Effects of Concurrency Techniques and Algorithm Performance

A Comparative Analysis of Single-Threaded, Multi-Threaded, and GPGPU Programming Techniques

---

Joshua Penton

Geocent, LLC

joshua.penton@geocent.com

May 2013

Table of Contents

Abstract

Deployment of parallel architectures in computing systems is increasing. In this paper we study the performance effects of a variety of programming techniques and technologies that utilize these parallel architectures as applied to example algorithms. We demonstrate that algorithms, which are highly parallel in nature, gain significant performance increases through proper application of both parallel computing methodologies and hardware. Using both Java and C++ environments with statistical and mathematical problem sets we demonstrate that proper utilization of parallel computing paradigms can reduce the execution time of algorithms by up to 99.997%.

Introduction

In order to meet the ever increasing computational demands of modern data-intensive applications hardware manufacturers have sought to provide increasingly advanced computational architectures. While classical designs for processing units focused on providing single core processors driven by high clock speeds modern architectures provide a more powerful and energy efficient solution through the application of multi-core processors that provide multiple lanes of computation and are driven by moderate clock speeds. Recent advances have also revealed the benefit of utilizing the highly parallel architecture of graphics processing units to provide applications with a basic processing pipeline capable of very high computational throughput.

Operating systems provide mechanisms such as threading architectures in order to access the multiple lanes of computation in a concurrent manner. Similarly, manufacturers have begun to provide APIs to access the parallel computational architecture of graphics processors.[1] However, while some compilers may provide basic optimizations on applications at compile time in order to take advantage of concurrent architectures the solutions usually far fall short from fully utilizing available resources[2]. In order to fully utilize the computational throughput of available resources software developers must design their applications with concurrency as a core element.

In this paper we show the evolution of several algorithms through the application of both language frameworks and concurrency techniques in order to highlight the inefficiencies of classical single pipeline development and to demonstrate the performance gains for applications by fully utilizing parallel architectures.

Concurrency Techniques

Single-Threaded

The most basic form of program execution is that in which a program is designed to follow a single execution path. Such programs assume a subscalar view of the processing unit in which only a single instruction that operates on a single memory element may be executed per processing cycle. Within an operational environment such a program would occupy a single thread of execution throughout the program's lifecycle.

Multi-Threaded

Advancements in programing methodologies led to the creation of multi-threading technologies in the 1970s. When supported by superscalar processing units and operational environments this technique allows for multiple independent threads of execution to be processed simultaneously. While each independent thread of execution typically performs a single instruction that operates on a single memory element per processing cycle, with the proper implementation of resource sharing constraints it is not required that the memory elements be independent of each other. When properly utilized by the developer or automatically implemented by compiler optimizations sections of the program which execute with multi-threading technology may decrease overall execution time nearly proportional to the number of threads supported by the operational environment[3].

General Purpose Graphics Processing Unit

Since the introduction of independent Graphics Processing Units (GPUs) developers have discovered novel methods to utilize the additional hardware to provide systems with additional general-purpose computational abilities. Such computational abilities are enabled through the availability of multiple independent programmable shaders on the GPU, which normally provide dedicated resources to perform vertex, pixel, and geometry calculations. Supporting architectures allow these shaders to be logically accessed as stream processing units and are exposed to the programmer through both device drivers and programming frameworks.

The hardware comprising a GPU stream processor is typically not as fully featured as that of a CPU core. However, the increased performance of the provided computational features coupled with the relatively large number of stream processing units located on modern graphics cards (2688 for the NVIDIA Tesla K20X[4] and 4096 for the Radeon HD 7990[5]) utilizing GPUs for applicable general-purpose tasks can provide substantial benefits to computational performance.

The general-purpose computing on graphics processing units (GPGPU) technique allows developers to provide a limited computational function (referred to as a kernel) to simultaneously execute on a defined number of available stream processors. The independent stream processors execute simultaneously and may access a stream of data to which each processing unit is provided access.

Numerical Approximation of π

The wide-ranging applications of the numerical constant π can be found throughout both pure mathematics and observable physics. While there exists highly efficient software[6] and methodologies[7] to calculate π to an arbitrary number of digits this paper will explore a rudimentary approximation method involving basic trigonometry and probability. The derived methodology will serve as an initial standardized benchmark and provide the basis for a comparative analysis of the applied computational frameworks, techniques, and architectures.

Algorithm Derivation

Consider a unit circle limited to the first Cartesian quadrant.

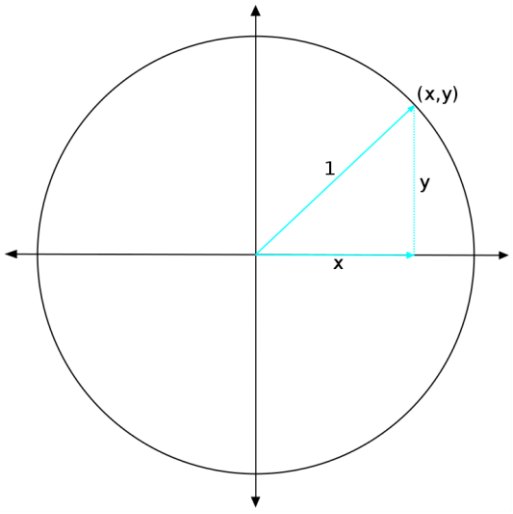


Fig. 1: A unit circle centered at the origin

The properties of this figure are such that the area within the quarter unit circle is defined as:

$$A_{circle} = \frac{\pi r^2}{4} = \frac{\pi}{4} \tag{1.1}$$

Therefore the probability of any random Cartesian coordinate between (0.0, 0.0) and (1.0, 1.0) falling within the quarter unit circle is equal to:

$$P_{circle} = \frac{A_{circle}}{A_{total}} = \frac{\pi/4}{1.0^2} = \frac{\pi}{4} \tag{1.2}$$

In order to experimentally calculate the specified probability P_{circle} we can generate a set of random Cartesian coordinates between (0.0, 0.0) and (1.0, 1.0) and determine the number of coordinates that fall within the area of the circle. This test can be calculated via the Pythagorean theorem where:

$$x_n^2 + y_n^2 \leq r^2 \tag{1.3}$$

Since the radius of the unit circle is 1.0 the test for the probability test can be simplified to:

(1.4) $$x_n^2 + y_n^2 \leq 1.0$$

As a result the experiment to calculate that value for P_{circle} becomes:

(1.5) $$P_{circle} = \frac{1}{N}\sum_{n=0}^{N-1}\begin{cases}1\ if\ x_n^2 + y_n^2 \leq 1.0\\0\ if\ x_n^2 + y_n^2 > 1.0\end{cases}$$

Finally, the experimental value for π can be calculated as:

(1.6) $$\pi = \frac{4}{N}\sum_{n=0}^{N-1}\begin{cases}1\ if\ x_n^2 + y_n^2 \leq 1.0\\0\ if\ x_n^2 + y_n^2 > 1.0\end{cases}$$

The Law of Large Numbers states that the number of trials of a random process increases, the percentage difference between the expected and actual values goes to zero. Therefore in our experiment the accuracy for the calculated value of π is directly proportional to the number of trials performed. In another words our aim should be to achieve the highest value of *N* within a limited experimental timeframe or otherwise decrease the time required to perform a fixed number of *N* trials.

Algorithm Pseudocode

From the derived equation the following pseudocode can be derived for further implementation:

```
SET NumberInside to 0
FOR i = 1 to NumberOfTrials
   Set RandomX to a random value between 0.0 and 1.0
   Set RandomY to a random value between 0.0 and 1.0
   IF RandomX * RandomX + RandomY * RandomY <= 1 THEN
      NumberInside = NumberInside + 1
   END IF
END FOR

Set Pi to 4.0 * NumberInside / NumberOfTrials
```

Java Implementations

Single-Threaded Implementation

The single-threaded Java implementation can be referenced in Appendix A and is an unmodified implementation of the derived algorithm pseudocode. The implementation performs a single loop for the total number of execution iterations. Within the loop new random values for the x and y coordinate are created and the coordinate it tested for its intersection within the quarter circle. Next the total number of valid points is directly accessed and the approximation of π is calculated.

Multi-Threaded Implementation

The multi-threaded Java implementation can be referenced in Appendix A and is a modified implementation of the derived algorithm code to allow for partitioning of the problem space and initiating concurrent execution in parallel threads.

The implementation moves the derived code to a private class `PiCalculator` which implements the `Callable<Long>` interface. Next the program creates a pool of

threads equal in size to the detected number of available processors. In this fashion once all threads become active the program should use all available processor cores to complete execution. Next the program creates number of `PiCalculator` objects equal in size to the detected number of available processors. The program initializes these objects with a total number of execution iterations to the total number of iterations specified divided by the total number of `PiCalculator` objects. Next the program links the executable objects to a thread in the allocated thread pool and initiates execution of the threads. Once the threads complete execution the results are fetched from the threads and the approximation of π is calculated.

C++ Implementations

Single-Threaded Implementation

The single-threaded C++ implementation can be referenced in Appendix A and is an unmodified implementation of the derived algorithm pseudocode. The implementation performs a single loop for the total number of execution iterations. Within the loop new random values for the x and y coordinate are created and the coordinate it tested for its intersection within the quarter circle. Next the total number of valid points is directly accessed and the approximation of π is calculated.

In order to provide an accurate comparative analysis between the Java and C++ versions a limited direct port of the `java.util.Random` class has been implemented. This implementation is provided to avoid performance skews introduced by differences in the default random functions provided by the frameworks.

Multi-Threaded Implementation

The multi-threaded C++ implementation can be referenced in Appendix A and is a modified implementation of the derived algorithm code to allow for partitioning of the problem space and initiating concurrent execution in parallel threads. In order to implement a cross-platform multi-threaded C++ solution the implementation utilizes the `Thread`[8] module from the Boost[9] library.

The implementation moves the derived code from the single-threaded implementation to a class `PiCalculator` which implements a self-managing thread technique. Next the program creates a vector of `PiCalculator` objects equal in size to the detected number of available processors. In this fashion once all threads become active the program should use all available processor cores to complete execution. The program initializes these objects with a total number of execution iterations to the total number iterations specified divided by the total number of `PiCalculator` objects. Next the program initiates execution of and joins the threads to await their completion. Once the threads complete execution the results are fetched from the objects and the approximation of π is calculated.

OpenCL Implementation

In order to provide a cross platform implementation of a GPGPU-based solution the OpenCL[10] framework was chosen to support both a wide variety of graphics cards and supporting hardware. The OpenCL implementation can be referenced in Appendix A and

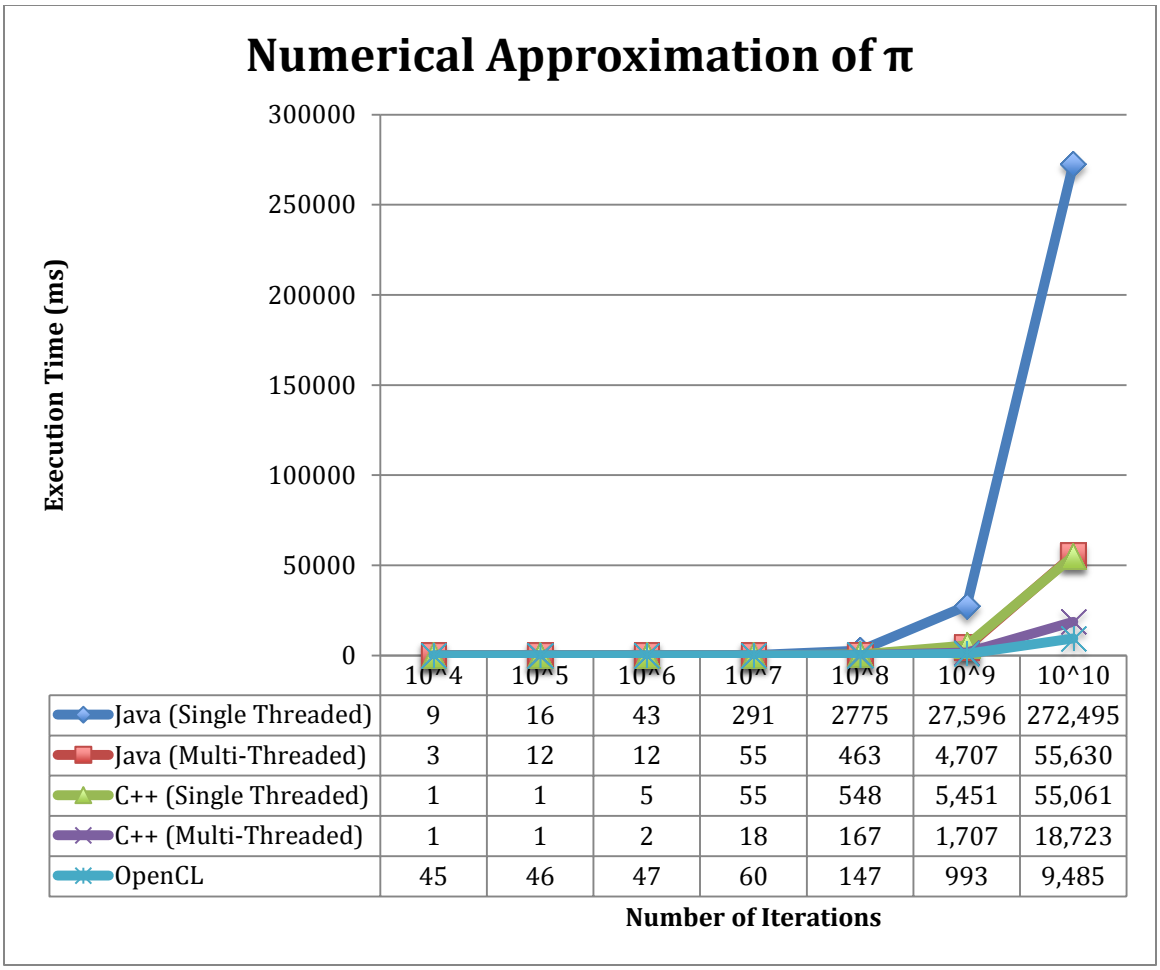
is a modified implementation of the derived algorithm code to allow for partitioning of the problem space into configurable work groups for execution in a highly parallel fashion.

The implementation creates a kernel, which iteratively executes the random functionality to generate random values for the x and y coordinate and the coordinate it tested for its intersection within the quarter circle. Next the implementation stores the calculated number of matching coordinates in a cache accessible by the current work group and sets a local fence to await for the completion of all workers in the local group. Once all of the workers in the group has finished the first worker sums all values for the locally accessible cache and writes the value out to a host-accessible cache.

In addition to a number of OpenCL-specific management functions the implementation also set up the execution of the kernel by defining the number of workers in each group, the number or iteration to be calculated by each worker, and the total number of workers to be created. The implementation also creates buffers to be read by the host and locally by the worker groups. The kernel work groups are then submitted for execution and upon completion the results are read from the host buffer and the approximation of π is calculated.

Benchmarks

Benchmarks were taken on a 15-inch MacBook Pro (Early 2011) running OSX Mountain Lion 10.7.5 with a 2.2Ghz Intel Core i7 processor, 8GB 1333 MHz DDR3 memory, and an AMD Radeon HD 6750M graphics card.



	10^4	10^5	10^6	10^7	10^8	10^9	10^10
Java (Single Threaded)	9	16	43	291	2775	27,596	272,495
Java (Multi-Threaded)	3	12	12	55	463	4,707	55,630
C++ (Single Threaded)	1	1	5	55	548	5,451	55,061
C++ (Multi-Threaded)	1	1	2	18	167	1,707	18,723
OpenCL	45	46	47	60	147	993	9,485

Fig. 2: Execution time for the specified number of approximation iterations

Slow Fourier Transform

The decomposition of data via Fourier transformation has many practical applications including signal processing, numerical analysis, and cryptography. The discrete-time Fourier transformation (DTFT) is one of the more applicable forms of the Fourier transformation, which operates on discretely sampled data and provides a representation of the data within the frequency domain. In the case of data, which is comprised of periodically sampled data the equation to calculate the transformation's complex solution, set is defined as:

(2.1)
$$X_k = \sum_{n=0}^{N-1} \left[x_n \times e^{\left(-i2\pi kn / N\right)} \right]$$

Deconstruction of the complex data solution into separate real and imaginary solution sets yields the following definitions:

$$Re_k = \sum_{n=0}^{N-1} \left[x_n \times \cos\left(2\pi kn / N\right) \right]$$
$$Im_k = \sum_{n=0}^{N-1} \left[x_n \times \sin\left(2\pi kn / N\right) \right] \quad (2.2)$$

Inspection of these equations shows that every element in the solution set requires fully processing all element of the input data set. However, each element of the solution set is not dependent on other solution elements. As a result the rudimentary solution for the Fourier transform is inherently parallelizable and will serve as another standardized benchmark and provide the basis for a comparative analysis of the applied computational frameworks, techniques, and architectures.

Algorithm Pseudocode

While much more efficient methods[11] have been derived for the calculation of Fourier transformations this paper will focus on the non-optimized representation of basic equations as executable code. This method is colloquially referred to as the Slow Fourier Transform and is represented by the following pseudocode:

```
FOR i = 1 to Input Array Size
   SET RealOutput[i] to 0.0
   SET ImaginaryOutput[i] to 0.0
   FOR j = 1 to Input Array Size
      RealOuput[i] += Input[j] * cos(2π*i*j / Input Array Size)
      ImaginaryOuput[i] += Input[j] * cos(2π*i*j / Input Array Size)
   END FOR
END FOR
```

Java Implementations

Single-Threaded Implementation

The single-threaded Java implementation can be referenced in Appendix B and performs a fixed number of iterations of an unmodified implementation of the derived algorithm pseudocode. Within the loop new random is created for the input data array. Each value of the real and imaginary output data arrays is calculated by accumulating the respective real and imaginary coefficients of each input data element within the array.

Multi-Threaded Implementation

The multi-threaded Java implementation can be referenced in Appendix B and is a modified implementation of the derived algorithm code to allow for partitioning of the problem space and initiating concurrent execution in parallel threads.

The implementation moves the derived code to a private class `FFTCalculator` which implements the `Callable<Long>` interface. Next the program creates a pool of threads equal in size to the detected number of available processors. In this fashion once all threads become active the program should use all available processor cores to complete execution. Next the program creates number of `FFTCalculator` objects equal in size to the detected number of available processors. The program initializes these

objects with a total number of execution iterations to the total number od iterations specified divided by the total number of `FFTCalculator` objects. Next the program links the executable objects to a thread in the allocated thread pool and initiates execution of the threads and awaits their completion.

C++ Implementations

Single-Threaded Implementation

The single-threaded C++ implementation can be referenced in Appendix B and performs a fixed number of iterations of an unmodified implementation of the derived algorithm pseudocode. Within the loop new random is created for the input data array. Each value of the real and imaginary output data arrays is calculated by accumulating the respective real and imaginary coefficients of each input data element within the array.

As with the C++ implementation of the numerical approximation of π in order to provide an accurate comparative analysis between the Java and C++ versions a limited direct port of the `java.util.Random` class has been implemented. This implementation is provided to avoid performance skews introduced by differences in the default random functions provided by the frameworks.

Multi-Threaded Implementation

The multi-threaded C++ implementation can be referenced in Appendix B and is a modified implementation of the derived algorithm code to allow for partitioning of the problem space and initiating concurrent execution in parallel threads. In order to implement a cross-platform multi-threaded C++ solution the implementation utilizes the `Thread` module from the Boost library.

The implementation moves the derived code from the single-threaded implementation to a class `FFTCalculator` which implements a self-managing thread technique. Next the program creates a vector of `FFTCalculator` objects equal in size to the detected number of available processors. In this fashion once all threads become active the program should use all available processor cores to complete execution. The program initializes these objects with a total number of execution iterations to the total number iterations specified divided by the total number of `FFTCalculator` objects. Next the program initiates execution of and joins the threads to await their completion.

OpenCL Implementation

As with the OpenCL implementation of the numerical approximation of π in order to provide a cross platform implementation of a GPGPU-based solution the OpenCL framework was chosen to support both a wide variety of graphics cards and supporting hardware. The OpenCL implementation can be referenced in Appendix B and is a modified implementation of the derived algorithm code to allow for partitioning of the problem space into configurable work groups for execution in a highly parallel fashion.

The implementation creates a kernel, whose first local work thread iteratively executes the random functionality to generate data for the work group's input data array. The kernel then calculates real and imaginary coefficients of the input data whose index is equal to that of the local worth thread id and stores the results in a cache accessible by the

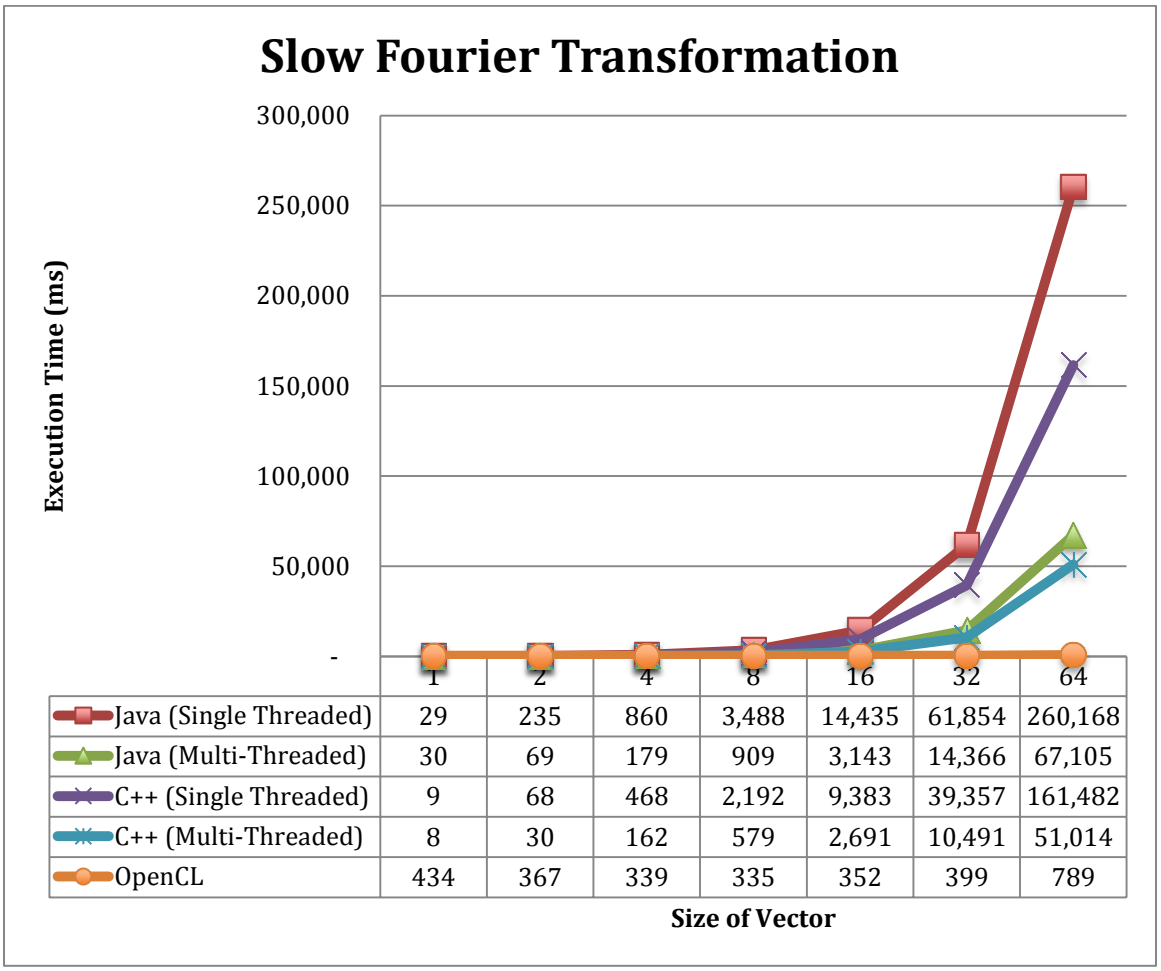
current work group and sets a local fence to await for the completion of all workers in the local group. Once all of the workers in the group have finished the first worker local work cache out to a host-accessible cache.

In addition to a number of OpenCL-specific management functions the implementation also set up the execution of the kernel by defining the number of workers in each group and the total number of workers to be created. The implementation also creates buffers to be read by the host and locally by the worker groups. The kernel work groups are then submitted for execution and upon completion the results are read from the host buffer.

Benchmarks

Benchmarks were taken on a 15-inch MacBook Pro (Early 2011) running OSX Mountain Lion 10.7.5 with a 2.2Ghz Intel Core i7 processor, 8GB 1333 MHz DDR3 memory, and an AMD Radeon HD 6750M graphics card.

Results



	1	2	4	8	16	32	64
Java (Single Threaded)	29	235	860	3,488	14,435	61,854	260,168
Java (Multi-Threaded)	30	69	179	909	3,143	14,366	67,105
C++ (Single Threaded)	9	68	468	2,192	9,383	39,357	161,482
C++ (Multi-Threaded)	8	30	162	579	2,691	10,491	51,014
OpenCL	434	367	339	335	352	399	789

Fig. 3: Execution time for 1,000,000 FFT calculations with the specified size of the data vector

Conclusions

The benchmark results indicate there is a clear performance benefit to utilizing a multi-threaded solution as opposed to a single-threaded implementation. For both algorithms there was an average decrease in execution time of 67.159% for the Java implementations and an average decrease in execution time of 53.608% for the C++ implementations. However, these improvements in execution time are not necessarily beneficial until the data set is of a sufficient size such that the savings in computation time exceeds the computational overhead associated with the setup and management of a multi-threaded solution.

This concept of a break-even point for computational time is clearly demonstrated when inspecting the performance improvements of the OpenCL implementation in comparison to both the single-threaded and multi-threaded C++ implementations of both algorithms. While at the largest sizes of data sets the OpenCL implementation offered an average decrease in execution time of 66.057% for the numerical approximation of π and 98.982% for the Slow Fourier Transform at smaller sized data sets the implementations were over 4000% slower. Such benchmarks highlights the overhead associated with utilizing OpenCL as a computational solution- most notably the high cost associated with transferring data to and from the host system to the OpenCL device and the inherent inefficiency associated with incomplete usage of computational resources when executing against smaller data sets.

In practice it can be seen that for computationally intensive algorithms there is a clear performance benefit to implement a multi-threaded solution when a target data set is expected to be of sufficient size. Furthermore, in cases where a multi-threaded solution exists for a highly computational algorithm extensive gains can be achieved through utilizing emerging GPGPU technologies such as OpenCL.

When implementing solutions to highly computational algorithms there is a clear benefit to designing with concurrency as key factor. While smaller data sets can fall back to a single-threaded solution where such overhead is determined to be unacceptable when facing moderately to large sized data sets can be expected to range from significant to extreme. Furthermore, with emerging technologies such as dedicated high-speed data busses between GPU and system memory, heterogeneous memory access architectures, and on-die multithreaded command arbitration modules the barrier to implementing efficient concurrency in both traditional and emerging architectures is being continually lowered and the usage of such technologies should be considered as a core component at all levels of software development.

References

[1] "Parallel Programming and Computing Platform | CUDA | NVIDIA" Accessed March 5, 2013. http://www.nvidia.com/object/cuda_home_new.html

[2] Singh, Jaswinder Pal, and John L. Hennessy. "An empirical investigation of the effectiveness and limitations of automatic parallelization." The MIT Press, Cambridge, Mass (1992): 213-240.

[3] Grochowski, Ed, Ronny Ronen, John Shen, and Hong Wang. "Best of both latency and throughput." In Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on, pp. 236-243. IEEE, 2004.

[4] "Tesla® Kepler™ GPU Accelerators" Last Modified October 2012. http://www.nvidia.com/content/tesla/pdf/Tesla-KSeries-Overview-LR.pdf

[5] "AMD Radeon™ HD 7990 Graphics" Accessed March 6, 2013. http://www.amd.com/us/products/desktop/graphics/7000/7990/Pages/radeon-7990.aspx

[6] "y-cruncher - A Multi-Threaded Pi-Program" Last Modified February 17, 2013. http://www.numberworld.org/y-cruncher/

[7] Bailey, David, Peter Borwein, and Simon Plouffe. "On the rapid computation of various polylogarithmic constants." Mathematics of Computation 66, no. 218 (1997): 903-914.

[8] "Chapter 30. Thread 4.0.0 - 1.53.0" Accessed March 7, 2013. http://www.boost.org/doc/libs/1_53_0/doc/html/thread.html

[9] "Boost C++ Libraries" Accessed March 7, 2013. http://www.boost.org/

[10] "OpenCL - The open standard for parallel programming of heterogeneous systems" Accessed March 7, 2013. http://www.khronos.org/opencl/

[11] Cooley, James W., and John W. Tukey. "An algorithm for the machine calculation of complex Fourier series." Mathematics of computation (1965): 297-301.

Appendix A – Numerical Approximation of π Source Code

Single-Threaded Java Implementation

App.java

```
package com.geocent.geocentlabs.pi;

import java.util.Random;

public class App
{
    public static final long NUM_ITERATIONS = 1000000000L;
    public static final Random RANDOM_GENERATOR = new Random();
    public static void main( String[] args )
    {
        long numInsideQuad = 0L;
        long startTime = System.currentTimeMillis();

        for(long i=0L; i < NUM_ITERATIONS; i++) {
            float x = RANDOM_GENERATOR.nextFloat();
            float y = RANDOM_GENERATOR.nextFloat();

            if(x*x + y*y < 1) {
                numInsideQuad++;
            }
        }
        long stopTime = System.currentTimeMillis();

        System.out.printf("PI: %.10f\n", 4 * (float)numInsideQuad / NUM_ITERATIONS);
        System.out.printf("Application run time: %dms\n", (stopTime - startTime));
    }
}
```

Multi-Threaded Java Implementation

App.java

```
package com.geocent.geocentlabs.pithreaded;

import java.util.LinkedList;
import java.util.List;
import java.util.Random;
import java.util.concurrent.*;

public class App {

    public static final long NUM_ITERATIONS = 1000000000L;
    public static final int NUM_THREADS = Runtime.getRuntime().availableProcessors();

    private static class PiCalculator implements Callable<Long> {

        private long numIterations = 0L;
        private final Random RANDOM_GENERATOR = new Random();

        public PiCalculator(long numIterations, long seed) {
            this.numIterations = numIterations;
            RANDOM_GENERATOR.setSeed(seed);
        }

        @Override
        public Long call() throws Exception {
            long numInsideQuad = 0L;

            for (int i = 0; i < this.numIterations; i++) {
                float x = RANDOM_GENERATOR.nextFloat();
                float y = RANDOM_GENERATOR.nextFloat();

                if (x*x + y*y < 1) {
                    numInsideQuad++;
                }
            }

            return numInsideQuad;
        }
    }

    public static void main(String[] args)
        throws ExecutionException, InterruptedException {
        System.out.printf("Executing with %d threads.\n", NUM_THREADS);

        ExecutorService threadPool = Executors.newFixedThreadPool(NUM_THREADS);
        ExecutorCompletionService agentCompletionService =
                new ExecutorCompletionService(threadPool);

        List<Future> taskList = new LinkedList<Future>();
        Random randomGenerator = new Random();
        for (long i = 0L; i < NUM_THREADS; i++) {
            taskList.add(agentCompletionService.submit((
                new PiCalculator(
                        NUM_ITERATIONS / NUM_THREADS, randomGenerator.nextLong()
                ))));
        }

        long startTime = System.currentTimeMillis();

        long numInsideQuad = 0L;
        while (!taskList.isEmpty()) {
            Future task = agentCompletionService.take();
            numInsideQuad += (Long)task.get();
            taskList.remove(task);
        }

        long stopTime = System.currentTimeMillis();
```

```
        threadPool.shutdown();

        System.out.printf("PI: %.10f\n", 4 * (double) numInsideQuad / NUM_ITERATIONS);
        System.out.printf("Application run time: %dms\n", (stopTime - startTime));
    }
}
```

Single-Threaded C++ Implementation

main.cpp

```
#include <cstdlib>
#include <ctime>
#include <iomanip>
#include <boost/timer/timer.hpp>

class Random {
private:
    long seed;
public:

    Random(long seed) {
        setSeed(seed);
    }

    void setSeed(long newSeed) {
        seed = (newSeed ^ 0x5DEECE66DL) & ((1L << 48) - 1);
    }

    int next(int bits) {
        seed = (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
        return (seed >> (48 - bits));
    }

    float nextFloat() {
        return next(24) / (float) (1 << 24);
    }

    long nextLong() {
        return ((long) next(32) << 32) +next(32);
    }

};

int main(int argc, char** argv) {
    const unsigned long NUM_ITERATIONS = 1000000000L;

    Random randomGenerator(time(0));

    boost::timer::cpu_timer timer;
    unsigned long numInsideQuad = 0;
    for (unsigned long i = 0L; i < NUM_ITERATIONS; i++) {
        float x = randomGenerator.nextFloat();
        float y = randomGenerator.nextFloat();

        if (x * x + y * y < 1) {
            numInsideQuad++;
        }
    }

    boost::timer::cpu_times const elapsed_times(timer.elapsed());
    boost::timer::nanosecond_type const elapsed(elapsed_times.wall);

    std::cout << "PI: " << std::setprecision(10) <<
        4 * (float) numInsideQuad / NUM_ITERATIONS << std::endl;
    std::cout << "Application run time: " << elapsed / 1000000LL << "ms" << std::endl;

    return 0;
}
```

Multi-Threaded C++ Implementation

main.cpp

```
#include <cstdlib>
#include <ctime>
#include <vector>
#include <boost/thread.hpp>
#include <boost/timer/timer.hpp>

class Random {
private:
    long seed;
public:

    Random(long seed) {
        setSeed(seed);
    }

    void setSeed(long newSeed) {
        seed = (newSeed ^ 0x5DEECE66DL) & ((1L << 48) - 1);
    }

    int next(int bits) {
        seed = (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
        return (seed >> (48 - bits));
    }

    float nextFloat() {
        return next(24) / (float) (1 << 24);
    }

    long nextLong() {
        return ((long) next(32) << 32) +next(32);
    }
};

class PiCalculator {
private:
    boost::thread m_Thread;

    unsigned long numIterations;
    unsigned long numInsideQuad;
    Random randomGenerator;

    void runProcess() {
        for (long i = 0L; i < numIterations; i++) {
            float x = randomGenerator.nextFloat();
            float y = randomGenerator.nextFloat();

            if (x * x + y * y < 1) {
                numInsideQuad++;
            }
        }
    }

public:

    PiCalculator(long numIterations, long seed) :
    numIterations(numIterations),
    numInsideQuad(0L),
    randomGenerator(0) {
        this->numIterations = numIterations;
        randomGenerator.setSeed(seed);
    }

    unsigned long getNumInsideQuad() {
        return numInsideQuad;
    }
```

```
    void start() {
        m_Thread = boost::thread(&PiCalculator::runProcess, this);
    }

    void join() {
        m_Thread.join();
    }
};

int main(int argc, char** argv) {
    const unsigned long NUM_ITERATIONS = 1000000000L;
    const int NUM_THREADS = boost::thread::hardware_concurrency();
    std::cout << "Executing with " << NUM_THREADS << " threads." << std::endl;

    std::vector<PiCalculator*> calculators;

    Random randomGenerator(time(0));

    boost::timer::cpu_timer timer;
    for (int i = 0; i < NUM_THREADS; i++) {
        calculators.push_back(new PiCalculator(
            NUM_ITERATIONS / NUM_THREADS, randomGenerator.nextLong()
        ));
    }

    for (std::vector<PiCalculator *>::iterator i = calculators.begin();
        i != calculators.end(); i++) {
        (*i)->start();
    }

    for (std::vector<PiCalculator *>::iterator i = calculators.begin();
        i != calculators.end(); i++) {
        (*i)->join();
    }

    unsigned long numInsideQuad = 0;
    for (std::vector<PiCalculator *>::iterator i = calculators.begin();
        i != calculators.end(); i++) {
        numInsideQuad += (*i)->getNumInsideQuad();
        delete *i;
    }

    boost::timer::cpu_times const elapsed_times(timer.elapsed());
    boost::timer::nanosecond_type const elapsed(elapsed_times.wall);

    std::cout << "PI: " << std::setprecision(10) <<
        4 * (float) numInsideQuad / NUM_ITERATIONS << std::endl;
    std::cout << "Application run time: " << elapsed / 1000000LL << "ms" << std::endl;
    return 0;
}
```

OpenCL Implementation

main.cpp

```
#include <exception>
#include <iostream>
#include <boost/timer/timer.hpp>

// OpenCL Includes
#include <oclUtils.h>
#include <shrQATest.h>
#include <CL/cl_platform.h>

class OClException : public std::exception {};

class OpenClManager {
private:
    cl_platform_id cpPlatform;
    cl_context cxGPUContext;
    cl_command_queue cqCommandQueue;
    cl_device_id cdDevice;
    cl_program cpProgram;
    cl_kernel ckKernel;

public:

    OpenClManager() {
        cl_int ciErr;

        //Get an OpenCL platform
        shrLog("clGetPlatformID...\n");
        if (clGetPlatformIDs(1, &cpPlatform, NULL) != CL_SUCCESS) {
            shrLog("Error in clGetPlatformID, Line %u in file %s !!!\n\n",
                __LINE__, __FILE__);
            throw OClException();
        }

        //Get the devices
        shrLog("clGetDeviceIDs...\n");
        if (clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &cdDevice, NULL) !=
            CL_SUCCESS) {
            shrLog("Error in clGetDeviceIDs, Line %u in file %s !!!\n\n",
                __LINE__, __FILE__);
            throw OClException();
        }

        //Create the context
        shrLog("clCreateContext...\n");
        cxGPUContext = clCreateContext(0, 1, &cdDevice, NULL, NULL, &ciErr);
        if (ciErr != CL_SUCCESS) {
            shrLog("Error in clCreateContext, Line %u in file %s !!!\n\n",
                __LINE__, __FILE__);
            throw OClException();
        }

        // Create a command-queue
        shrLog("clCreateCommandQueue...\n");
        cqCommandQueue = clCreateCommandQueue(cxGPUContext, cdDevice, 0, &ciErr);
        if (ciErr != CL_SUCCESS) {
            shrLog("Error in clCreateCommandQueue, Line %u in file %s !!!\n\n",
                __LINE__, __FILE__);
            throw OClException();
        }
    }

    ~OpenClManager() {
        if (cxGPUContext) {
            clReleaseContext(cxGPUContext);
        }
```

```
        if (cqCommandQueue) {
            clReleaseCommandQueue(cqCommandQueue);
        }

        if (ckKernel) {
            clReleaseKernel(ckKernel);
        }

        if (cpProgram) {
            clReleaseProgram(cpProgram);
        }
    }

    cl_kernel createKernel(const char* sourceFile, const char* kernelName) {
        cl_int ciErr;
        size_t szKernelLength;

        // Read the OpenCL kernel in from source file
        shrLog("oclLoadProgSource (%s)...\n", sourceFile);
        char* cPathAndName = shrFindFilePath(sourceFile, NULL);
        char* cSourceCL = oclLoadProgSource(cPathAndName, "", &szKernelLength);

        // Create the program
        shrLog("clCreateProgramWithSource...\n");
        cpProgram = clCreateProgramWithSource(cxGPUContext, 1, (const char **)
            &cSourceCL, &szKernelLength, &ciErr);
        if (cPathAndName) {
            free(cPathAndName);
        }
        if (cSourceCL) {
            free(cSourceCL);
        }
        if (ciErr != CL_SUCCESS) {
            shrLog("Error in clCreateProgramWithSource, Line %u in file %s !!!\n\n",
                __LINE__, __FILE__);
            throw OClException();
        }

        shrLog("clBuildProgram...\n");
        if (clBuildProgram(cpProgram, 0, NULL, NULL, NULL, NULL) != CL_SUCCESS) {
            shrLog("Error in clBuildProgram, Line %u in file %s !!!\n\n",
                __LINE__, __FILE__);
            throw OClException();
        }

        // Create the kernel
        shrLog("clCreateKernel (%s)...\n", kernelName);
        ckKernel = clCreateKernel(cpProgram, kernelName, &ciErr);
        if (ciErr != CL_SUCCESS) {
            shrLog("Error in clCreateKernel, Line %u in file %s !!!\n\n",
                __LINE__, __FILE__);
            throw OClException();
        }

        return ckKernel;
    }

    void setKernelArg(cl_uint argumentIndex, void* argumentValue, size_t argumentSize) {
        shrLog("clSetKernelArg %d...\n", argumentIndex);
        if (clSetKernelArg(ckKernel, argumentIndex, argumentSize, argumentValue) !=
            CL_SUCCESS) {
            shrLog("Error in clSetKernelArg, Line %u in file %s !!!\n\n",
                __LINE__, __FILE__);
            throw OClException();
        }
    }

    void launchKernel(const size_t *globalWorkSize, const size_t *localWorkSize) {
        cl_int ciErr;
        cl_event event;
```

```
        shrLog("clEnqueueNDRangeKernel (G:%d) (L:%d)...\n", *globalWorkSize,
            *localWorkSize);
        ciErr = clEnqueueNDRangeKernel(cqCommandQueue, ckKernel, 1, NULL, globalWorkSize,
            localWorkSize, 0, NULL, &event);
        clWaitForEvents(1, &event);
        if (ciErr != CL_SUCCESS) {
            shrLog("Error in clEnqueueNDRangeKernel (%d), Line %u in file %s !!!\n\n",
                ciErr, __LINE__, __FILE__);
            throw OClException();
        }
        clReleaseEvent(event);
    }

    cl_mem createBuffer(size_t bufferSize) {
        cl_int ciErr;
        cl_mem cmDevDst = clCreateBuffer(cxGPUContext, CL_MEM_WRITE_ONLY, bufferSize,
            NULL, &ciErr);
        shrLog("clCreateBuffer (%d)...\n", bufferSize);
        if (ciErr != CL_SUCCESS) {
            shrLog("Error in clCreateBuffer, Line %u in file %s !!!\n\n",
                __LINE__, __FILE__);
            throw OClException();
        }
        return cmDevDst;
    }

    void readBuffer(cl_mem buffer, void* results, size_t bufferSize) {
        cl_int ciErr;
        cl_event event;

        shrLog("clEnqueueReadBuffer...\n\n");
        ciErr = clEnqueueReadBuffer(cqCommandQueue, buffer, CL_TRUE, 0, bufferSize,
            results, 0, NULL, &event);
        clWaitForEvents(1, &event);

        if (ciErr != CL_SUCCESS) {
            shrLog("Error in clEnqueueReadBuffer, Line %u in file %s !!!\n\n",
                __LINE__, __FILE__);
            throw OClException();
        }
        clReleaseEvent(event);
    }
};

int main(int argc, char **argv) {
    shrQAStart(argc, argv);

    const size_t szLocalWorkSize = 256;
    const size_t szLocalWorkIterations = 1024;
    const size_t szGlobalWorkSize = szLocalWorkSize * 38147;
    const size_t szGlobalGroups = szGlobalWorkSize / szLocalWorkSize;
    unsigned long seed = time(0);

    std::cout << "Calculating PI with " <<
        szGlobalWorkSize * szLocalWorkIterations << " iterations." << std::endl;

    boost::timer::cpu_timer timer;
    cl_mem cmResults;
    cl_uint* hostResults = (cl_uint *) malloc(sizeof (cl_uint) * szGlobalWorkSize);

    try {
        OpenClManager oclManager;
        oclManager.createKernel("PiCalculator.cl", "PiCalculator");
        cmResults = oclManager.createBuffer(sizeof (cl_uint) * szGlobalGroups);
        oclManager.setKernelArg(0, (void *) &cmResults, sizeof (cl_mem));
        oclManager.setKernelArg(1, NULL, sizeof (cl_ushort) * szLocalWorkSize);
        oclManager.setKernelArg(2, (void *) &szLocalWorkIterations, sizeof (cl_ushort));
        oclManager.setKernelArg(3, (void *) &seed, sizeof (unsigned long));
        oclManager.launchKernel(&szGlobalWorkSize, &szLocalWorkSize);
```

```
        oclManager.readBuffer(cmResults, hostResults, sizeof (cl_uint) * szGlobalGroups);

        unsigned long numInsideQuad = 0;
        for (unsigned long i = 0; i < szGlobalGroups; i++) {
            numInsideQuad += hostResults[i];
        }
        boost::timer::cpu_times const elapsed_times(timer.elapsed());
        boost::timer::nanosecond_type const elapsed(elapsed_times.wall);

        shrLog("PI: %.10f\n",
            4.0 * numInsideQuad / (szGlobalWorkSize * szLocalWorkIterations));
        shrLog("Application run time: %dms\n", elapsed / 1000000LL);

    } catch (OClException &ex) {
        // finalize logs and leave
        shrQAFinishExit(argc, (const char **) argv, QA_FAILED);
    }

    if (cmResults) {
        clReleaseMemObject(cmResults);
    }

    if (hostResults) {
        free(hostResults);
    }

    // finalize logs and leave
    shrQAFinishExit(argc, (const char **) argv, QA_PASSED);
}
```

PiCalculator.cl

```
__kernel void PiCalculator(__global unsigned int* results, __local unsigned short*
summation, unsigned short iterations,
    unsigned long kernelSeed) {
    if(get_global_id(0) > get_global_size(0)) {
        return;
    }

    unsigned long _seed = ((get_global_id(0) * kernelSeed) ^ 0x5DEECE66DL) &
         ((1L << 48) - 1);
    float _x = 0.0;
    float _y = 0.0;
    unsigned short sum = 0;
    for(unsigned short _i = 0; _i < iterations; _i++) {
        _seed = (_seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
        _x = (_seed >> (24)) / (float) (1 << 24);
        _seed = (_seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
        _y = (_seed >> (24)) / (float) (1 << 24);

        if((_x*_x + _y*_y) < 1) {
            sum++;
        }
    }
    summation[get_local_id(0)] = sum;

    barrier(CLK_LOCAL_MEM_FENCE);
    if(get_local_id(0) == 0) {
        unsigned int _result = 0;
        for(unsigned short _i = 0; _i < get_local_size(0); _i++) {
            _result += summation[_i];
        }
        results[get_global_id(0) / get_local_size(0)] = _result;
    }
}
```

Appendix B – Slow Fourier Transform Source Code

Single-Threaded Java Implementation

App.java

```
package com.geocent.geocentlabs.fft;

import java.util.Random;

public class App {

    public static final long NUM_ITERATIONS = 1000000;
    public static final int FFT_DIMENSION = 64;
    public static final Random RANDOM_GENERATOR = new Random();

    public static void main(String[] args) {
        long startTime = System.currentTimeMillis();

        float[] dataInput = new float[FFT_DIMENSION];
        float[] realOutput = new float[FFT_DIMENSION];
        float[] imagOutput = new float[FFT_DIMENSION];

        for (long iter = 0L; iter < NUM_ITERATIONS; iter++) {

            for (int i = 0; i < FFT_DIMENSION; i++) {
                dataInput[i] = RANDOM_GENERATOR.nextFloat();
            }

            for (int i = 0; i < FFT_DIMENSION; i++) {
                realOutput[i] = 0;
                imagOutput[i] = 0;

                for (int j = 0; j < FFT_DIMENSION; j++) {
                    realOutput[i] += dataInput[j] * Math.cos(2.0 * Math.PI * i * j /
                        FFT_DIMENSION);
                    imagOutput[i] += dataInput[j] * Math.sin(2.0 * Math.PI * i * j /
                        FFT_DIMENSION);
                }
            }
        }

        long stopTime = System.currentTimeMillis();

        System.out.printf("Application run time: %dms\n", (stopTime - startTime));
    }
}
```

Multi-Threaded Java Implementation

App.java

```
package com.geocent.geocentlabs.fftthreaded;

import java.util.LinkedList;
import java.util.List;
import java.util.Random;
import java.util.concurrent.*;

public class App {

    public static final long NUM_ITERATIONS = 1000000;
    public static final int FFT_DIMENSION = 64;
    public static final int NUM_THREADS = Runtime.getRuntime().availableProcessors();

    private static class FFTCalculator implements Callable<Long> {

        private long numIterations = 0L;
        private final Random RANDOM_GENERATOR = new Random();

        public FFTCalculator(long numIterations, long seed) {
            this.numIterations = numIterations;
            RANDOM_GENERATOR.setSeed(seed);
        }

        @Override
        public Long call() throws Exception {
            float[] dataInput = new float[FFT_DIMENSION];
            float[] realOutput = new float[FFT_DIMENSION];
            float[] imagOutput = new float[FFT_DIMENSION];

            for (long iter = 0L; iter < numIterations; iter++) {

                for (int i = 0; i < FFT_DIMENSION; i++) {
                    dataInput[i] = RANDOM_GENERATOR.nextFloat();
                }

                for (int i = 0; i < FFT_DIMENSION; i++) {
                    realOutput[i] = 0;
                    imagOutput[i] = 0;

                    for (int j = 0; j < FFT_DIMENSION; j++) {
                        realOutput[i] += dataInput[j] *
                            Math.cos(2.0 * Math.PI * i * j / FFT_DIMENSION);
                        imagOutput[i] += dataInput[j] *
                            Math.sin(2.0 * Math.PI * i * j / FFT_DIMENSION);
                    }
                }
            }

            return 0L;
        }
    }

    public static void main(String[] args)
        throws ExecutionException, InterruptedException {
        System.out.printf("Executing with %d threads.\n", NUM_THREADS);

        ExecutorService threadPool = Executors.newFixedThreadPool(NUM_THREADS);
        ExecutorCompletionService agentCompletionService =
            new ExecutorCompletionService(threadPool);

        List<Future> taskList = new LinkedList<Future>();
        Random randomGenerator = new Random();
        for (long i = 0L; i < NUM_THREADS; i++) {
            taskList.add(agentCompletionService.submit((new FFTCalculator(
                NUM_ITERATIONS / NUM_THREADS, randomGenerator.nextLong())
        )));
```

```
        }

        long startTime = System.currentTimeMillis();

        while (!taskList.isEmpty()) {
            Future task = agentCompletionService.take();
            taskList.remove(task);
        }

        long stopTime = System.currentTimeMillis();

        threadPool.shutdown();

        System.out.printf("Application run time: %dms\n", (stopTime - startTime));
    }
}
```

Single-Threaded C++ Implementation

main.cpp

```
#include <cmath>
#include <cstdlib>
#include <ctime>
#include <boost/timer/timer.hpp>

class Random {
private:
    long seed;
public:

    Random(long seed) {
        setSeed(seed);
    }

    void setSeed(long newSeed) {
        seed = (newSeed ^ 0x5DEECE66DL) & ((1L << 48) - 1);
    }

    int next(int bits) {
        seed = (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
        return (seed >> (48 - bits));
    }

    float nextFloat() {
        return next(24) / (float) (1 << 24);
    }

    long nextLong() {
        return ((long) next(32) << 32) + next(32);
    }

};

int main(int argc, char** argv) {
    const unsigned long NUM_ITERATIONS = 1000000;
    const int FFT_DIMENSION = 64;

    Random randomGenerator(time(0));

    boost::timer::cpu_timer timer;

    float* dataInput = new float[FFT_DIMENSION];
    float* realOutput = new float[FFT_DIMENSION];
    float* imagOutput = new float[FFT_DIMENSION];

    for (unsigned long iter = 0L; iter < NUM_ITERATIONS; iter++) {

        for (int i = 0; i < FFT_DIMENSION; i++) {
            dataInput[i] = randomGenerator.nextFloat();
        }

        for (int i = 0; i < FFT_DIMENSION; i++) {
            realOutput[i] = 0.0;
            imagOutput[i] = 0.0;

            for(int j = 0; j < FFT_DIMENSION; j++) {
                realOutput[i] += dataInput[j] * cos(2.0 * M_PI * i * j / FFT_DIMENSION);
                imagOutput[i] += dataInput[j] * sin(2.0 * M_PI * i * j / FFT_DIMENSION);
            }
        }
    }

    delete dataInput;
    delete realOutput;
    delete imagOutput;
```

```
    boost::timer::cpu_times const elapsed_times(timer.elapsed());
    boost::timer::nanosecond_type const elapsed(elapsed_times.wall);

    std::cout << "Application run time: " << elapsed / 1000000LL << "ms" << std::endl;

    return 0;
}
```

Multi-Threaded C++ Implementation

main.cpp

```
#include <cstdlib>
#include <ctime>
#include <vector>
#include <boost/thread.hpp>
#include <boost/timer/timer.hpp>

class Random {
private:
    long seed;
public:

    Random(long seed) {
        setSeed(seed);
    }

    void setSeed(long newSeed) {
        seed = (newSeed ^ 0x5DEECE66DL) & ((1L << 48) - 1);
    }

    int next(int bits) {
        seed = (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
        return (seed >> (48 - bits));
    }

    float nextFloat() {
        return next(24) / (float) (1 << 24);
    }

    long nextLong() {
        return ((long) next(32) << 32) +next(32);
    }
};

class FFTCalculator {
private:
    boost::thread m_Thread;

    unsigned long numIterations;
    unsigned int fftDimensions;
    Random randomGenerator;

    void runProcess() {
        float* dataInput = new float[fftDimensions];
        float* realOutput = new float[fftDimensions];
        float* imagOutput = new float[fftDimensions];

        for (unsigned long iter = 0L; iter < numIterations; iter++) {

            for (int i = 0; i < fftDimensions; i++) {
                dataInput[i] = randomGenerator.nextFloat();
            }

            for (int i = 0; i < fftDimensions; i++) {
                realOutput[i] = 0.0;
                imagOutput[i] = 0.0;

                for (int j = 0; j < fftDimensions; j++) {
                    realOutput[i] += dataInput[j] *
                        cos(2.0 * M_PI * i * j / fftDimensions);
                    imagOutput[i] += dataInput[j] *
                        sin(2.0 * M_PI * i * j / fftDimensions);
                }
            }
        }

        delete dataInput;
```

```
        delete realOutput;
        delete imagOutput;

    }

public:

    FFTCalculator(long numIterations, int fftDimensions, long seed) :
    numIterations(numIterations),
    fftDimensions(fftDimensions),
    randomGenerator(0) {
        randomGenerator.setSeed(seed);
    }


    void start() {
        m_Thread = boost::thread(&FFTCalculator::runProcess, this);
    }

    void join() {
        m_Thread.join();
    }
};

int main(int argc, char** argv) {
    const unsigned long NUM_ITERATIONS = 1000000;
    const int FFT_DIMENSION = 64;

    const int NUM_THREADS = boost::thread::hardware_concurrency();
    std::cout << "Executing with " << NUM_THREADS << " threads." << std::endl;

    std::vector<FFTCalculator*> calculators;

    Random randomGenerator(time(0));

    boost::timer::cpu_timer timer;
    for (int i = 0; i < NUM_THREADS; i++) {
        calculators.push_back(new FFTCalculator(NUM_ITERATIONS / NUM_THREADS,
            FFT_DIMENSION, randomGenerator.nextLong()));
    }

    for (std::vector<FFTCalculator *>::iterator i = calculators.begin();
        i != calculators.end(); i++) {
        (*i)->start();
    }

    for (std::vector<FFTCalculator *>::iterator i = calculators.begin();
        i != calculators.end(); i++) {
        (*i)->join();
    }

    boost::timer::cpu_times const elapsed_times(timer.elapsed());
    boost::timer::nanosecond_type const elapsed(elapsed_times.wall);

    std::cout << "Application run time: " << elapsed / 1000000LL << "ms" << std::endl;
    return 0;
}
```

OpenCL Implementation

main.cpp

```
#include <exception>
#include <iostream>
#include <boost/timer/timer.hpp>

// OpenCL Includes
#include <oclUtils.h>
#include <shrQATest.h>
#include <CL/cl_platform.h>

class OClException : public std::exception {};

class OpenClManager {
private:
    cl_platform_id cpPlatform;
    cl_context cxGPUContext;
    cl_command_queue cqCommandQueue;
    cl_device_id cdDevice;
    cl_program cpProgram;
    cl_kernel ckKernel;

public:
    OpenClManager() {
        cl_int ciErr;

        //Get an OpenCL platform
        shrLog("clGetPlatformID...\n");
        if (clGetPlatformIDs(1, &cpPlatform, NULL) != CL_SUCCESS) {
            shrLog("Error in clGetPlatformID, Line %u in file %s !!!\n\n",
                __LINE__, __FILE__);
            throw OClException();
        }

        //Get the devices
        shrLog("clGetDeviceIDs...\n");
        if (clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &cdDevice, NULL) !=
            CL_SUCCESS) {
            shrLog("Error in clGetDeviceIDs, Line %u in file %s !!!\n\n",
                __LINE__, __FILE__);
            throw OClException();
        }

        //Create the context
        shrLog("clCreateContext...\n");
        cxGPUContext = clCreateContext(0, 1, &cdDevice, NULL, NULL, &ciErr);
        if (ciErr != CL_SUCCESS) {
            shrLog("Error in clCreateContext, Line %u in file %s !!!\n\n",
                __LINE__, __FILE__);
            throw OClException();
        }

        // Create a command-queue
        shrLog("clCreateCommandQueue...\n");
        cqCommandQueue = clCreateCommandQueue(cxGPUContext, cdDevice, 0, &ciErr);
        if (ciErr != CL_SUCCESS) {
            shrLog("Error in clCreateCommandQueue, Line %u in file %s !!!\n\n",
                __LINE__, __FILE__);
            throw OClException();
        }
    }

    ~OpenClManager() {
        if (cxGPUContext) {
            clReleaseContext(cxGPUContext);
        }

        if (cqCommandQueue) {
```

```
            clReleaseCommandQueue(cqCommandQueue);
        }

        if (ckKernel) {
            clReleaseKernel(ckKernel);
        }

        if (cpProgram) {
            clReleaseProgram(cpProgram);
        }
    }

    cl_kernel createKernel(const char* sourceFile, const char* kernelName) {
        cl_int ciErr;
        size_t szKernelLength;

        // Read the OpenCL kernel in from source file
        shrLog("oclLoadProgSource (%s)...\n", sourceFile);
        char* cPathAndName = shrFindFilePath(sourceFile, NULL);
        char* cSourceCL = oclLoadProgSource(cPathAndName, "", &szKernelLength);

        // Create the program
        shrLog("clCreateProgramWithSource...\n");
        cpProgram = clCreateProgramWithSource(cxGPUContext, 1,
            (const char **) &cSourceCL, &szKernelLength, &ciErr);
        if (cPathAndName) {
            free(cPathAndName);
        }
        if (cSourceCL) {
            free(cSourceCL);
        }
        if (ciErr != CL_SUCCESS) {
            shrLog("Error in clCreateProgramWithSource, Line %u in file %s !!!\n\n",
                __LINE__, __FILE__);
            throw OClException();
        }

        shrLog("clBuildProgram...\n");
        if (clBuildProgram(cpProgram, 0, NULL, NULL, NULL, NULL) != CL_SUCCESS) {
            shrLog("Error in clBuildProgram, Line %u in file %s !!!\n\n",
                __LINE__, __FILE__);
            throw OClException();
        }

        // Create the kernel
        shrLog("clCreateKernel (%s)...\n", kernelName);
        ckKernel = clCreateKernel(cpProgram, kernelName, &ciErr);
        if (ciErr != CL_SUCCESS) {
            shrLog("Error in clCreateKernel, Line %u in file %s !!!\n\n",
                __LINE__, __FILE__);
            throw OClException();
        }

        return ckKernel;
    }

    void setKernelArg(cl_uint argumentIndex, void* argumentValue, size_t argumentSize) {
        cl_int ciErr;
        shrLog("clSetKernelArg %d (%d)...\n", argumentIndex, argumentSize);
        ciErr = clSetKernelArg(ckKernel, argumentIndex, argumentSize, argumentValue);
        if (ciErr != CL_SUCCESS) {
            shrLog("Error (%d) in clSetKernelArg, Line %u in file %s !!!\n\n", ciErr,
                __LINE__, __FILE__);
            throw OClException();
        }
    }
    void launchKernel(const size_t *globalWorkSize, const size_t *localWorkSize) {
        cl_int ciErr;
        cl_event event;
```

```
        shrLog("clEnqueueNDRangeKernel (G:%d) (L:%d)...\n", *globalWorkSize,
            *localWorkSize);
        ciErr = clEnqueueNDRangeKernel(cqCommandQueue, ckKernel, 1, NULL, globalWorkSize,
            localWorkSize, 0, NULL, NULL);
        clWaitForEvents(1, &event);
        if (ciErr != CL_SUCCESS) {
            shrLog("Error in clEnqueueNDRangeKernel (%d), Line %u in file %s !!!\n\n",
                ciErr, __LINE__, __FILE__);
            throw OClException();
        }
        clReleaseEvent(event);
    }

    cl_mem createBuffer(size_t bufferSize) {
        cl_int ciErr;
        cl_mem cmDevDst = clCreateBuffer(cxGPUContext, CL_MEM_WRITE_ONLY, bufferSize,
            NULL, &ciErr);
        shrLog("clCreateBuffer (%d)...\n", bufferSize);
        if (ciErr != CL_SUCCESS) {
            shrLog("Error in clCreateBuffer, Line %u in file %s !!!\n\n",
                __LINE__, __FILE__);
            throw OClException();
        }
        return cmDevDst;
    }

    void readBuffer(cl_mem buffer, void* results, size_t bufferSize) {
        cl_int ciErr;
        cl_event event;

        shrLog("clEnqueueReadBuffer...\n\n");
        ciErr = clEnqueueReadBuffer(cqCommandQueue, buffer, CL_TRUE, 0, bufferSize,
            results, 0, NULL, &event);
        clWaitForEvents(1, &event);

        if (ciErr != CL_SUCCESS) {
            shrLog("Error in clEnqueueReadBuffer, Line %u in file %s !!!\n\n",
                __LINE__, __FILE__);
            throw OClException();
        }
        clReleaseEvent(event);
    }
};

int main(int argc, char **argv) {
    shrQAStart(argc, argv);

    int FFT_DIMENSION = 64;
    const size_t szLocalWorkSize = FFT_DIMENSION;
    const size_t szGlobalWorkSize = 1000000;
    unsigned long seed = time(0);


    std::cout << "Calculating FFT with " << szGlobalWorkSize << " iterations." <<
        std::endl;
    cl_mem cmResults;
    cl_float* hostResults = (cl_float *) malloc(sizeof (cl_float) *  szGlobalWorkSize *
        FFT_DIMENSION * 2);

    boost::timer::cpu_timer timer;
    try {
        OpenClManager oclManager;
        oclManager.createKernel("FFTCalculator.cl", "FFTCalculator");
        cmResults = oclManager.createBuffer(sizeof (cl_float) * szGlobalWorkSize *
            FFT_DIMENSION * 2);
        oclManager.setKernelArg(0, (void *)&cmResults, sizeof (cl_mem));
        oclManager.setKernelArg(1, NULL, sizeof (cl_float) * FFT_DIMENSION * 3);
        oclManager.setKernelArg(2, (void *) &FFT_DIMENSION, sizeof (int));
        oclManager.setKernelArg(3, (void *) &seed, sizeof (unsigned long));
        for(int i=0; i <= 1000000 / szGlobalWorkSize; i++) {
                oclManager.launchKernel(&szGlobalWorkSize, &szLocalWorkSize);
```

```
        }
        oclManager.readBuffer(cmResults, hostResults,
            sizeof (cl_float) * szGlobalWorkSize * FFT_DIMENSION * 2);

        boost::timer::cpu_times const elapsed_times(timer.elapsed());
        boost::timer::nanosecond_type const elapsed(elapsed_times.wall);

        shrLog("Application run time: %dms\n", elapsed / 1000000LL);

    } catch (OClException &ex) {
        // finalize logs and leave
        shrQAFinishExit(argc, (const char **) argv, QA_FAILED);
    }

    if (cmResults) {
        clReleaseMemObject(cmResults);
    }

    if (hostResults) {
        free(hostResults);
    }

    // finalize logs and leave
    shrQAFinishExit(argc, (const char **) argv, QA_PASSED);
}
```

FFTCalculator.cl

```
__kernel void FFTCalculator(__global float* results, __local float* workspace, int
fftDimension, unsigned long kernelSeed) {
    if(get_global_id(0) > get_global_size(0)) {
        return;
    }

    if(get_local_id(0) == 0 ) {
        unsigned long _seed = ((get_global_id(0) * kernelSeed) ^ 0x5DEECE66DL) &
            ((1L << 48) - 1);
        for(int _i = 0; _i < fftDimension; _i++) {
            _seed = (_seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
            workspace[_i] = (_seed >> (24)) / (float) (1 << 24);
        }
    }

    barrier(CLK_LOCAL_MEM_FENCE);

    float _realOutput = 0.0;
    float _imagOutput = 0.0;
    for(int _j = 0; _j < fftDimension; _j++) {
        _realOutput += workspace[_j] * cos((float)(2.0 * M_PI * get_local_id(0) *
            _j / fftDimension));
        _imagOutput += workspace[_j] * sin((float)(2.0 * M_PI * get_local_id(0) *
            _j / fftDimension));
    }
    workspace[get_local_id(0) + (fftDimension)] = _realOutput;
    workspace[get_local_id(0) + (fftDimension * 2)] = _imagOutput;

    barrier(CLK_LOCAL_MEM_FENCE);

    if(get_local_id(0) == 0 ) {
        event_t copyEvent = async_work_group_copy(
            &results[(fftDimension * 2) * (get_global_id(0) / get_local_size(0))],
            &workspace[fftDimension], fftDimension * 2, (event_t) 0);
        wait_group_events(1, &copyEvent);
    }
}
```