GPU Accelerated Fast Fourier Transform

Matt McCarthy

Christopher Newport University
CPSC 621 Parallel Processing
matthew.mccarthy.12@cnu.edu

December 2015

Abstract We empirically investigate the performance benefits of parallel fast Fourier transform running on the GPU over a sequential version running on the GPU.

1 Background

1.1 Discrete Fourier Transform

The discrete Fourier transform is a mathematical transformation that takes a set of Complex-valued signals and outputs a set of Complex-valued frequencies. For an n-dimensional Complex-valued vector $\mathbf{X}$, the discrete Fourier transform $\mathbf{Y} = \mathcal{F}(X)$ is given by

$$Y_j = \sum_{k=0}^{n} x_k \omega^{jk}$$

where ω is the n-th root of unity, $e^{2\pi i/n}$. Since $\mathbf{Y}$ is an n-dimensional, Complex-valued vector, we can see that the discrete Fourier transform has a complexity of $\Theta(n^2)$.

1.2 Fast Fourier Transform

Furthermore, we can split the discrete Fourier transform into even and odd sums for $n = 2m$, yielding

$$Y_j = \sum_{k=0}^{m} x_{2k} \omega^{2jk} + \omega^j \sum_{k=0}^{m} x_{2k+1} \omega^{2jk}$$

which is two separate discrete Fourier transforms. Suppose $n = 2^k$. If we iterate this process, we get the following algorithm called the one-dimensional, unordered radix 2, fast Fourier transform in Algorithm 1.1.

Algorithm 1.1. Recursive FFT

```
1:  function R-FFT(X,Y,n,ω)
2:      if n=1 then
3:          y_0 = x_0
4:      else
5:          Let Q = 0, T = 0 ∈ C^n
6:          Let X_e = (x_0, x_2, ..., x_{n-2})
7:          Let X_o = (x_1, x_3, ..., x_{n-1})
8:          R-FFT(X_e,Q_e,n/2,ω^2)
9:          R-FFT(X_o,T_o,n/2,ω^2)
10:         for all j ∈ {0, 1, ..., n − 1} do
11:             y_j = q_{j mod n/2} + ω^i t_{j mod n/2}
12:         end for
13:     end if
14: end function
```

1.2.1 Cooley Tukey

Furthermore, we have an iterative formulation of the prior algorithm, called the Cooley Tukey algorithm for one-dimensional, unordered radix 2, fast Fourier transforms.

From this pseudo-code, we can determine the complexity of fast Fourier transform as described by Algorithm 1.2. We begin by noting that we iterate through the outer loop precisely $\lg n$ times and the inner loop n times. Therefore our complexity is $T_1 = \Theta(n \lg n)$.

1.3 Parallelization

For our parallelization, we use a simplified version of the binary exchange algorithm, a parallelization of the Cooley Tukey algorithm designed for use on a hypercube. Since our implementation runs on a single GPU, any thread can access any memory location via

Algorithm 1.2. Cooley-Tukey FFT

```
1: function I-FFT(X,Y,n)
2:     t := lg n
3:     R = X
4:     for m = 0 to t − 1 do
5:         S = R
6:         for l = 0 to n − 1 do
7:             Let (b_0 b_1 ... b_{t−1}) be the binary expansion of l
8:             j := (b_0 ... b_{m−1} 0 b_{m+1} ... b_{t−1})
9:             k := (b_0 ... b_{m−1} 1 b_{m+1} ... b_{t−1})
10:            r_i := s_j + s_k ω^{(b_m b_{m−1} ... b_0 0 ... 0)}
11:        end for
12:    end for
13:    Y := R
14: end function
```

a pointer. However, this also complicates the matter by introducing a potential for data races. We solve this by modifying the algorithm to work as follows in Algorithm 1.3. In each iteration, we only write to **R**

Algorithm 1.3. Parallel FFT

```
1: function PAR-FFT(X,Y,n)
2:     t := lg n, BLK := n/p
3:     R = X
4:     S = 0
5:     for m = 0 to t − 1 do
6:         Swap pointers R and S
7:         spawn process for l = 0 to BLK − 1 do
8:             for c = l · BLK, to l · (BLK + 1) do
9:                 Let (b_0 b_1 ... b_{t−1}) be the binary expansion of c
10:                j := (b_0 ... b_{m−1} 0 b_{m+1} ... b_{t−1})
11:                k := (b_0 ... b_{m−1} 1 b_{m+1} ... b_{t−1})
12:                r_i := s_j + s_k ω^{(b_m b_{m−1} ... b_0 0 ... 0)}
13:            end for
14:        end spawn
15:        sync
16:    end for
17:    Y := R
18: end function
```

and only read from **S**. Since we wait until each thread is complete before moving on to the next iteration, we avoid the potential to use old or incorrect data.

If we inspect the parallelization, we see that the outer loop runs $\lg n$ times. However, the inner loop is ran on p processes, each of which handle n/p iterations. Therefore, the computation cost is $\Theta(n/p \lg n)$. Moreover, for communication we simply do two reads from VRAM on each iteration of the inner loop, however some values will be in cache so we may get better performance than that. Ergo, the communication cost is $O(n/p \lg n)$. Thus, the parallel runtime will be $T_p = \Theta(n/p \lg n)$ and is cost optimal if and only if $p \leq n$.

2 Experimental Design

The goal of the experiment is to empirically determine the effect of parallelization on the runtime of the Cooley Tukey algorithm. To this end we want to measure the runtime of fast Fourier transform using differing sizes of n, the dimension of our Complex valued vector, and t, the number of threads. Let $\mathcal{N}$ be the set of vector dimensions we will test and let $\mathcal{T}$ be the set of thread counts we will test. Furthermore, we specify that each $n \in \mathcal{N}$ and each $t \in \mathcal{T}$ be powers of two.

For $t = 1$, we will run the Cooley Tukey algorithm on random Complex valued vectors of dimension n for each $n \in \mathcal{N}$. Otherwise, we run our parallelization of the Cooley Tukey algorithm on Complex valued vectors of dimension n for each $n \in \mathcal{N}$. Moreover, before and after each run, we copy the input to the GPU and the output from the GPU. We take the system time before copying the input to the GPU and again after copying the output to the host. We do not check the output for correctness during the test in order to save time, but instead checked the correctness of the algorithm beforehand.

3 Test Environment

3.1 Test System

The machine used to run the test has an Intel i7 4770k running at 4.2GHz, 16GB of RAM, and a Nvidia GTX 970 with a core clock of 1342MHz and memory clock of 7000MHz. The computer was running Arch Linux on Linux kernel 4.2.5 with Nvidia driver version 358.16 and CUDA 7.5.

3.2 Test Program

Firstly, all code can be located at `https://github.com/matt-mccarthy/fast-fourier-transform` in the `fast-fourier` folder. We also include a version of the code in the Appendix.

We wrote all code in CUDA C++. Furthermore all code was compiled with the CUDA toolkit 7.5 and GCC 5.2.0 using the command `nvcc -std=c++11 -rdc=true -arch=compute_52 -code=sm_52 -O3`. The library source file is `src/fast-fourier.cu`, and the sequential and parallel main functions are located in `sequential-fft.cu` and `parallel-fft.cu` respectively.

In both the sequential fast Fourier and parallel fast Fourier, we ran into issues with dynamically allocating memory in a CUDA kernel. Namely, if we allocated and freed often enough, we ran out of memory. To avoid this issue entirely, we made all allocations of giant arrays occur in main once and then reused those arrays. As a note, the `binary_stor` array is where we store the binary representation of our index l for each thread. While this design is suboptimal (ideally, each thread would create its own so others can not touch it), it works and the performance impact should be neglible for large enough n.

Moreover, our GPU code is entirely without any conditional statements except the ones for determining whether or not our for loop is within its bounds. Another note about the code is that at the end of each iteration of the outer for loop, we call `cudaDeviceSynchronize` which blocks until our `transformer` kernel completes. We do this in order to ensure syncrhonization across all threads and prevent data races. Furthermore, the `transformer` kernel must always be called from another kernel that runs on exactly one thread. We do this in order to keep all code running on the device and prevent useless communication between the CPU and GPU.

For our test we took, $\mathcal{N} = \{2^{17}, 2^{19}, 2^{21}, 2^{23}, 2^{25}\}$ and $\mathcal{T} = \{1, 4, 16, 64, 256, 1024\}$, all of which are less than the number of cores on our GTX 970. We chose $\mathcal{N}$ such that we would get a runtime for 1MB of data on the low end, and 1GB of data on the high end since each complex value is a typedef'd `float2`. Moreover, we ran each combination of n and t 10 times each. All runtimes were measured and recorded in milliseconds, and our result files are included in Appendix B.

4 Results

Figure 2 lists the runtimes for each t (row index) at size n (column index). Moreover, the graphs all look like the standard $x \ln x$ curve. By inspection, we see that $T_{2k} \approx T_{2k+2}/4$, which is exactly what we are looking for. So naively, we can say that this has a linear speedup.

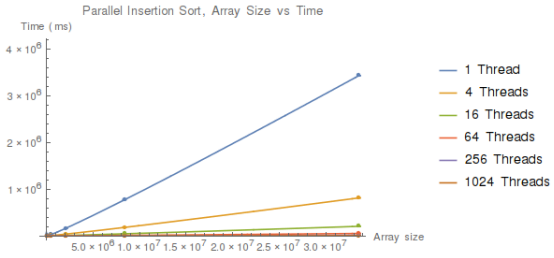

Figure 1: Runtime vs Array Size

	2^{17}	2^{19}	2^{21}	2^{23}	2^{25}
2^0	7.37	35.8	162	773	$3.42 \cdot 10^6$
2^2	1.75	8.47	38.5	183	813
2^4	.446	2.15	9.75	46.4	205.8
2^6	.117	.552	2.51	11.9	52.8
2^8	.034	.149	.669	3.14	14.3
2^{10}	0.011	.046	.201	.975	4.65

Figure 2: The table of runtimes at (n, t) in s

We also attempted to compute on 1GB of random complex numbers, or precisely, 2^{27} complex numbers. The parallel runtime for $t = 1024$ was approximately 22s, however, after many hours of running the sequential version at this n did not complete. Upon deeper analysis, we learned that with perfect linear speedup, the sequential version would have a runtime roughly 1000 times greater than T_{1024}, namely 6 hours. Due to time constraints, we had to abandon this size of n and make our maximum 2^{25}. Furthermore, due to memory constraints, we could not test for any $n \geq 2^{28}$ since we have to allocate two arrays of 2GB each in order to compute on it and our 970 has 4GB VRAM.

5 Future Work

In our study, VRAM limitations and time constraints prevented us from harnessing extremely large values of n. One way to extend this study to larger values of n is to simply use GPUs that have more than 4GB of VRAM like the Titan X and simply run tests with larger values of n.

Another, more interesting path to explore is to distribute the problem across multiple GPUs. If we split it smartly, we should be able to utilize the extra VRAM without incurring too much communication overhead. Of course, the inter-GPU communication is also a factor and we would need to decide the best

way to manage it. This latter method can also scale to a GPU farm.

6 Conclusion

Our analysis of the algorithm determined that for $t \leq n$, we should get perfect linear speedup and the data provides empirical evidence for that, with 1024 threads executing approximately 1000 times faster than the sequential.

References

[1] Ananth Grama et al. *Introduction to Parallel Computing*. 2nd ed. ISBN: 9780201648652.

Appendix A: Code

run.sh

```
out_dir="results"
trials=10
# Must be powers of two.
# I go from 1M (131072 float2's) to 512M (33554432 float2's)
arr_size=(131072 524288 2097152 8388608 33554432)
# Must also be powers of two. I pass this twice since I use num_blks=num_thds.
num_thds=(2 4 8 16 32)
# The names of the executables
seq_file="./sequential-fft"
par_file="./parallel-fft"
# Clean the result directory
rm -rf $out_dir
mkdir $out_dir
# Run sequential
for size in "${arr_size[@]}"
do
    echo 1 $size
    out=$($seq_file $size $trials)
    echo -e $size'\t'$out >> $out_dir/1.txt
done
# Run parallel
for threads in "${num_thds[@]}"
do
    # Total number of threads = num_blk * num_thd
    exprans=`expr $threads \\* $threads`
    for size in "${arr_size[@]}"
    do
        echo $exprans $size
        out=$($par_file $size $trials $threads $threads)
        echo -e $size'\t'$out >> $out_dir/$exprans.txt
    done
done
```

sequential-fft.cu

```
// This program takes n and trial count as parameters and nothing more.
// It is assumed that n is a power of 2.
// Compiles with nvcc -std=c++11 -rdc=true -arch=compute_50 -code=sm_50
#include <chrono>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <utility>

#include <math_functions.h>

#include "include/fast-fourier.h"

using namespace std;
using namespace chrono;
using namespace fast_fourier;

void    gen_array(cfloat* output, int n);
long double    average(long double* in, int n);
long double    std_dev(long double* in, int n, long double average);

__global__
void run_test(cfloat* input, cfloat* output, int n, bool* binary_stor)
{
    fast_fourier_transform(input, output, n, binary_stor);
}

int main(int argc, char** argv)
{
    if (argc < 3)
    {
        cerr << "Usage is " << argv[0] << " n num_trials" << endl;
        return 1;
    }

    int        n(atoi(argv[1]));
    int        trial_count(atoi(argv[2]));

    cfloat*    input(new cfloat[n]);
    cfloat* output(new cfloat[n]);
    cfloat* d_input(nullptr);
    cfloat* d_output(nullptr);
    bool*    binary_stor(nullptr);

    long double    times[trial_count];
    high_resolution_clock::time_point tp2, tp1;
    duration<long double, ratio<1,1000> > time_span;

    // Allocate device arrays
    if (cudaMalloc( &d_input, sizeof(cfloat) * n ) != cudaSuccess)
    {
        auto t = cudaGetLastError();
        cout << "Failed to allocate input: "
            << cudaGetErrorName(t) << ", "
            << cudaGetErrorString(t) << endl;
        return 1;
    }
    if (cudaMalloc( &d_output, sizeof(cfloat) * n ) != cudaSuccess)
    {
        auto t = cudaGetLastError();
        cout << "Failed to allocate output: "
            << cudaGetErrorName(t) << ", "
            << cudaGetErrorString(t) << endl;
        return 1;
    }
    if (cudaMalloc( &binary_stor, sizeof(bool) * ilogbf(n) ) != cudaSuccess)
    {
        auto t = cudaGetLastError();
        cout << "Failed to allocate boolean storage: "
            << cudaGetErrorName(t) << ", "
            << cudaGetErrorString(t) << endl;
        return 1;
    }

    // Run experiment
```

```
    for (int j(0) ; j < trial_count ; j++)
    {
        // Generate random input
        gen_array(input, n);

        // Run the test
        tp1 = system_clock::now();
        // Copy the input array to the GPU
        if (cudaMemcpy( d_input, input, (long) n * sizeof(cfloat), cudaMemcpyHostToDevice ) != cudaSuccess)
        {
            auto t = cudaGetLastError();
            cout << "Iteration: " << j
                << " Input failed to copy: "
                << cudaGetErrorName(t) << ", "
                << cudaGetErrorString(t) << endl;
            return 1;
        }
        run_test<<<1,1>>>(d_input, d_output, n, binary_stor);
        if (cudaMemcpy( output, d_output, (long) n * sizeof(cfloat), cudaMemcpyDeviceToHost ) != cudaSuccess)
        {
            auto t = cudaGetLastError();
            cout << "Iteration: " << j
                << " Output failed to copy: "
                << cudaGetErrorName(t) << ", "
                << cudaGetErrorString(t) << endl;
            return 1;
        }
        tp2 = system_clock::now();

        time_span    = duration_cast< duration<long double, ratio<1,1000> > >(tp2 - tp1);
        times[j]    = time_span.count();
    }

    // Calculate statistics
    long double av(average(times, trial_count));
    long double sd(std_dev(times, trial_count, av));

    cout << av << "\t" << sd << endl;

    cudaFree( binary_stor );
    cudaFree( d_input );
    cudaFree( d_output );
    return 0;
}

void gen_array(cfloat* output, int n)
{
    srand(time(nullptr));

    for (int j = 0; j < n; j++)
        output[j] = cfloat(rand(), rand());
}

long double    average(long double* in, int n)
{
    long double s(0.0);

    for (int j(0) ; j < n ; j++)
        s += in[j];

    return s/n;
}

long double    std_dev(long double* in, int n, long double average)
{
    long double var = 0;
    long double tmp = 0;

    for (int i = 0 ; i < n ; i++)
    {
        tmp = (in[i] - average);
        var += tmp * tmp;
    }

    long double stdDev = sqrt(var/n);

    return stdDev;
```

```
}
```

parallel-fft.cu

```
// This program takes n, block count, thread count, and number of trials as parameters.
// It is assumed that these are all powers of 2.
// Compiles with nvcc -std=c++11 -rdc=true -arch=compute_50 -code=sm_50
#include <chrono>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <utility>

#include "include/fast-fourier.h"

using namespace std;
using namespace chrono;
using namespace fast_fourier;

void    gen_array(cfloat* output, int n);
long double    average(long double* in, int n);
long double    std_dev(long double* in, int n, long double average);

int main(int argc, char** argv)
{
    if (argc < 3)
    {
        cerr << "Usage is " << argv[0] << " n num_trials num_blk num_thd" << endl;
        return 1;
    }

    int        n(atoi(argv[1]));
    int        trial_count(atoi(argv[2]));
    int        num_blk(atoi(argv[3]));
    int        num_thd(atoi(argv[4]));

    cfloat*    input(new cfloat[n]);
    cfloat* output(new cfloat[n]);
    cfloat* d_input(nullptr);
    cfloat* d_output(nullptr);
    bool*    binary_stor(nullptr);

    long double    times[trial_count];
    high_resolution_clock::time_point tp2, tp1;
    duration<long double, ratio<1,1000> > time_span;

    // Allocate device arrays
    if (cudaMalloc( &d_input, sizeof(cfloat) * n ) != cudaSuccess)
    {
        auto t = cudaGetLastError();
        cout << "Failed to allocate input: "
            << cudaGetErrorName(t) << ", "
            << cudaGetErrorString(t) << endl;
        return 1;
    }
    if (cudaMalloc( &d_output, sizeof(cfloat) * n ) != cudaSuccess)
    {
        auto t = cudaGetLastError();
        cout << "Failed to allocate output: "
            << cudaGetErrorName(t) << ", "
            << cudaGetErrorString(t) << endl;
        return 1;
    }
    if (cudaMalloc( &binary_stor, sizeof(bool) * ilogbf(n) * num_thd * num_blk) != cudaSuccess)
    {
        auto t = cudaGetLastError();
        cout << "Failed to allocate boolean storage: "
            << cudaGetErrorName(t) << ", "
            << cudaGetErrorString(t) << endl;
        return 1;
    }

    // Run experiment
    for (int j(0) ; j < trial_count ; j++)
    {
        // Generate random input
        gen_array(input, n);

        // Run the test
```

```
        tp1 = system_clock::now();
        // Copy the input array to the GPU
        if (cudaMemcpy( d_input, input, (long) n * sizeof(cfloat), cudaMemcpyHostToDevice ) != cudaSuccess)
        {
            auto t = cudaGetLastError();
            cout << "Iteration: " << j
                << " Input failed to copy: "
                << cudaGetErrorName(t) << ", "
                << cudaGetErrorString(t) << endl;
            return 1;
        }
        fast_fourier_transform<<<1,1>>>(d_input, d_output, n, num_blk, num_thd, binary_stor);
        if (cudaMemcpy( output, d_output, (long) n * sizeof(cfloat), cudaMemcpyDeviceToHost ) != cudaSuccess)
        {
            auto t = cudaGetLastError();
            cout << "Iteration: " << j
                << " Output failed to copy: "
                << cudaGetErrorName(t) << ", "
                << cudaGetErrorString(t) << endl;
            return 1;
        }
        tp2 = system_clock::now();

        time_span    = duration_cast< duration<long double, ratio<1,1000> > >(tp2 - tp1);
        times[j]    = time_span.count();
    }

    // Calculate statistics
    long double av(average(times, trial_count));
    long double sd(std_dev(times, trial_count, av));

    cout << av << "\t" << sd << endl;

    cudaFree( binary_stor );
    cudaFree( d_input );
    cudaFree( d_output );
    return 0;
}

void gen_array(cfloat* output, int n)
{
    srand(time(nullptr));

    for (int j = 0; j < n; j++)
         output[j] = cfloat(rand(), rand());
}

long double    average(long double* in, int n)
{
    long double s(0.0);

    for (int j(0) ; j < n ; j++)
        s += in[j];

    return s/n;
}

long double    std_dev(long double* in, int n, long double average)
{
    long double var = 0;
    long double tmp = 0;

    for (int i = 0 ; i < n ; i++)
    {
        tmp = (in[i] - average);
        var += tmp * tmp;
    }

    long double stdDev = sqrt(var/n);

    return stdDev;
}
```

include/fast-fourier.h

```
#ifndef FAST_FOURIER_H
#define FAST_FOURIER_H

#include <thrust/complex.h>

namespace fast_fourier
{
    typedef thrust::complex<float> cfloat;

    /// Does fast fourier transform.
    /// @param x The input vector (MUTABLE!!!).
    /// @param y The output vector.
    /// @param n The size of the vectors.
    /// @param binary_stor A place to store binary representations.
    __host__ __device__
    void fast_fourier_transform(cfloat* x, cfloat* y, unsigned n, bool* binary_stor);
    /// Does parallel fast fourier transform.
    /// @param x The input vector (MUTABLE!!!).
    /// @param y The output vector.
    /// @param n The size of the vectors.
    /// @param blk_count The number of blocks you wish to spawn.
    /// @param thd_count The number of threads per block.
    /// @param binary_stor A place to store binary representations.
    __global__
    void fast_fourier_transform(cfloat* x, cfloat* y, unsigned n, int blk_count, int thd_count, bool* binary_stor);
    /// Does discrete fourier transform.
    /// @param x The input vector.
    /// @param n The size of the vector.
    /// @return A pointer to the output vector.
    __host__ __device__
    cfloat* discrete_fourier_transform(cfloat* x, unsigned n);
}

#endif
```

include/src/fast-fourier.cu

```
// Compiles with nvcc -std=c++11 -rdc=true -arch=compute_50 -code=sm_50
#include <thrust/fill.h>
#include <thrust/copy.h>
#include <math_constants.h>
#include <math_functions.h>

#include "../fast-fourier.h"

using thrust::copy_n;
using thrust::exp;
using thrust::fill_n;

using fast_fourier::cfloat;

/// Increments a number represented in binary.
/// @param i Our binary representation.
/// @param lg_n The size of the bool vector (it had better be lg n!).
__host__ __device__
void    binary_inc(bool* i, int lg_n);
/// Converts a binary representaion to an integer.
/// @param i Our binary representation.
/// @param lg_n The size of the bool vector (it had better be lg n!).
/// @return The decimal value
__host__ __device__
int        bin2dec(const bool* i, int lg_n);
/// Writes a binary representation to a bit vector.
/// @param i Our binary representation.
/// @param l The int to convert.
/// @param lg_n The size of the bool vector (it had better be lg n!).
__host__ __device__
void    dec2bin(bool* i, int l, int lg_n);
/// Our wierd binary exponent.
/// @param l Our binary representation.
/// @param lg_n The size of the bool vector (it had better be lg n!).
/// @param m The current iteration.
__host__ __device__
int        wierd_bin_thingy(const bool* l, int lg_n, int m);
/// Returns the nth root of unity raised to the k
__host__ __device__
cfloat    k_root_unity(int k, int n);

/// The inner loop of FFT
/// @param r The vector to which we write.
/// @param s The vector from which we read.
/// @param lg_n lg n.
/// @param blk_off The block offset (used for reading from binary_stor).
/// @param thd_off The thread offset (used for reading from binary_stor).
/// @param n The size of our I/O vectors.
/// @param m The current iteration.
/// @param binary_stor our bit vector where we store binary representations.
/// @param thd_count The number of threads per block.
__global__
void transformer(cfloat* r, cfloat* s, unsigned lg_n, unsigned blk_off,
                 unsigned thd_off, unsigned n, int m, bool* binary_stor, int thd_count);
/// A parallel copy algorithm.
/// @param src The source.
/// @param dst The destination.
/// @param n The size of the vectors.
/// @param blk_off The number of things each block gets.
/// @param thd_off The number of things each thread gets.
__global__
void parallel_copy(const cfloat* src, cfloat* dst, unsigned n, unsigned blk_off,
                   unsigned thd_off);

cfloat* fast_fourier::discrete_fourier_transform(cfloat* x,    unsigned n)
{
    cfloat* y(new cfloat[n]);

    fill_n(y, n, cfloat(0.0f));

    for (int j(0) ; j < n ; j++)
        for (int k(0) ; k < n ; k++)
            y[j] += x[k] * k_root_unity(k*j, n);

    return y;
```

```
}

void fast_fourier::fast_fourier_transform(cfloat* x, cfloat* y, unsigned n,
                                          bool* binary_stor)
{
    cfloat*    s(x);
    cfloat*    r(y);
    cfloat*    tmp_ptr;
    int        lg_n(ilogbf(n));
    int        j,k,u_exp;
    bool*    l_bi(binary_stor);
    bool    tmp(false);

    for (int j(0) ; j < lg_n ; j++ )
        l_bi[j] = false;

    for (int m(0) ; m < lg_n ; m++)
    {
        tmp_ptr = s;
        s = r;
        r = tmp_ptr;

        for (int l(0) ; l < n ; l++)
        {
            tmp = l_bi[m];
            l_bi[m] = false;

            j = bin2dec(l_bi, lg_n);
            k = j + (int)exp2f(lg_n - m - 1);

            l_bi[m] = tmp;

            u_exp = wierd_bin_thingy(l_bi, lg_n, m);
            r[l] = s[j] + s[k] * k_root_unity(u_exp, n);

            binary_inc(l_bi, lg_n);
        }
    }

    for (int j(0) ; j < n ; j++)
        y[j] = r[j];
}

__global__
void fast_fourier::fast_fourier_transform(cfloat* x, cfloat* y, unsigned n,
                                          int blk_count, int thd_count, bool* binary_stor)
{
    int        lg_n(ilogbf(n));

    int        blk_off = n / blk_count;
    int        thd_off    = blk_off / thd_count;

    cfloat *r(x), *s(y);
    cfloat*    tmp_ptr;

    for (int m(0) ; m < lg_n ; m++)
    {
        // Swap s and r so the last output becomes the new input
        tmp_ptr = r;
        r        = s;
        s        = tmp_ptr;

        // Perform the next step of the transform
        transformer<<<blk_count, thd_count>>>(r, s, lg_n, blk_off, thd_off, n, m, binary_stor, thd_count);
        cudaDeviceSynchronize();
    }

    // Copy r into y
    parallel_copy<<<blk_count, thd_count>>>(r, y, n, blk_off, thd_off);
    cudaDeviceSynchronize();

    // delete[] r, s;
}

__global__
void transformer(cfloat* r, cfloat* s, unsigned lg_n, unsigned blk_off,
                 unsigned thd_off, unsigned n, int m, bool* binary_stor, int thd_count)
```

```
{
    int        l_min( blockIdx.x * blk_off + threadIdx.x * thd_off );
    int        l_max( l_min + thd_off );
    int        j, k, u_exp;

    // bool*    l_bi(new bool[lg_n]);
    bool*    l_bi(binary_stor + (blockIdx.x * thd_count + threadIdx.x) * lg_n);
    bool    tmp;

    dec2bin(l_bi, l_min, lg_n);

    for (int l(l_min) ; l < l_max ; l++)
    {
        tmp = l_bi[m];

        l_bi[m]    = false;
        j        = bin2dec(l_bi, lg_n);
        k        = j + (int)exp2f(lg_n - m - 1);
        l_bi[m]    = tmp;

        u_exp    = wierd_bin_thingy(l_bi, lg_n, m);
        r[l]    = s[j] + s[k] * k_root_unity(u_exp, n);

        binary_inc(l_bi, lg_n);
    }

    // delete[] l_bi;
}

__global__
void parallel_copy(const cfloat* src, cfloat* dst, unsigned n, unsigned blk_off,
                   unsigned thd_off)
{
    int        l_min( blockIdx.x * blk_off + threadIdx.x * thd_off );
    int        l_max( l_min + thd_off );

    for (int l(l_min) ; l < l_max ; l++)
        dst[l] = src[l];
}

void binary_inc(bool* i, int lg_n)
{
    bool flag(true);

    for (int j(lg_n - 1) ; j > -1 && flag ; j--)
    {
        flag = i[j];
        i[j] = !flag;
    }
}

int bin2dec(const bool* i, int lg_n)
{
    int m(0), two_j(1);

    for (int j(lg_n - 1) ; j > -1 ; j--)
    {
        m += (int)i[j] * two_j;
        two_j *= 2;
    }

    return m;
}

int wierd_bin_thingy(const bool* l, int lg_n, int m)
{
    int exponent(0), two_j(1);

    for (int j(0) ; j < m + 1 ; j++)
    {
        exponent    += l[j] * two_j;
        two_j        *= 2;
    }

    for (int j(m + 1) ; j < lg_n ; j++)
        exponent    *= 2;
```

```
    return exponent;
}

cfloat k_root_unity(int k, int n)
{
    float    e_img(2.0f * ((float)k) * CUDART_PI_F /((float) n));
    cfloat    exponent(cfloat(0,e_img));

    return exp(exponent);
}

void dec2bin(bool* i, int l, int lg_n)
{
    for (int j(lg_n - 1) ; j > -1 ; j--)
    {
        i[j]    = (bool) (l % 2);
        l        /= 2;
    }
}
```

Appendix B: Results

1.txt

```
131072    7373.71 9.24972
524288    35790.6 6.01608
2097152    162529 20.0583
8388608    773815 44.6278
33554432    3.42766e+06 295.368
```

4.txt

```
131072    1747.24 9.69852
524288    8465.2 8.9587
2097152    38511.3 9.40846
8388608    183374 18.6204
33554432    813669 64.7037
```

16.txt

```
131072    445.596 9.95591
524288    2145.66 9.62443
2097152    9747.84 7.09251
8388608    46412 12.7099
33554432    205829 6.82086
```

64.txt

```
131072    117.242 5.8156
524288    552.491 9.52933
2097152    2508.03 7.97299
8388608    11884 8.44934
33554432    52810.5 7.01352
```

256.txt

```
131072    33.8304 1.67292
524288    148.929 6.68021
2097152    668.6 7.76817
8388608    3142.33 7.90265
33554432    14307.3 10.0786
```

1024.txt

```
131072    11.0459 0.0331117
524288    46.11 2.78446
2097152    201.333 6.6144
8388608    975.441 8.89214
33554432    4653.17 6.70649
```