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
```