
	for (unsigned int i = 0; i < even.size(); i++)
	{
		even.at(i) = input.at(i * 2);
		odd.at(i) = input.at((i * 2) + 1);
	}

	//recursive call to reorder even and odd index vectors
	even = reorder_input(even);
	odd = reorder_input(odd);

	//regroup returned vectors and return
	std::vector<doublec> result;
	result.reserve(input.size());
	result.insert(result.end(), even.begin(), even.end());
	result.insert(result.end(), odd.begin(), odd.end());

	return result;
}
```

Mapping and implementation of interactions

The number of interactions occurring at each stage is half that of the input size. For example, in the given diagram, the sample size is 8. The left column of arrows,

however, always shows two pairs of nodes interacting, creating a total of four interactions in that stage. Similarly, stages 2 and 4 also have 4 interacting pairs. As such, the number of threads spawned is half of the input size.

Thread mapping

One major challenge to this approach was finding the corresponding mapping from the thread number to the nodes it operates on in a specific stage.

As an example, the diagram shows thread0 mapped to index 0 and index 1 of the input array. Similarly, thread 0 is mapped to index 0 and 2 and index 0 and 4 at stages 2 and 4, respectively.

By analyzing the pattern, the following mapping was found to output an index, given the thread number and the current stage.

```
unsigned int thread_index_map(unsigned int thread_id, unsigned int stage)
{
      return (stage * 2 * (thread_id / stage)) + thread_id % stage;
}
```

Finally, after retrieving the one thread to index mapping, the other interacting index can be found simply by adding the current stage to the returned mapping.

Interaction implementation

Once the correct mapping is found, the two data items at those indices interact with each other, as well as an additional root of unity.

This interaction is defined as:

result[home_index] = result[home_index] + (result[target_index] * root)

result[target_index] = result[home_index] – (result[target_index] * root)

Of interest is the fact that the root for a particular interaction remains constant. As such, the (target_index*root) can be computed only once, further optimizing for speed of execution.

This is implemented in the code shown:

```
doublec pq = doublec_mul(result.at(target_index), roots.at(home_root));
doublec top = doublec_add(pq, result.at(home_index));
doublec bottom = doublec_sub(result.at(home_index), pq);
```

Computation and mapping of roots of unity

The required degree of the roots of unity for this FFT implementation is directly related to the size of the input. For instance, and input of size 8 will require 8 roots of unity. As such, instead of computing the necessary root for that particular stage and thread, the required roots are computed at the beginning of the algorithm and are simply referenced later via an additional mapping function.

Additionally, because roots are symmetric around the unit circle, we can save time by simply computing half of them. For example, an input size of 8 will only require 4 roots to be computer. Coincidentally, since 4 threads are spawned for the same input size, each thread is tasked with computing one root.

This computation is shown below:

```
for (unsigned int i = 0; i < roots.capacity(); i++)
{
	double arg = (2 * pi * (double)i) / ((double)result.capacity());
	roots.at(i).real = cos(arg);
	roots.at(i).imag = -1 * sin(arg);
}
```

Like the thread to index mapping function, threads need a mapping function to find the correct root to use, as shown below:

```
unsigned int thread_root_map(unsigned int thread_id, unsigned int estage,
unsigned int istage)
{
	return istage * (thread_id % estage);
}
```

In this function, estage counts up (1, 2, 4, 8, ...) while istage counts in reverse (..., 8, 4, 2, 1).

Parallelization and expected results

Examination of the diagram of the FFT algorithm reveals that the number of stages of the algorithm is logarithmically related to the size of the input. Similarly, the number of operations performed within each stage is linearly related to the size of the input. Additionally, the non-recurring cost of computing the roots is also linearly related to the size of the input.
Thus, the runtime complexity of the serially executing algorithm O(n+n*log(n)) where n is the size of the input. This reduces to just O(n*log(n)).
Analyses of the algorithm reveals several two primary candidates for optimization:

1. Root computation
2. FFT algorithm

Parallelization

Root computation

Parallelizing the root computation simply maps one root computation to each thread. Thus, the runtime complexity for this portion reduces from O(n) to O(1).

FFT algorithm

The stages of the algorithm cannot be parallelized, as they rely on previous stages' results. However, the interactions within each stage are easily parallelized by mapping each interaction to its own thread. Thus, the runtime complexity for this portion reduces from O(n*log(n)) to O(log(n)).

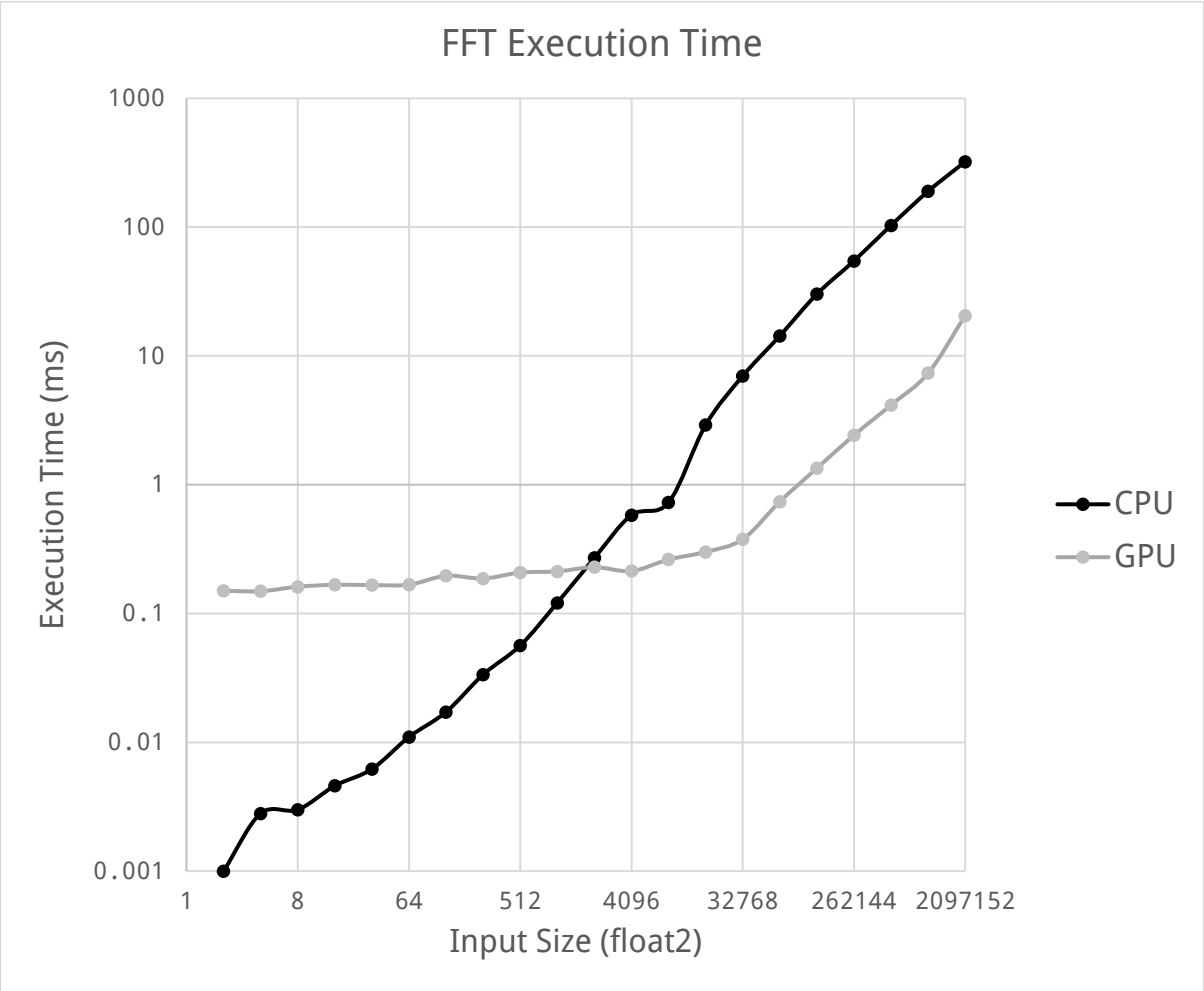
Overall this results in a runtime complexity reduction from O(n*log(n) to O(log(n)).

Expected Results

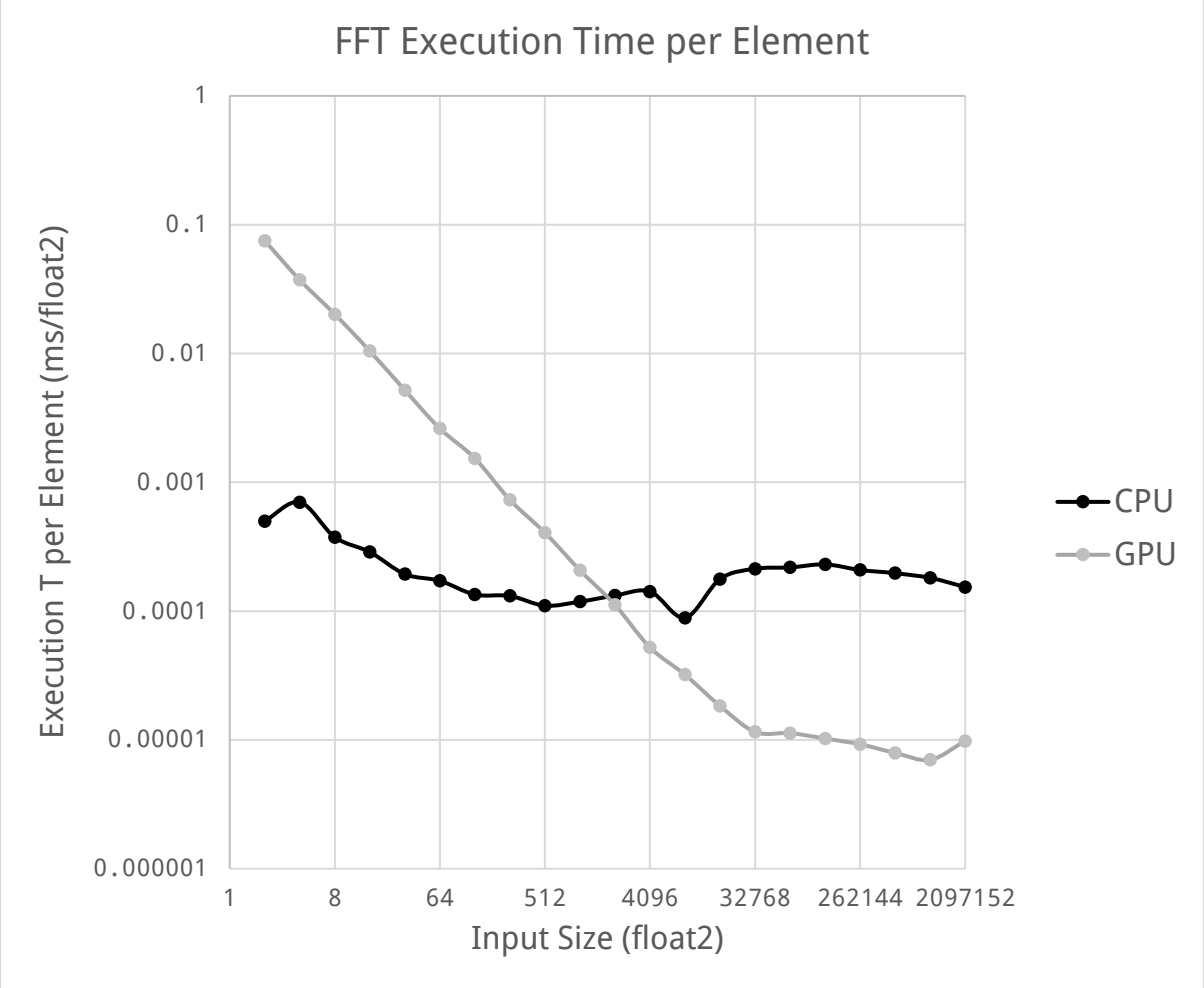
Timing results from the implementation are expected to roughly follow these complexities.
However, there is an overhead cost associated with moving data the OpenCL device as well as an associated cost in starting up the computation on the device. This introduces an overhead and causes a breakeven point in the total run time. This means that for dataset inputs below a certain size, the OpenCL overhead would cause the theoretically faster OpenCL implementation to be slower.

Results

The implementation was executed 5 times over a large range of input sizes. The results were averaged with each other to minimize spurious results.
These averaged results were graphed.



This graph clearly shows the breakeven point. This point is located at the intersections of the CPU and GPU lines. Thus, any dataset with a size of around 2048 or larger should be run on the GPU implementation as the execution time cost is lower. Sizes below this should not be run on the GPU as the GPU overhead kills destroys its parallelized execution improvements.



This graph more clearly shows the overhead cost of executing on a GPU. The long, negative slope line from 2 to about 32768 relates well to the increasing size of the input. This indicates that most of the execution time on the GPU seen in this range is the overhead. From 32768 to the end of the graph, this overhead appears to amortized and we see the true execution time per element.

Sources

Appendix

FFT.cpp

```
#include <stdio.h>

#include <Core/Assert.hpp>
#include <Core/Time.hpp>
#include <OpenCL/cl-patched.hpp>
#include <OpenCL/Program.hpp>
#include <OpenCL/Event.hpp>
#include <OpenCL/Device.hpp>

#include <fstream>
#include <sstream>
#include <iostream>
#include <cmath>

#include "doublec.hpp"

using namespace Core;
using namespace std;

#define pi 3.14159265359f

int read_input(std::vector<cl_float2>* input, string filename);
void print_vector(std::vector<cl_float2>* input);
std::vector<cl_float2> reorder_input(std::vector<cl_float2> input);
unsigned int thread_index_map(unsigned int thread_id, unsigned int
stage);
unsigned int thread_root_map(unsigned int thread_id, unsigned int estage,
unsigned int istage);

void print_vector(std::vector<cl_float2>* input)
{
	cout << endl;
	for (unsigned int i = 0; i < input->size(); i++)
	{
		cout << doublec_to_string(input->at(i)) << endl;
	}
}

int read_input(std::vector<cl_float2>* input, string filename)
{
	input->clear();
	//read file
	ifstream inputFile;

	inputFile.open(filename.c_str(), ifstream::in); //input.csv
	//verify that file is open
	if (inputFile.is_open())
	{
		//log success
	}
	else
	{
		//log fail
		return -1;
```

```
	}

	string inputLine;

	//count number of lines (samples) in file
	while (getline(inputFile, inputLine))
	{
		cl_float2 sample;
		sample.y = 0;
		try
		{
			double line = ::atof(inputLine.c_str());
			sample.x = line;
		}
		catch (...)
		{
			return -1;

		}
		input->push_back(sample);
	}

	//check sample size for usability
	//TODO
	if (input->size() < 2)
	{
		return -1;
	}

	unsigned int powertwo = 2;

	//find smallest power of two larger than the input vector
	while (powertwo < input->size())
		powertwo = powertwo * 2;

	//pad input vector with 0s until it's a power of two size
	while (input->size() != powertwo)
	{
		cl_float2 sample;
		sample.x = 0;
		sample.y = 0;
		input->push_back(sample);
	}
	return true;
}


std::vector<cl_float2> reorder_input(std::vector<cl_float2> input)
{
	//check base case
	if (input.size() <= 2)
		return input;

	//create even and odd index reorder vectors
	std::vector<cl_float2> even(input.size()/2);
	std::vector<cl_float2> odd(input.size()/2);

	//split even and odd indices
	for (unsigned int i = 0; i < even.size(); i++)
```

```
	{
		even.at(i) = input.at(i * 2);
		odd.at(i) = input.at((i * 2) + 1);
	}

	//recursive call to reorder even and odd index vectors
	even = reorder_input(even);
	odd = reorder_input(odd);

	//regroup returned vectors and return
	std::vector<cl_float2> result;
	result.reserve(input.size());
	result.insert(result.end(), even.begin(), even.end());
	result.insert(result.end(), odd.begin(), odd.end());

	return result;
}

std::vector<cl_float2> fft_cpu(std::vector<cl_float2> input)
{
	std::vector<cl_float2> result(input);
	std::vector<cl_float2> roots;
	roots.reserve(input.size()/2);

	//populate root vector with empty complex numbers
	for (unsigned int i = 0; i < roots.capacity(); i++)
	{
		cl_float2 root;
		root.x = 0;
		root.y = 0;
		roots.push_back(root);
	}

	//calculate half of the roots
	//we do this to match the number of threads (size/2) that run on
the OpenCL implementation
	for (unsigned int i = 0; i < roots.capacity(); i++)
	{
		double arg = (2 * pi * (double)i) /
((double)result.capacity());
		roots.at(i).x = cos(arg);
		roots.at(i).y = -1 * sin(arg);
	}

#ifdef DEBUG
	cout << "unity roots";
	print_vector(&roots);
	cout << endl;
#endif

	//threads always operate on pairs of indices in each stage
	//so we start half as many threads as the input size
	unsigned int num_of_threads = (unsigned int)(result.size() / 2);

	//we maintain a power of two up counter and a power of two down
counter to avoid exp or log functions
	//here we loop through the stages which has a runtime complexity of
log(n)
	unsigned int istage = num_of_threads;
```

```
	for (unsigned int estage = 1; estage <= num_of_threads; estage =
estage * 2)
	{
#ifdef DEBUG
		cout << "STAGE " << estage << endl;
#endif
		//the inner loops simply executes our "threads".
		//this will be parallelized in the opencl implementation
		//runtime for this is n
		//total for fft serial execution is nlogn
		for (unsigned int thread_id = 0; thread_id < num_of_threads;
thread_id++)
		{
			//we query for our home and target indeces
			//as well as our home and target root indeces
			unsigned int home_index = thread_index_map(thread_id,
estage);
			unsigned int target_index = home_index + estage;
			unsigned int home_root = thread_root_map(thread_id,
estage, istage);
#ifdef DEBUG
			cout << "t" << thread_id << ": " << "hi" << home_index
<< ", ti" << target_index << endl;
			cout << "t" << thread_id << ": " << "hr" << home_root
<< endl;
#endif

			cl_float2 pq = doublec_mul(result.at(target_index),
roots.at(home_root));
			cl_float2 top = doublec_add(pq, result.at(home_index));
			cl_float2 bottom = doublec_sub(result.at(home_index),
pq);
			result.at(home_index) = top;
			result.at(target_index) = bottom;
		}
#ifdef DEBUG
		cout << endl;
#endif
		istage = istage / 2;
	}
	return result;
}

unsigned int thread_index_map(unsigned int thread_id, unsigned int stage)
{
	return (stage * 2 * (thread_id / stage)) + thread_id % stage;
}

unsigned int thread_root_map(unsigned int thread_id, unsigned int estage,
unsigned int istage)
{
	return istage *  (thread_id % estage);
}

float round(float x){
	float y = ceilf(x*100)/100;
	return y;
}
```

```
unsigned int error_calculator(std::vector<cl_float2> fromCPU,
std::vector<cl_float2> fromGPU)
{
	unsigned int errorCount = 0;
	for(size_t i = 0; i < fromCPU.size(); i++){
		float xdif = abs(fromCPU[i].x - fromGPU[i].x);
		float ydif = abs(fromCPU[i].y - fromGPU[i].y);
		if(xdif > 0.0001 || ydif > 0.0001)
			errorCount++;
	}
	return errorCount;
}

std::vector<cl_float2>  opencl(std::vector<cl_float2> input,
Core::TimeSpan *gpuTime)
{
	std::vector<cl_float2> output;
	// Create a context
	//cl::Context context(CL_DEVICE_TYPE_GPU)
	std::vector<cl::Platform> platforms;
	cl::Platform::get(&platforms);
	if (platforms.size() == 0) {
		std::cerr << "No platforms found" << std::endl;
		return output;
	}
	int platformId = 0;
	for (size_t i = 0; i < platforms.size(); i++) {

		//cout << platforms[i].getInfo<CL_PLATFORM_NAME>() << endl;
		platformId = i;
	}
	cl_context_properties prop[4] = { CL_CONTEXT_PLATFORM,
(cl_context_properties)platforms[platformId](), 0, 0 };
	//std::cout << "Using platform '" <<
platforms[platformId].getInfo<CL_PLATFORM_NAME>() << "' from '" <<
platforms[platformId].getInfo<CL_PLATFORM_VENDOR>() << "'" << std::endl;
	cl::Context context(CL_DEVICE_TYPE_GPU, prop);

	// Get a device of the context
	//std::cout << "Context has " <<
context.getInfo<CL_CONTEXT_DEVICES>().size() << " devices" << std::endl;
	cl::Device device = context.getInfo<CL_CONTEXT_DEVICES>()[0];
	std::vector<cl::Device> devices;
	devices.push_back(device);

	// Create a command queue
	cl::CommandQueue queue(context, device, CL_QUEUE_PROFILING_ENABLE);

	// Load the source code
	cl::Program program = OpenCL::loadProgramSource(context,
"src/FFT.cl");
	// Compile the source code. This is similar to
program.build(devices) but will print more detailed error messages
	OpenCL::buildProgram(program, devices);

	std::size_t data_size = input.size()*sizeof(cl_float2);
	cl::Buffer d_data(context, CL_MEM_READ_WRITE, data_size);
	cl::Buffer d_roots(context, CL_MEM_READ_WRITE, (input.size()/2) *
sizeof(cl_float2));
```

```
	cl::Event copytoGPU;
	queue.enqueueWriteBuffer(d_data, true, 0 , data_size, input.data(),
NULL, &copytoGPU);

	cl::Event event;

	// Create a kernel object
	cl::Kernel kernel1(program, "fftKernel");
	kernel1.setArg(0, d_data);
	kernel1.setArg(1, d_roots);

	unsigned int WG_size = input.size()/2;
	if(WG_size > 256)
		WG_size = 256;

	queue.enqueueNDRangeKernel(kernel1, 0, input.size()/2, WG_size, 0,
&event);

	cl::Event copytoCPU;
	queue.enqueueReadBuffer(d_data, true, 0, data_size, input.data(),
NULL, &copytoCPU);
	queue.finish();

	Core::TimeSpan gpuExecTime = OpenCL::getElapsedTime(event);
	Core::TimeSpan copyTime = OpenCL::getElapsedTime(copytoGPU) +
OpenCL::getElapsedTime(copytoCPU);
	*gpuTime = gpuExecTime + copyTime;

	//cout << "Total time in GPU is " << gpuTime << endl;
	return input;

}

void speed_test(unsigned int size)
{
	ofstream myfile;
	myfile.open ("output.txt");

	unsigned vector_size = 1;
	for(unsigned int i = 0; i <= size; i ++)
	{
		vector_size = vector_size * 2;
		std::vector<cl_float2> test;
		for(unsigned int j = 0; j < vector_size; j++)
		{
			cl_float2 sample;
			sample.y = 0;
			sample.x = (float)(j+1);
			test.push_back(sample);
		}

		//CPU
		Core::TimeSpan cpuStart = Core::getCurrentTime();
		vector<cl_float2> result = fft_cpu(test);
		Core::TimeSpan cpuEnd = Core::getCurrentTime();
		Core::TimeSpan cpuTime = cpuEnd - cpuStart;
		cout << "CPU, " << vector_size << ", " << cpuTime << endl;
```

```
            //GPU
            Core::TimeSpan gpuTime = Core::getCurrentTime();
            vector<cl_float2> resultgpu = opencl(test, &gpuTime);
            cout << "GPU, " << vector_size << ", " << gpuTime << endl;

            myfile << "CPU, " << vector_size << ", " << cpuTime << ",
GPU, " << vector_size << ", " << gpuTime << endl;
      }
      myfile.close();
}

int main()
{
      std::vector<cl_float2> h_input;

      if (read_input(&h_input, "input.txt"))
      {
            cout << "Input has " << h_input.size() << " samples." <<
endl;
      }
      else
      {
            cout << "Input invalid" << endl;
            return 1;
      }

      cout << "raw input";
      print_vector(&h_input);
      cout << endl;

      h_input = reorder_input(h_input);

#ifdef DEBUG
      cout << "reordered input";
      print_vector(&h_input);
      cout << endl;
#endif

      Core::TimeSpan cpuStart = Core::getCurrentTime();
      vector<cl_float2> result = fft_cpu(h_input);
      Core::TimeSpan cpuEnd = Core::getCurrentTime();

      Core::TimeSpan cpuTime = cpuEnd - cpuStart;

      cout << "fft result";
      print_vector(&result);
      cout << endl;

      Core::TimeSpan gpuTime = Core::getCurrentTime();
      vector<cl_float2> resultgpu = opencl(h_input, &gpuTime);

      cout << "opencl result";
      print_vector(&resultgpu);
      cout << endl;

      cout << "Found errors in " << error_calculator(result,resultgpu) <<
" values" << endl;

      cout << "Total time in CPU is " << cpuTime << endl;
```

```
        cout << "Total time in GPU is " << gpuTime << endl;

        speed_test(20);
}
```

FFT.cl

```
#ifndef __OPENCL_VERSION__
#include <OpenCL/OpenCLKernel.hpp> // Hack to make syntax highlighting in Eclipse
work
#endif

#define pi 3.14159265359f

unsigned int thread_index_map(unsigned int thread_id, unsigned int stage)
{
	return (stage * 2 * (thread_id / stage)) + thread_id % stage;
}

unsigned int thread_root_map(unsigned int thread_id, unsigned int estage,
unsigned int istage)
{
	return istage *  (thread_id % estage);
}

float2 doublec_add(float2 a, float2 b)
{
	float2 c;
	c.x = a.x + b.x;
	c.y = a.y + b.y;
	return c;
}

float2 doublec_sub(float2 a, float2 b)
{
	float2 c;
	c.x = a.x - b.x;
	c.y = a.y - b.y;
	return c;
}

float2 doublec_mul(float2 a, float2 b)
{
	float2 c;
	c.x = a.x*b.x - a.y*b.y;
	c.y = a.y*b.x + a.x*b.y;
	return c;
}

__kernel void fftKernel(__global float2* d_input, __global float2* d_roots) {

	size_t i = get_global_id(0);

	size_t count = get_global_size(0);

	//populate root array
	float arg = (2 * pi * (float)i) / ((float)count * 2);
	d_roots[i].x = cos(arg);
	d_roots[i].y = -1 * sin(arg);

	barrier(CLK_GLOBAL_MEM_FENCE);

	unsigned int istage = count;
	for (unsigned int estage = 1; estage <= count; estage = estage * 2)
	{
		//we query for our home and target indeces
		//as well as our home and target root indeces
		unsigned int home_index = thread_index_map(i, estage);
		unsigned int target_index = home_index + estage;
		unsigned int home_root = thread_root_map(i, estage, istage);
```

```
                float2 pq = doublec_mul(d_input[target_index], d_roots[home_root]);
                float2 top = doublec_add(pq, d_input[home_index]);
                float2 bottom = doublec_sub(d_input[home_index], pq);
                d_input[home_index] = top;
                d_input[target_index] = bottom;

                istage = istage / 2;

                barrier(CLK_GLOBAL_MEM_FENCE);
        }
}
```

doublec.hpp

```
#ifndef __OPENCL_VERSION__
#include <OpenCL/OpenCLKernel.hpp> // Hack to make syntax highlighting in Eclipse
work
#endif

#define pi 3.14159265359f

unsigned int thread_index_map(unsigned int thread_id, unsigned int stage)
{
	return (stage * 2 * (thread_id / stage)) + thread_id % stage;
}

unsigned int thread_root_map(unsigned int thread_id, unsigned int estage,
unsigned int istage)
{
	return istage * (thread_id % estage);
}

float2 doublec_add(float2 a, float2 b)
{
	float2 c;
	c.x = a.x + b.x;
	c.y = a.y + b.y;
	return c;
}

float2 doublec_sub(float2 a, float2 b)
{
	float2 c;
	c.x = a.x - b.x;
	c.y = a.y - b.y;
	return c;
}

float2 doublec_mul(float2 a, float2 b)
{
	float2 c;
	c.x = a.x*b.x - a.y*b.y;
	c.y = a.y*b.x + a.x*b.y;
	return c;
}

__kernel void fftKernel(__global float2* d_input, __global float2* d_roots) {

	size_t i = get_global_id(0);

	size_t count = get_global_size(0);

	//populate root array
	float arg = (2 * pi * (float)i) / ((float)count * 2);
	d_roots[i].x = cos(arg);
	d_roots[i].y = -1 * sin(arg);

	barrier(CLK_GLOBAL_MEM_FENCE);

	unsigned int istage = count;
	for (unsigned int estage = 1; estage <= count; estage = estage * 2)
	{
		//we query for our home and target indeces
		//as well as our home and target root indeces
		unsigned int home_index = thread_index_map(i, estage);
		unsigned int target_index = home_index + estage;
		unsigned int home_root = thread_root_map(i, estage, istage);
```

```
                float2 pq = doublec_mul(d_input[target_index], d_roots[home_root]);
                float2 top = doublec_add(pq, d_input[home_index]);
                float2 bottom = doublec_sub(d_input[home_index], pq);
                d_input[home_index] = top;
                d_input[target_index] = bottom;

                istage = istage / 2;

                barrier(CLK_GLOBAL_MEM_FENCE);
        }
}
```