# Applying Source Level Auto-Vectorization to Aparapi Java

By

Frank Curtis Albert

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Binoy Ravindran, Chair
Robert P. Broadwater
Chao Wang

May 6, 2014
Blacksburg, Virginia

Keywords: Auto-Vectorization, Aparapi, Java, GPGPU Computing, SIMD, Parallelism, Threaded

# Applying Auto-Vectorization to Aparapi Java Source Code

Frank Curtis Albert

# Abstract

Ever since chip manufacturers hit the power wall preventing them from increasing processor clock speed, there has been an increased push towards parallelism for performance improvements. This parallelism comes in the form of both data parallel single instruction multiple data (SIMD) instructions, as well as parallel compute cores in both central processing units (CPUs) and graphics processing units (GPUs). While these hardware enhancements offer potential performance enhancements, programs must be re-written to take advantage of them in order to see any performance improvement

Some lower level languages that compile directly to machine code already take advantage of the data parallel SIMD instructions, but often higher level interpreted languages do not. Java, one of the most popular programming languages in the world, still does not include support for these SIMD instructions. In this thesis, we present a vector library that implements all of the major SIMD instructions in functions that are accessible to Java through JNI function calls. This brings the benefits of general purpose SIMD functionality to Java.

This thesis also works with the data parallel Aparapi Java extension to bring these SIMD performance improvements to programmers who use the extension without any additional effort on their part. Aparapi already provides programmers with an API that allows programmers to declare certain sections of their code parallel. These parallel sections are then run on OpenCL capable hardware with a fallback path in the Java thread pool to ensure code reliability. This work takes advantage of the knowledge of independence of the parallel sections of code to automatically modify the Java thread pool fallback path to include the vectorization library through the use of an auto-vectorization tool created for this work. When the code is not vectorizable the auto-vectorizer tool is still able to offer performance improvements over the default fallback path through an improved looped implementation that executes the same code but with less overhead.

Experiments conducted by this work illustrate that for all 10 benchmarks tested the auto-vectorization tool was able to produce an implementation that was able to beat the default Aparapi fallback path. In addition it was found that this improved fallback path even outperformed the GPU implementation for several of the benchmarks tested.

# Dedication

I dedicate this thesis to my friends and family.

*Without their support, I would not have been able to complete this work.*

# Acknowledgements

I would like to thank my advisor, Dr. Binoy Ravindran, for his encouragement along the way. There were several times I was ready to give up and move on to a different project, but every time I went into his office to tell him this, I walked out re-convinced I was on the right path and excited to continue pursuing this work.

I would like to thank Dr. Robert Broadwater and Dr. Chao Wang for serving on my committee. I would also like to thank Dr. Alastair Murray for his assistance and guidance throughout this work. I would also like to thank Robert Lyerly. Both he and Alastair provided invaluable assistance debugging initial problems as well as providing a sounding board for me to bounce ideas off of.

Finally I would like to thank my friends and family for their love and support that has made this thesis possible.

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

As individual cores on a processor have hit their clock speed limits, chip makers and programmers have had to look elsewhere to find performance enhancements. One of those avenues is data parallel programming. These parallel enhancements allow programmers to program across multiple cores that can each perform different operations at the same time as well as take advantage of single-instruction multiple-data (SIMD) or vector instructions that can perform the same operation on several operands at the same time. The push for data parallelism has even gone as far as to take advantage of the thousands of simple processing cores on a graphics processing unit (GPU) to enable performance enhancements through massive parallel execution.

## 1.1 Language Support for Data Parallel Computations

These data parallel optimizations are not without their costs. In order to be able to take advantage of multiple processing cores in a CPU, a programmer must create code that can be split up across the multiple cores in order to fully harness the potential of multi-core processors. Language APIs have made this relatively simple for new code. Almost all of the common programming languages have some sort of API for creating threads that can be run on different CPU cores at the same time. While this enables new code to take advantage of these potential speedups it is of little use to older sequential code without it being rewritten with the APIs in order for it to be able to take advantage of the parallelism provided by newer processors.

Harnessing the parallel power of a GPU can be even more difficult. There are low level programming languages such as CUDA and OpenCL that enable a programmer to offload parallel calculations to a GPU, but these require complex programming techniques that most programmers

deem too difficult to warrant their use for their needs and therefore these languages are often only utilized in the realm of high performance computing. There has been recent work abstracting away the complexity of general purpose GPU (GPGPU) programming for the "average programmer" through the use of high level APIs. This work touches on two of them written for Java: Rootbeer and Aparapi.

SIMD performance enhancements have been easier for most programmers to take advantage of. Many compilers have adapted SIMD instructions in their normal optimization process. Low level compilers like C and C++ already utilize them as part of their standard optimizations and higher level JIT languages are beginning to add them as well. Just recently Microsoft announced it has begun work on adding SIMD instructions to its C# JIT [1].

There have also been requests for SIMD functionality in Java [2], and while there has been no published documentation for SIMD functionality in Java, a careful analysis of JIT output yields that it may be in the works. For very basic loop cases the Hotspot JIT compiler in Oracle Java 7u55 will use SIMD instructions and OpenJDK did not use SIMD instructions at all. However, even Oracle's Hotspot JIT does a very poor job and most vectorizable code is never vectorized. Even when specific functions were created to isolate the vectorizable code into something that the JIT should be able to easily recognize as vectorizable, there was no discernable performance gain.

## 1.2 Thesis Contributions

This work brings the data parallel performance enhancements of SIMD instructions to Java through a simple vector library. As Java's JIT compiler cannot reliably take advantage of the performance enhancements of SIMD instructions, this work presents a C++ library that is accessed by Java through JNI calls to perform SIMD calculations. This library can be either be utilized directly by the programmer in vectorizable regions or can be automatically added through an auto-vectorizer tool also presented in this work.

To handle the automatic utilization of the vector library, this work presents an auto-vectorizer that works with the high level Java extension Aparapi. Aparapi brings the performance enhancement potential of the massively parallel GPU hardware to the average programmer without forcing them to learn or write any low level code The Aparapi extension provides the programmer with a high level API that enables them to offload data parallel calculations to an OpenCL device. By using the Aparapi API, the programmer has already stated that the calculations are independent of each other in stating that they can run in parallel on a GPU. The auto-vectorizer takes advantage of this knowledge of independence to help it prove sections of the code vectorizable and correctly utilize the vector library without any additional effort from the programmer

Another contribution of this work is to enhance the abilities of the selected Java extension: Aparapi. Aparapi provides a fallback path to ensure that even if the code fails to run on a GPU, Aparapi will transfer execution back to the CPU and the program will execute in the Java thread pool (JTP). However, this fallback path is not without its faults. The default fallback path is to run

the created kernels in the Java thread pool, but as a CPU cannot handle as many parallel calculations as a GPU, this creates additional overhead by creating more kernels than necessary to perform the calculation. This work seeks to improve on this fallback path by providing two better options. The first is a vectorized implementation created by the auto-vectorizer tool utilizing the vector library. The other is a looped implementation that combines kernels to enable each new kernel to loop through its calculations performing the work of multiple kernels without incurring the extra overhead.

Analysis of the results of several benchmark applications has even revealed that there are several cases where the vectorized implementation even beats the GPU implementation. In cases with execution times on the order of seconds or cases requiring large amounts of data transfer to and from the GPU, this work provides the fastest implementation of the benchmark through the auto-vectorized implementation.

## 1.3 Thesis Organization

The remainder of this thesis is organized as follows:

- Chapter 2 provides a brief history and overview of the current state of the art approaches to general purpose graphics processing unit (GPGPU) computing.

- Chapter 3 compares two Java extensions, Rootbeer and Aparapi, that bring GPGPU capabilities to Java through an easy to use API.

- Chapter 4 describes the vector library presented by this work to bring SIMD functionality and speedups to the Java programming language.

- Chapter 5 describes the auto-vectorization tool also presented by this work to implement the vector library into existing Aparapi Java programs with no added effort for the developer.

- Chapter 6 presents a thorough performance analysis of the vector library in several benchmarks to highlight the speedups made possible through the SIMD extensions.

- Chapter 7 presents the overall conclusions of this work.

- Chapter 8 offers suggestions for future work.

# Chapter 2

# Literature Review

In this chapter, we survey the state of the art approach to high performance programming. In particular we focus on the efficiency of Java vs C++ as well as the growing trend of increased performance through parallelization. This chapter discusses the rise in popularity of general purpose GPU (GPGPU) computing as well as its state of the art programming approaches.

## 2.1 Java vs C++ Performance

Java often has a bad reputation for being a poor performer in the field of high performance computing. Java started out as a high level language that was designed to abstract the programmer almost completely away from the machine. While this supports the "compile once, run anywhere" model of the Java Virtual Machine (JVM), this layer of abstraction does not always result in code that is well suited for computationally intensive tasks. Traditionally, these tasks are better left for lower level languages like C and C++ or Fortran that compile to machine code for a specific processor and therefore can take advantage of the specific architecture. However, with the advancement of Just-In-Time (JIT) compiler technology, adaptive optimization, and improved garbage collection, Java has closed the performance gap with lower level languages like C and C++ [3], [4], [5], [6], [7].

The JIT plays a major role in increasing the performance of a Java application. It takes Bytecode that it deems beneficial to compile and compiles it down to native code during runtime. This can result in dramatic performance increases for code that is run multiple times [5]. Compiling at runtime also gives Java an advantage in that it can collect data on how the code is running at runtime and use this data to reconfigure and recompile sections of the code to increase performance.

Another factor that Java does not have to deal with when optimizing is pointers. In an example like the code below in Figure 1, the C compiler cannot guarantee that p is not pointing to

x and therefore cannot keep $x$ in a register. This is the same for $arr[j]$ since in C and C++ arrays act like pointers [3]. However, Java's JIT can learn this at runtime and adapt accordingly.

```
x = y + 2 * (...)
*p = ...
arr[j] = ...
z = x + ...
```

*Figure 1: C Pointer Code [3]*

As seen below in Table 1 and Figure 2, all of these optimizations combine to allow Java to have comparable and sometimes better performance in computational benchmarks when compared to C++. In the 12 benchmarks run below in Figure 2, C++ and Java spilt the benchmarks at 6 apiece and Java even came out ahead on the average [6]. High performance computing applications written in Java have even won the GraySort benchmarking contest. In 2009 Hadoop broke the data-sorting world record when it correctly sorted 100 TB of data at a rate of 0.578 TB/min and an improved version sorted 1 PB of data at 1.03 TB/min [8].

Java is no longer left out of consideration when developing high performance applications. It has closed the performance gap with lower level languages like C, C++, and Fortran, and has many benefits like its portability and memory abstraction. Java handles all memory allocation and freeing for the programmer, which can often cause difficult to debug problems in C and C++ applications if the programmer is not careful. Companies are now willing to overlook any performance penalty associated with running their code in Java due to its portability and ease of use.

| | | Pentium III, NT | Pentium III, Linux | Sun UltraSparc II | Compaq ES40 |
|---|---|---|---|---|---|
| C | FFT | 2.26 | 1.49 | 2.07 | 2.15 |
| | HeapSort | 1.00 | 0.95 | 1.29 | 2.48 |
| | SOR | 1.17 | 1.17 | 1.40 | 2.45 |
| | LUFact | 1.17 | 0.93 | 1.34 | 2.84 |
| | Series | 0.87 | 0.87 | 1.54 | 8.29 |
| | Sparse | 1.30 | 1.07 | 2.61 | 2.64 |
| | Euler | 1.48 | 1.41 | 1.94 | 17.6 |
| | MolDyn | 1.09 | 1.12 | 1.49 | 2.84 |
| | Mean | 1.23 | 1.07 | 1.61 | 4.08 |
| Fortran | LUFact | 1.18 | 1.16 | 1.48 | 5.07 |
| | MolDyn | 1.27 | 2.27 | 1.39 | 2.38 |
| | Mean | 1.22 | 1.62 | 1.43 | 3.47 |

*Table 1: Ratios of fastest Java execution times to fastest C/Fortran execution time. [7]*

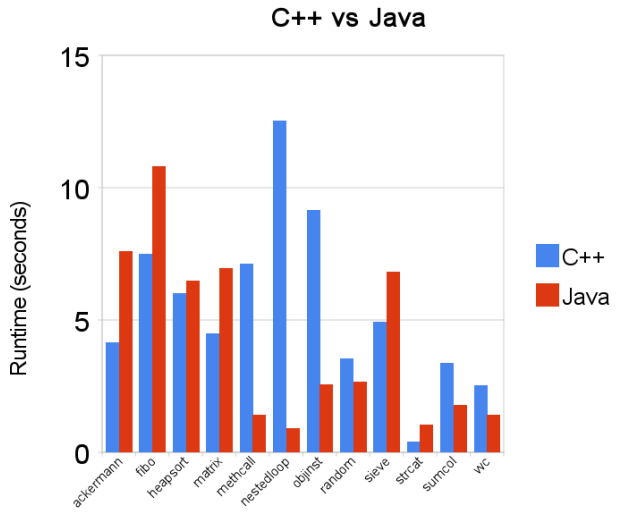


*Figure 2: C++ vs Java Performance Benchmarks [6]*

# 2.2 Data Parallel Programming

In the quest to keep improving processor speed, chip makers have had to get creative recently. It used to be that to make a processor faster, a manufacturer could simply increase the clock speed of the chip. With this kind of improvement code compiled for one machine could be moved to a new machine and be faster just by running on the newer, faster processor. However, around 2002 – 2003, chip manufacturers hit the power wall as illustrated by the clock speed and power curves in Figure 3 below. Chips could not be clocked any faster or they would become too hot and start melting components.

Other techniques that manufacturers used to get extra performance out of each clock cycle such as pipelining and instruction level parallelism also hit a relative peak around the same time, illustrated by the leveling off of the Perf/Clock (ILP) curve in Figure 3 below. The only curve below that hasn't hit its peak is the number of transistors that manufacturers are able to fit onto a chip. As manufacturing processes improve, manufacturers are able to shrink the size of transistors and fit more of them into the same space. This allowed manufactures to start putting multiple cores into a processor as well as add data parallel compute units.

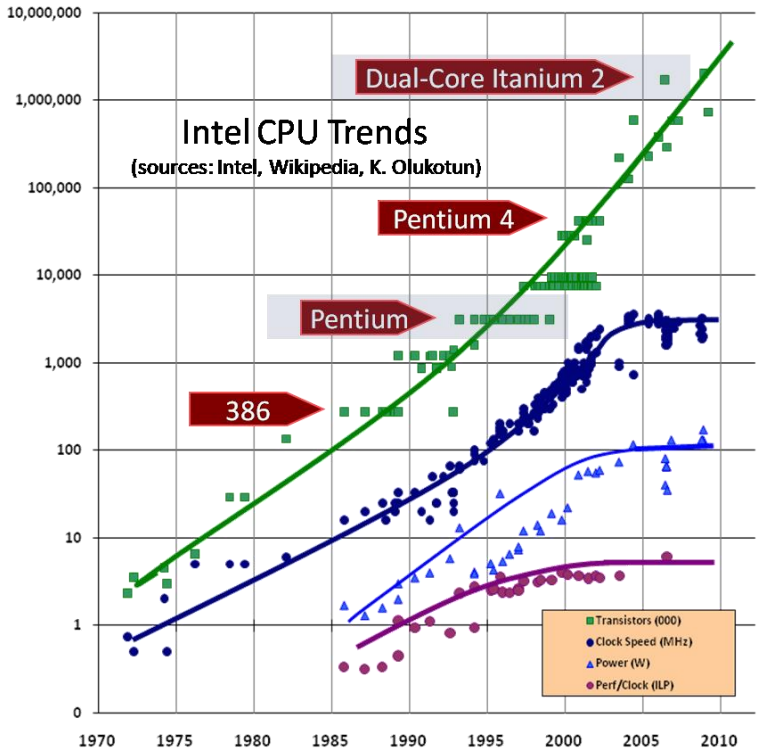


*Figure 3: Intel CPU Trends [9]*

The down side associated with data parallel compute units and multi-core processors is that while the chips can theoretically handle more computations than a single core processor, software must be modified to take advantage of it. The same program discussed above that used to run faster by just running on a new machine now no longer sees the same performance boost by running on a newer processor. Since none of the individual cores in a multi core processor are faster than preceding single core chips and the new data parallel compute units must be accessed through new SIMD instructions, the program will not run faster and may even run slower. Programs must be modified or rewritten to take advantage of this parallelism if they want to harness the full processing potential of a multi-core processor.

# 2.3 Data Parallel Compute Units

Data parallel compute accessed through Intel's SSE and AVX instructions utilize a single instruction multiple data (SIMD) approach to achieve speedups. These instructions allow multiple computations to be performed through a single instruction to increase computational throughput. Many languages that compile to machine code like C, C++ and Fortran already have support for SIMD instructions through their compiler optimizations. This allows programmers to utilize these performance enhancing instructions with no added effort or modification to their program.

There have been two major attempts to bring this functionality to Java. The first [10], only utilizes pseudo-SIMD functionality. It does not use SIMD instructions like those in the SSE or AVX instruction set, but instead uses standard integer operations on short and byte data types to achieve SIMD functionality. This work does present a solution that is integrated into the JVM and can achieve speedups of 1.6x when combining 2 short data types, and 2.5x speedup when combining 4 byte data types. However, this work has many limiting factors. First it only works when both operands are vectors of type short or byte. Second it only works on single operand loops that have over $10^7$ iterations and the operand is an either *and*, *or*, *add*, or *subtract*. These limitations drastically reduce the practically of this work.

The second work bringing vectorizations to Java [11] claims to avoid these limitations. It utilizes an automatic vectorizer that it builds into the Harmony JVM's JIT compiler Jitrino. The auto-vectorizer claims to be able to vectorize Java loops of any size unlike the previous work's [10] single instruction loop limitation. The work also supports a much larger selection of SSE and AVX instructions to bring true vectorization to Java. However, the results of this work leave much room for improvement. The work only sees speedups on 2 of the sub-benchmarks of SPECjvm2008 and even these speedups are nowhere near the speedups possible through the use of SIMD instructions. The work claims a maximum 1.55x and 2.07x speedup on a desktop implementation and a maximum 1.45x and 2.0x speedup on a server implementation, but these speedups reduce to practically nothing as the number of threads and data input size increases. This is far from ideal as a multi-threaded approach can often provide a larger performance gain than a single thread SIMD approach. The vectorization implementation in this work seeks to improve on the performance numbers in order to achieve speedups much closer to the theoretical 4x and 8x provided by SIMD instructions.

## 2.4 General Purpose Graphics Processing Unit Computing

The multi-core trend has been taken even further when it comes to graphics cards. Instead of having a handful of cores like multi-core CPUs, graphics cards can have thousands of cores. The Nvidia Tesla K40 has 2880 CUDA cores that can be utilized to process extremely parallel data [10]. However it can be difficult to fully utilize GPUs for general purpose programming.

Graphics Processing Units (GPUs) were originally designed as simple graphics accelerators and only supported a small set of specific fixed-function pipelines. In the late 1990s the programmability of this hardware began to grow. Nvidia released its first GPU in 1999 and it wasn't long after that that researchers began to harness the potential of GPUs for scientific computing. However these original scientific computations were limited to problems that could be mapped to triangles and polygons to fit the narrow processing capabilities of a GPU [11].

In 2003 the brook compiler looked to change these limitations. Created by a group of researchers at Stanford University lead by Ian Buck, the brook compiler became the first widely

used programming model extension for C to simplify general purpose GPU computing. It that used streams, kernels, and reduction operations instead of triangles and polygons to reduce the effort required to program a scientific calculation onto a GPU. [12] Buck would later join Nvidia to help them release CUDA in 2006 [11].

Ever since then, General Purpose Graphics Processing Unit (GPGPU) computing has been a growing trend in high performance computing. Graphics cards are no longer just used to render pixels or power high quality video games. These many core architectures are now being exploited to perform massively parallel independent computations. The United States' fastest supercomputer, Titan, takes advantage of 18688 NVIDIA tesla K20 GPUs to enable up to 20 plus petaflops of computational power [11].

### 2.4.1 CUDA and OpenCL

In order to enable computational programming on GPUs two different programming models have been created. NVIDIA created and maintains CUDA (Compute Unified Device Architecture) [12], while OpenCL is an open standard that was developed by Apple and is maintained by the non-profit Khronos Group [13]. Both of these implementations are similar in that they provide programming models to offload computations to a GPU. However, CUDA only works on NVIDIA GPUs, while OpenCL can run across a variety of heterogeneous hardware with support from vendors such as Apple, Intel, Qualcomm, AMD, Nvidia, Altera, Samsung, Vivante, and ARM Holdings.

Programming in CUDA C involves defining specific parallel functions called kernels in a C program. Each of these kernels is designed to be run across multiple GPU cores in parallel when called. In order to call these functions a few steps must be taken in the host-side code. The host-side code must

1. allocate memory on the host device for any variables on the host side,
2. allocate memory on the device for any variables on the device,
3. copy memory from host memory to device memory,
4. invoke the kernel,
5. copy the result back from device memory to host memory,
6. free all device memory,
7. and finally free all host memory.

Once CUDA code is written, it is compiled with the Nvidia CUDA Compiler (NVCC). The NVCC compiles the kernel sections into assembly form (PTX code) and/or binary form (cubin object) as well as modifying the host code to properly interact with the newly compiled kernels. Any code that is left in the PTX form must be Just-in-Time compiled at runtime. At runtime the program passes the PTX code to the appropriate device driver which then compiles down to its own native binaries. This binary is then saved for any future runs of this code. While Just-in-Time compilation does add additional overhead at runtime, it allows for one binary to work across

multiple GPUs as well as allowing the program to benefit from any new compiler improvements without having to recompile every program each time a new device driver comes out [14].

Programming OpenCL involves a similar process of writing computation kernels that are to execute on the GPU. These compute kernels can be written in a subset of C that does not include function pointers, recursion, bit fields, variable-length arrays, and standard C99 header files. The source code consists of host-side code that must

1. find and select an available OpenCL device,
2. create an OpenCL context to manage objects,
3. create a command queue to control the device,
4. create a memory object to transfer data to and from device,
5. read an OpenCL kernel file,
6. create a program object,
7. compile the OpenCL code down to a binary format specific to the runtime device,
8. create a kernel object for each kernel function,
9. set each kernel's arguments,
10. add any kernels to be executed to the task queue,
11. read any results from the memory buffer,
12. and finally free all of the objects created.

Just like CUDA code, OpenCL kernels can be either precompiled or compiled at runtime. In the precompilation case, the kernel is compiled when the rest of the source code is compiled. This eliminates any overhead for compiling the OpenCL kernel at runtime, but comes with the caveat that every OpenCL device that the code might run on must be listed at compile-time and compiled to. This increases the size of the binary as well as limits the portability of the OpenCL kernel.

In the online (compile at runtime) case, the host code reads in the kernel source file and Just-in-Time compiles the kernel source code for the specific device. This adds increased portability as the same compiled host code will work with any device with a valid OpenCL driver. Online compilation also has a few drawbacks. JIT compilation does take time and may not be suited for embedded or short running applications. Also any code distribution will have the OpenCL kernel in its source code form which may not be acceptable for commercial applications [14].

## 2.4.2 Low Level GPGPU Code Complexity

The current problem with graphics card processing is that in order to harness the power of a GPU a programmer must use a low-level programming language such as CUDA or OpenCL. This may be fine for programmers trained in the field of high performance computing, but the average programmer is going to be quickly deterred by the added coding and debugging complexity that comes with CUDA or OpenCL. Section 2.2.1 above lists the numerous steps that

a programmer must follow that quickly transforms even a simple program into a much more complicated program that has several more places for errors. Figure 4 below shows the code required to perform a sequential array addition in Java. It is only six lines long and any beginner programmer can read, write and understand it. The same code written in CUDA to run on a graphics card, Figure 7, is over 30 lines long and requires complex memory allocations and transfers that most application programmers do not wish to deal with.

Over the last few years, several groups have emerged to try and reduce the coding complexity required for GPGPU programming and bring GPGPU speedups to Java. Figure 5 shows that with only a few added lines, the sequential Java array add can be converted into parallel code that can run on any OpenCL device, or in the Java thread pool itself. There are also plans to simplify this even further through Java 8's lambda extensions enabling the same code to be run with very minimal additions to standard Java [15]. These high level approaches are discussed further in Chapter 3.

```
final float inA[] = ...
final float inB[] = ...
final float result = new float[inA.length];
for (int i=0; i<array.length; i++){
   result[i]=intA[i]+inB[i];
}
```

*Figure 4: Java Array Add*

```
final float inA[] = ...
final float inB[] = ...
final float result = new float[inA.length];
Kernel kernel = new Kernel(){\
  @Override public void run(){
    int i= getGlobalId();
    result[i]=intA[i]+inB[i];
  }
};
Range range = Range.create(result.length);
kernel.execute(range);
```

*Figure 5: Aparapi Array Add*

```
final float inA[] = ...
final float inB[] = ...
final float result = new float[inA.length];
Device.getBest().forEach(result.length, id -> result[id] = intA[id]+inB[id]);
```

*Figure 6: Aparapi Array Add in Java 8*

```
int main()
{
int *a, *b, *c_cpu, *c_gpu;
int *d_a, *d_b, *d_c;
int size = N * N * sizeof (int); // Number of bytes of an N x N matrix
// Allocate host memory
a = (int *) malloc (size);
b = (int *) malloc (size);
c_gpu = (int *) malloc (size);
// Initialize host memory
...
// Allocate GPU/device memory
cudaMalloc( &d_a, size );
cudaMalloc( &d_b, size );
cudaMalloc( &d_c, size );
// Copy the initialized data to the GPU
cudaMemcpy( d_a, a, size, cudaMemcpyHostToDevice );
cudaMemcpy( d_b, b, size, cudaMemcpyHostToDevice );

dim3 threads_per_block (16, 16, 1); // A 16 x 16 block threads
dim3 number_of_blocks ((N / threads_per_block.x) + 1, (N /
threads_per_block.y) + 1, 1);
vecAddGPU <<< number_of_blocks, threads_per_block >>> ( d_a, d_b,
d_c );
cudaMemcpy( c_gpu, d_c, size, cudaMemcpyDeviceToHost );
// Free all our allocated memory
free( a ); free( b ); free( c_cpu ); free( c_gpu );
cudaFree( d_a ); cudaFree( d_b ); cudaFree( d_c );
}
__global__ void vecAddGPU( int * a, int * b, int * c )
{
   int val = 0;
   int row = blockIdx.x * blockDim.x + threadIdx.x;
   int col = blockIdx.y * blockDim.y + threadIdx.y;
   if (row < N && col < N)
   {
      c[row * N + col]  = a[row * N + k] + b[k * N + col];
    }
}
```

*Figure 7: CUDA Array Add*

### 2.4.3 High Level GPGPU Languages

As described in section 2.2.2 above, programming GPUs for general purpose programming often involves complex low level data transfers and code complexity that average programmers do not wish to deal with. In order to bring general purpose GPU computing to the masses extensions to current languages and even entirely new domain specific languages and have been created.

One of the promising new languages is Liquid Metal or (LIME) developed by IBM. Liquid Metal seeks to create one unified programming language that will allow programmers to program across CPUs, GPUs, and FPGAs. Liquid Metal is an extension to Java that can be run within the Eclipse compiler. It allows for the programmer to prototyped the program in Java and then add LIME constructs to express types and invariants allowing the compiler to generate Verilog for selected sections of code. The programmer can than execute the program in the JVM and a Verilog simulator to analyze performance and tune the program before finally synthesizing to an FPGA [18]. However, at the time this work began LIME was not available to the public and even now it is only available in alpha form that has basic support for FPGAs but no support for GPUs. Because of these factors it was not considered for this work [19].

Another domain specific language designed to bring GPGPU computing to the masses is Chestnut. Chestnut's goal was to provide the simplest programming language possible to the programmer that would enable processing across CPUs and GPUs. To do this Chestnut does not build upon any existing languages, but instead creates their own new language. The Chestnut compiler then performs source to source code translation to convert the Chestnut code into CUDA-C code that can be then compiled and run on a machine [20]. Because this work created an entirely new language that programmers must learn and transfer existing programs to, Chestnut was not further pursued for this work.

Harlan is another recent declarative domain specific language that provides a programmer high level access to the GPU. Harlan, like the others, seeks to abstract away the data movement from the programmer allowing the programmer to worry about the larger goal of their code and allowing the compiler to optimize data transfers. Harlan's compiler also detects when sections of code are independent of GPU code and allows them to run on the CPU while it is waiting for the GPU to finish the parallel calculations. Again as this was a new domain specific language it was not further utilized in this work.

The two most promising high level Java extensions that easily bring GPGPU computing to the average programmer, Rootbeer and Aparapi, are discussed in detail in chapter 3.

# Chapter 3

# Java GPGPU Computing

There are currently two significant released projects working to bring GPGPU computing to Java: Rootbeer [17] and Aparapi [18]. Both offer a simplified API that allows a programmer to achieve much of the performance advantages that low level GPGPU computing has to offer without having to deal with the complex memory allocations and transfers. This opens up the world of GPGPU computing to developers who have deemed the development cost of CUDA or OpenCL to high.

The few benchmarks below illustrate that these APIs can provide significant speedups over sequential or even threaded Java. The matrix multiply benchmark even shows that these speedups are close to the perfect hand coded OpenCL case without sacrificing the ease of programming in high level.

| Matrix Multiply (8192 x 8192) | Time (s) |
|---|---|
| Seq Java | 847.60 |
| Aparapi OpenCL on CPU | 262.66 |
| Java Thread Pool | 287.41 |
| Aparapi OpenCL on GPU | 2.6 |
| OpenCL | 1.4 |

*Table 2: Matrix Multiply Times Across Several Implementations*

| MandelBrot | miliSec/Frame |
|---|---|
| Sequential | 71.43 |
| Aparapi OpenCL on CPU | 20.00 |
| Java Thread Pool | 19.61 |
| Aparapi OpenCL on GPU | 8.13 |

*Table 3: MandelBrot Times Across Several Implementations*

| Life | miliSec/Gen |
|---|---|
| Aparapi OpenCL on CPU | 6.71 |
| Java Thread Pool | 11.46 |
| Aparapi OpenCL on GPU | 4.08 |

*Table 4: Life Times Across Several Implementations*

# 3.1 Rootbeer

Rootbeer is an API developed at Syracuse University by Dr. Pratt-Szeliga. With the Rootbeer API a programmer creates sections of Java code that can run on a GPU without having to worry about the serialization and deserialization of data, memory transfers, or kernel creation or kernel launch that are required for a CUDA or OpenCL program. The programmer simply creates a new class that implements a predefined Kernel interface. Inside the new class the programmer can create any methods that are to run on the GPU. The only main Java feature that Rootbeer does not support is dynamic method invocation. Rootbeer handles all of the low level details turning the Java kernel class into corresponding CUDA code including automatically finding any reachable variables or objects inside the kernel and copying these variables and objects to the GPU before compilation and back again after compilation.

Figure 8 below shows an example of an ArraySum kernel class that performs an array summation on a GPU through the Rootbeer API. In this example multiple kernels are created on the GPU to perform the task of summing several arrays. Each kernel is passed a different array to sum as the first argument, *src.* The second argument passed is a shared array that all kernels will place their resulting sum in and the third argument is the index value of the kernel so it knows where to store its sum.

```
public class ArraySum implements Kernel {
  private int[] source;
  private int[] ret;
  private int index;
  public ArraySum (int[] src, int[] dst, int i){
    source = src; ret = dst; index = i;
  }
  public void gpuMethod(){
    int sum = 0;
    for(int i = 0; i<array.length; ++i){
      sum += array[i];
    }
    ret[index] = sum;
  }
}
```

*Figure 8: Rootbeer ArraySum Kernel Class [18]*

Once the kernel class has been created, it can be called from other Java code with a few simple lines needed to setup and launch the kernel objects. Figure 9 below illustrates the code required to call the ArraySum Kernel created in Figure 8. First a list is created to hold all of the kernels. Then a new kernel is created for each element to be added. Finally a Rootbeer object is created to run all of the kernel objects in the list.

```
public class ArraySumApp {
public int[] void sumArrays(List<int[]> arrays){
  List<Kernel> jobs = new ArrayList<Kernel>();
  int[] ret = new int[arrays.size()];
  for(inti = 0; i<arrays.size(); ++i){
    jobs.add(new ArraySum(arrays.get(i), ret, i));
  }
  Rootbeer rootbeer = new Rootbeer();
  rootbeer.runAll(jobs);
  return ret;
  }
}
```

*Figure 9: ArraySumApp calling Rootbeer kernel to run ArrayAdd on a GPU*

The code is then compiled first by the regular javac compiler into a jar file and then passed through the Rootbeer Static Transformer to generate the final output jar that can be run like any other Java jar file [17].

### 3.1.1 Pros

After careful analysis of both Rootbeer and its competitors, Rootbeer was determined to have several positive aspects. Unlike its competitors, Rootbeer was able to handle almost all aspects of Java including

- instance and static methods and fields,
- array types of any dimension of any base type,
- composite reference types,
- arbitrary object graphs including cycles,
- synchronized static and instance methods,
- locking on an object,
- inner classes,
- dynamic memory allocation,

- strings and exceptions,
- and null pointer/ out of memory exceptions are thrown.

This enables developers to develop Rootbeer Java entirely in their existing IDE environments such as Netbeans or Eclipse. It also allows for a much easier conversion process from existing applications to the Rootbeer API.

### 3.1.2 Cons

Rootbeer, however, is not without its drawbacks. While it does allow for code to run on a GPU, it relies on the fact that a GPU with a CUDA driver will be available at runtime. If there is not a GPU with a CUDA driver the code crashes and will not execute. This is not a safe assumption as Java applications that are meant to run on a wide variety of systems and not many people have a CUDA driver installed or even have a CUDA compatible Nvidia GPU. Also since Rootbeer generates CUDA code it is rendered useless on machines with AMD GPUs as CUDA is an Nvidia proprietary product. Rootbeer also experienced a performance problem with extremely large data sets explained further in section 3.3.

## 3.2 Aparapi

Aparapi stands for "A PARallel API" and was originally developed by AMD, but was released as open source in 2011. Aparapi's API was developed as a way to express parallel sections of code in Java so that it could be either run in the Java thread pool or converted by a runtime component into OpenCL that can run on any OpenCL capable device. This extends Java's write once run anywhere goal past CPUs and on to include any OpenCL capable hardware [19].

Like Rootbeer, Aparapi is programmed through the use of a kernel. However, unlike Rootbeer, this kernel can be inline with the rest of the source code, see Figure 5 and Figure 6 in section 2.2.2. A programmer simply needs to instantiate a kernel with an overridden run method and tell that program to execute. Inside the kernel the Aparapi API provides functions like getGlobalId() that returns the overall identification number of the individual kernel and as well as other functions to allow the developer to have each kernel behave differently based on which kernel it is.

At compile time the application code is compiled the same way any other Java application would be compiled with the Aparapi jar file as an included library. Then at execution time Aparapi goes through a series of logic to determine where to run a kernel. If the programmer has specified a device for the kernel, Aparapi tries that first. If no device is selected, the default execution device is GPU. However, Aparapi first checks to see if an OpenCL device is available at run time. If there

is not, or if it detects that the OpenCL kernel has failed in its execution, Aparapi will default back to running code in the Java thread pool [18].

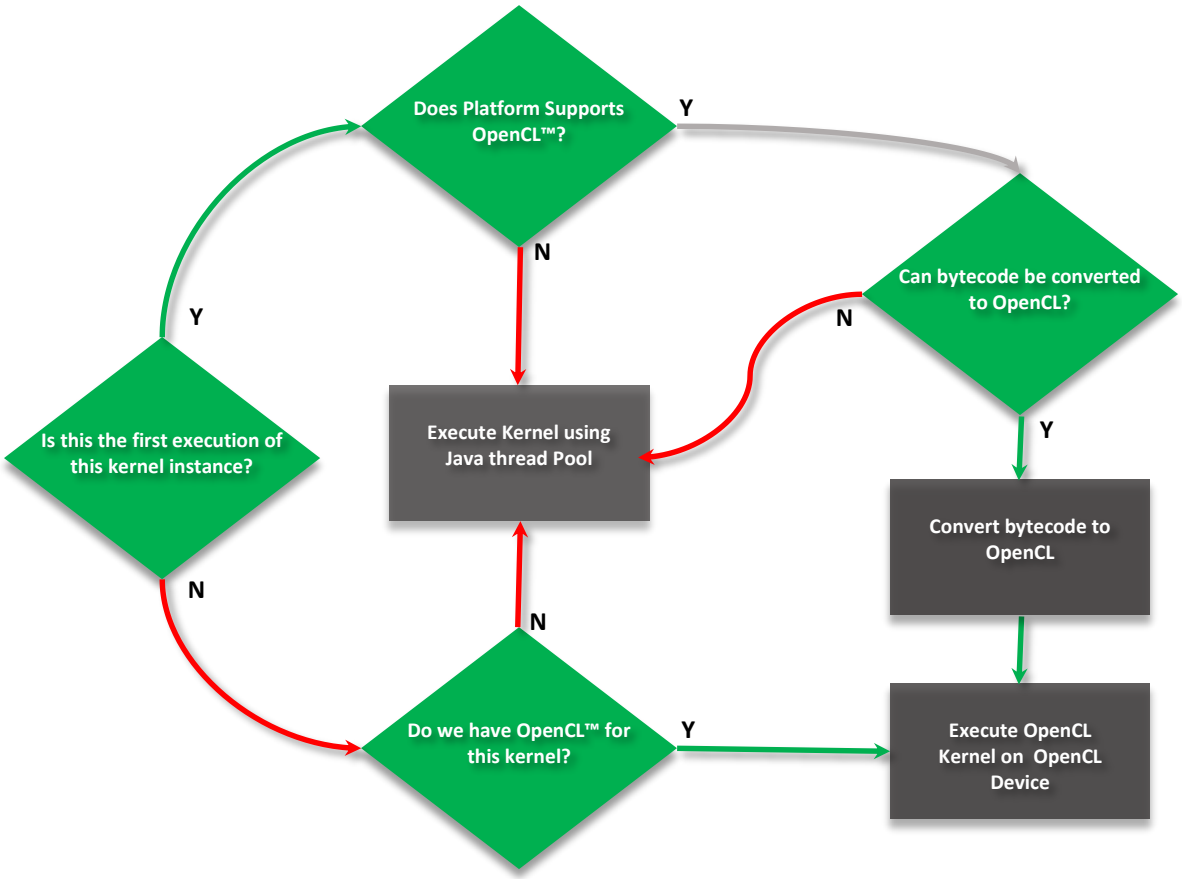


*Figure 10: Aparapi Execution Logic [18]*

## 3.2.1 Pros

Like Rootbeer, since Aparapi is an add-on to Java, a programmer can develop and compile Aparapi programs from within their favorite IDE. Aparapi's implementation of the kernel interface is also simpler that Rootbeer's in that it does not require a separate class for each kernel. Instead the code for a Kernel can be implemented right above the call to execute it.

The other major benefit of Aparapi is its fallback path. If the programmer marked a section of code to run on the GPU, but at runtime there are no OpenCL devices available, the code will throw an exception, and fallback and execute on the CPU. Similarly if the programmer does something that Aparapi cannot handle on the GPU and Aparapi detects a failure on the OpenCL

device, Aparapi will notify the programmer and fall back to the Java thread pool ensuring a proper execution.

### 3.2.2 Cons

Unlike Rootbeer, not all Java bytecode can be converted into OpenCL by Aparapi. Aparapi can only handle primitive data types and single dimensional arrays of primitive data types. This is due to the fact that Aparapi does not have a serializer like Rootbeer and therefore cannot copy the non-contiguous memory of multi-dimensional arrays or Java objects.

Aparapi is also a work in progress, and the source code downloaded from their repository had some bugs that had to be fixed. Aparapi claims to work with multi-dimensional ranges, but when their samples were run it was found that multi-dimensional ranges failed on the Java thread pool and the source code had to be changed to enable proper execution on the Java thread pool.

## 3.3 Comparison

After careful consideration, this work chose to use Aparapi. Even though Aparapi supports a smaller subset of Java. It provides an easier to use API that allowed for easier benchmark conversion as well as provide robustness through its fallback path ensuring that the benchmarks would yield correct results on any machine they were run on.

Another major contributor to the decision to choose Aparapi over Rootbeer was Rootbeer's horrible performance times as problem size grew. Newer versions of the tool may fix the problem, but at the time of the comparison test, Aparapi drastically outperformed Rootbeer for any problem size over 8192 kernels. Figure 11 below illustrates this drastic difference in performance for large problem sizes.

The third factor that contributed to the selection of Aparapi was developer support. Aparapi was developed by AMD and is still being actively worked on by a large community. Code updates were regular and forum questions were answered within hours of posting. The project has the support of a large corporation and is likely to develop into something more widely used. Rootbeer, however, is simply a research project at Syracuse University and while it is still being updated, it is not growing at the same rate as Aparapi. Rootbeer also still requires an older version of Java, Java 6, to run.

The combination of these three factors led to this work pursuing the implementation of the vectorization library with the Aparapi interface. However, since the library is not tied directly into Aparapi, hand vectorized code will still work in Rootbeer and the Auto-Vectorized could be adapted to handle the Rootbeer interface as well.

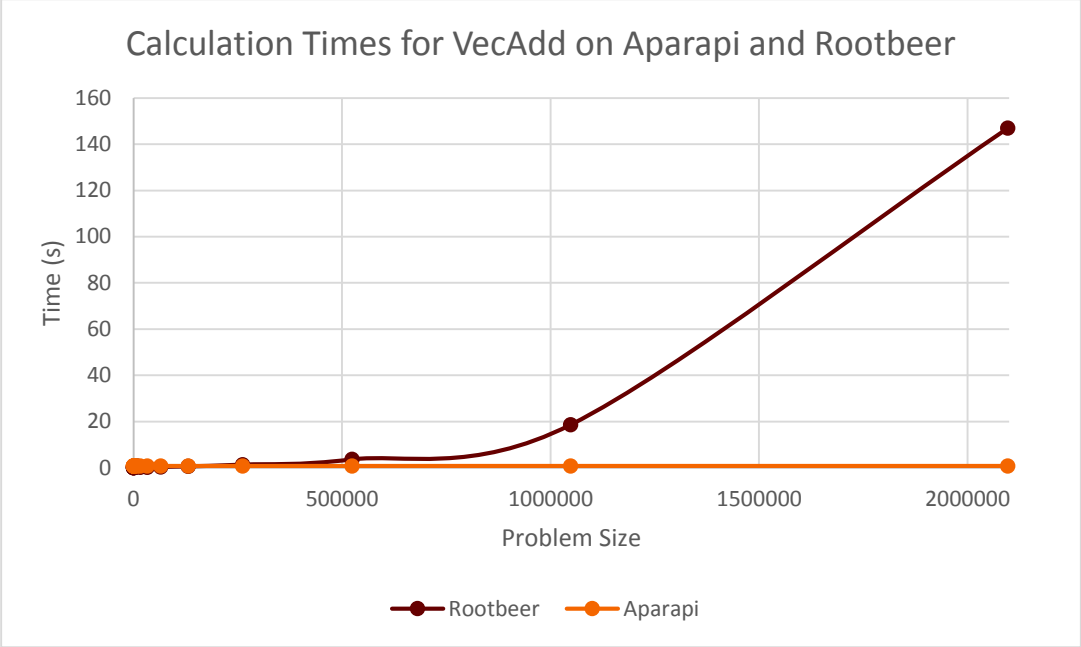


*Figure 11: Comparison of Aparapi and Rootbeer performance for VecAdd*

# 3.4 Performance Analysis Java Thread Pool vs. GPU

For parallel code sections with lots of calculations and a smaller amount of data the GPU achieved massive speedups. In all three benchmarks above (Table 2, Table 3,Table 4), the GPU was able to achieve significant speedups. For a 4096 x 4096 matrix multiply the GPU was able to achieve a 78x speedup vs the Java thread pool running on a 64 core machine (see Chapter 6).

However, just because a piece of code is parallelizable does not mean that it should necessarily run faster on the GPU. The GPU can perform a large number of calculations in parallel, but it does have its overheads. In Aparapi testing it was determined that the GPU had an initial startup of around 0.78 seconds for any Aparapi kernel in addition to the time it takes to transfer the data to and from the GPU. Aparapi kernels run in the Java thread pool only saw a 0.0027 second overhead cost from the Aparapi kernel overhead and have no data transfer cost. Even massively parallel operations like array addition are better performed on the CPU due to the low computation complexity and high data transfer costs. In testing a simple array addition, the GPU took on average 10 times as long as the vector add on the CPU due to the compile and data transfer

overheads experienced by the GPU. Rootbeer's documentation even goes as far as to suggest that only calculations with at least complexity $O(n^2)$ or $O(n^3)$ be sent to the GPU [16].

Another factor that can hurt a GPU kernel implementation's performance is its inability to handle large amounts of control flow well. In benchmarks with lots conditional statements the GPU loses its abilities to perform parallel calculations and incurs a slowdown when compared to the Java thread pool (JTP). In a GPU branch conditions are handled on a per warp basis and when there is a branch divergence within a warp, both code paths are executed sequentially [20]. This causes the GPU to lose its parallel advantage and drastically hinders performance.

The JTP implementation can run into overhead problems as well. While Aparapi's JTP kernel overhead is small, it can add up when thousands of kernels are created. Since most modern CPUs can far fewer threads at a time than get created, this work also implements a looped kernel implementation to try and reduce the kernel overhead. Instead of creating a new kernel for each kernel instance, the looped implementation creates a reduced number of kernels and has each of them perform the calculations for multiple kernel instances and therefore performing the same calculations without having to undergo the additional kernel overhead.

# Chapter 4

# Vectorization

Vector instructions enable a processor to perform element wise operations on two multi-element operands either stored in memory or specific vector registers that can be loaded from memory with a single instruction. Therefore from one load, execute, and store multiple calculations can be performed. For example, Figure 12 below illustrates that while a normal add operation can only compute the result of one pair of operands, a vector instruction can compute the sum of multiple, in this case four, pairs of operands in a single operation. While no individual calculation is sped up, these instructions dramatically increasing performance by increasing throughput and simplifying the process of issuing enough operations to keep the floating point pipelines full [13].

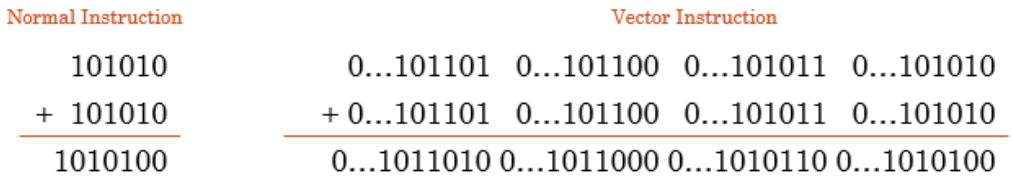


*Figure 12: SIMD Example*

## 4.1 Vector Library

Java is still at the early stages of supporting SIMD instructions. The only documentation claiming Java supports SIMD instructions is in bug reports [2], and in practice the Hotspot JIT compiler never vectorized anything beyond a simple test program with a single sequential loop containing an array addition. Anytime threads were involved or the loop became more complicated the Hotspot JIT compiled to non SIMD instructions. Because of this, a special vector library was needed in order to enable SIMD operations in Java.

```
…
0x00007f90a9065560:   vmovdqu   0x10(%r11,%rbx,4),%xmm0
0x00007f90a9065567:   vaddps    0x10(%r8,%rbx,4),%xmm0,%xmm0
0x00007f90a906556e:   vmovdqu   %xmm0,0x10(%rbp,%rbx,4)
…
```

*Figure 13: Java JIT Assembly with SSE instructions*

```
…
0x00007f29d9062c13:   vmovss    0x10(%r8,%rbx,4),%xmm1
0x00007f90a9065567:   vaddss    0x10(%r11,%rbx,4),%xmm1,%xmm0
0x00007f90a906556e:   vmovss    %xmm0,0x10(%rbp,%rbx,4)
…
```

*Figure 14: Java JIT Assembly without SSE instructions*

### 4.1.1 Vector Library Implementation

As a first step to bring SIMD instructions to Java this work tried a library of simple vectorizable functions written in Java to see if by separating the functions into single line, independent loops the library could get Java's JIT compiler to compile the functions using SIMD instructions. This proved to be unsuccessful as analysis of the JIT compiler output as well as performance numbers indicated that the JIT was still not vectorizing these functions. Therefore an external library was needed.

The next attempt was to create a C++ library, called through Java's JNI interface, which contained all of the basic SIMD functions. This library would then read in the arrays to be processed, perform the correct SIMD operations, and return the results back to Java. To perform the SIMD calculation portion of this task, this work compared hand coded SIMD assembly intrinsic instructions with a basic C++ loop compiled to SIMD instructions through GCC's auto-vectorization compiler optimizations. After multiple comparisons it was found that there were no discernable differences in the performance times and therefore, the implementation involving basic C++ loops compiled with GCC was used to increase ease of code creation and portability. With this implementation, the GCC compilation handled odd sized problems by vectorizing the loop iterations that fit into vectors and then individually calculating the rest. Also if the library needed

to run on an architecture with a different set of SIMD instructions a simple command could be used to recompile the entire library instead of having to worry about hand coding the new instructions.

Due to overheads discussed later in section 4.1.4, the vector library utilizes a buffer to transfer all of the data to and from the vector library. On the C++ side the buffer is loaded through the GetDirectBufferAddress() command and then can be treated as a pointer to an array. Once the C++ library had the address to the buffer all it needs is the offset to any variables involved in the calculation and the size of the vector operation to be performed. After the vector library has determined where the operand arrays are located in the buffer and set their pointers, the operands can be treated as normal arrays. An example of this is provided below in Figure 15. In this example three arrays are being accessed through the buffer: two operand arrays and one destination array. From there the GCC optimized assembly code takes care of determining how many iterations of the loop can be vectorized and how many iterations need to be calculated individually. Since Java and C++ are accessing the same area in main memory there is no need to return any results. They are already in the destination portion of the buffer waiting for Java to use.

Functions like the one in Figure 15 below were written for all possible vector operations needed: add, subtract, multiply, divide, modulus, and, or, square, square root, cos, sin, max, min, abs, log, and exp. In addition, each operation had functions that could handle int, float, double, short or long inputs and any combination of constants and arrays within the input. For example there are separate functions that would handle *a[] - b* and *a - b[]* and *a[] – b[]* for each data type: int, float, double, short, and long. The combination of all of these functions and their associated header file created the C++ side of the vector library.

```
/*********************************************************
* Name: addXIntBuff
* Inputs: JNIenv * env, and jobject obj are handled by Java and required for JNI Functions
* Inputs: aStart (offset of first element of first array from the start of the buffer)
* Inputs: bStart (offset of first element of second array from the start of the buffer)
* Inputs: cStart (offset of first element of the destination array from the start of the buffer)
* Inputs: size (number of elements to be summed)
* Return: none (result is stored in the array at cStart)
* Description: C[0,1,...,size] = A[0,1,...,size] + B[0,1,...,size]
*********************************************************/
JNI_JAVA(void, VectorSubstitutions, addXIntBuff)
(JNIEnv * env, jobject obj, jobject buff, jint aStart, jint bStart, jint cStart, jint size) {
  jint* a = (jint*)env->GetDirectBufferAddress(buff);
  jint* b = &a[bStart];
  jint* c = &a[cStart];

  a+= aStart;

  jint i;
  for(i=0; i<size; i++) {
     c[i] = a[i] + b[i];
  }
   return;
}
```

*Figure 15: Vector Library addXIntBuff*

## 4.1.2 Java Math Library

In implementing vector library functions equivalent to Java Math library functions, an interesting behavior was found. The vector instructions outperformed their Java Math library functions by far more than possible simply due to SIMD performance gains. After further investigation it was found that not all Java Math library functions take advantage of the hardware equivalents built into many processors. Instead many of them call a software equivalent in StrictMath which implements the function in software to ensure accuracy of the result [24]. This allows the vector library equivalent to gain an additional performance gain resulting from utilizing the available hardware implementations of these functions. To ensure that the same accuracy is achieved each benchmark was checked for valid results upon completion and all of the vectorized implementations yielded the same results as their Java Math equivalents.

## 4.1.3 NIO ByteBuffer

The optimal way to interact with the newly created vector library was through the Java New I/O (NIO) API. The NIO API was developed to allow Java to interact with features that are I/O intensive in J2SE 1.4 [23]. These APIs are meant to provide access to low-level software that can perform the most efficient function implementations for a specific set of hardware. Data transfers through the NIO interface are performed through buffers. These buffers are a contiguous memory that is visible to both Java and non-Java code. By having access to the same memory from outside Java, it eliminates the need for additional expensive data copying [24].

In the vector library implementation, all of the variables are passed through the use of a single ByteBuffer. Figure 16 below illustrates the layout of a typical buffer in a vectorized Aparapi implementation. Variables declared outside the kernel are only stored in the buffer once as they are to be consistent among all kernels. Variables declared within the kernel, however, are kernel dependent and need to be kept separately. In the vector implementation there is one kernel per thread and each vector kernel executes the code equivalent to the number of kernels represented by the vectorization length. Therefore to ensure each kernel gets its own set of internal variables within the buffer, each thread gets its own set of internal variable arrays. Each array then contains the internal variable for each of the original replaced by that specific vector implementation. For example, if Figure 16 represents the buffer of an 8 threaded vector implementation of 1024 Aparapi kernels, Thread 1 Internal Variables would contain arrays corresponding to internal variables for kernels 0-127, Thread 2 Internal Variables would contain arrays corresponding to internal variables for kernels 128-255, and so on.

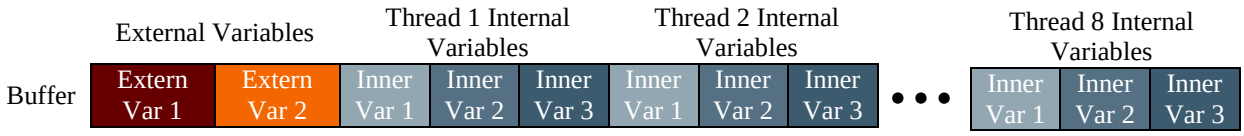


*Figure 16: Buffer Layout*

Interacting with these NIO buffers on the Java side is done through a series of basic commands. First the buffer must be allocated to the correct size. Since the single buffer will contain all of the data transferred back and forth throughout the entire program, this buffer must be large enough to hold all of the variables accessed within the kernel. After the buffer is allocated the byte order must be set to ensure the data is written and read in the same order. In the case of this research, the low level library used a little endian byte order.

Once the buffer is created data can be easily written or read through the put and get functions; both of which have implementations for float, int, double, short and long. To access a specific variable in the vectorized Aparapi code, its buffer index must be calculated. This is done

by taking the starting point of the variable's array plus the variable's array index. This value is then multiplied by the number of bytes used to store a variable of that type: 2 for short, 4 for float and int, and 8 for double and long. An example of each type of buffer interaction from the Java side can be seen below in Figure 17.

```
…
int bufferSize = …
final ByteBuffer buffer = ByteBuffer.allocateDirect(bufferSize);
buffer.order(ByteOrder.LITTLE_ENDIAN);
…
buffer.putInt((aBufferStart + elementIndex)*sizeOfInt, a[elementIndex]);
…
a[elementIndex] = buffer.getInt((aBufferStart + elementIndex)*sizeOfInt);
…
```

*Figure 17: Code Snippet Illustrating the use of the ByteBuffer*

### 4.1.4 Java Library interaction.

On the Java side, this work implemented a VectorSubstitutions class in Aparapi that loads the library and contains all of the function prototypes for the C++ vector library. The function names are broken into 5 pieces. The first of which is the operand name. The operations available in the vector library are add, subtract, multiply, divide, modulus, and, or, square, square root, cos, sin, max, min, abs, log, and exp. All of these functions also have an accumulation form that adds the result to the value already stored in the buffer. This functionally is achieved by adding Accum to any of the function names, i.e. mulAccum. This is immediately followed by an X indicating that the vector length is a parameter of the function. Initial versions of the library contained fixed sized functions, but since there was no performance penalty for passing in the length these were considered redundant and removed. The next part of the function name is the operand type. This can be Int, Float, Double, Short, or Long depending on the operand and destination type. At this time the library only contains functions for operations of all the same type, but functions for operations with multiple types could be added. The fourth part of the function name is not always included. For operations involving only arrays this field of the name is omitted. For operations where the first operand is a constant and the rest of the operation contains arrays Const1 is used. Similarly if operand 2 is the only constant Const2 is used. The final piece of function name signifies the library type. For all calls to the C++ vector library Buff is used. For function calls to the completely Java vector library that did not end up vectorizing, Java is used for this part of the function name. Figure 18 below illustrates the correct combination of these elements to create a

function name while Figure 19 illustrates a few actual function examples within the VectorSubstitutions class.

operandXTypeConstantsLibrary

*Figure 18: Function Name Breakdown*

```
public class VectorSubstitutions{
  static {
    System.loadLibrary("vectorSubstitutions");
  }
  public native void addXIntBuff(ByteBuffer buff, int aStart, int bStart, int cStart, int size);
  public native void addXIntConst1Buff(ByteBuffer buff, int a, int bStart, int cStart, int size);
  ... // function prototypes for all other operation, type and variable type combinations
}
```

*Figure 19: Vector Library Function Prototypes Used to Interact with the C++ Vector Library*

Any Java function that wants to use a vector function simply needs to include the VectorSubstitutions class in the include portion of the java file, ensure that it is using a NIO buffer to store any data that needs to be used by a vector function in the kernel and then call the appropriate vector functions in the vector kernel. These steps shown below in Figure 20 illustrate the Java code to implement a vector add Aparapi kernel.

```
import com.amd.aparapi.Kernel;
import com.amd.aparapi.Range;
import com.amd.aparapi.internal.jni.VectorSubstitutions;
import java.nio.*;


public class Main{

  public static void main(String[] _args) {

    final int size;
    final int remainder;
    final int remainderStart;
    final int threads = Runtime.getRuntime().availableProcessors();
    final int size = 8192*1024;

    final VectorSubstitutions sub = new VectorSubstitutions();
```

```
final ByteBuffer buffer = ByteBuffer.allocateDirect(size*4*3);
buffer.order(ByteOrder.LITTLE_ENDIAN);

final int aStart = 0;
final int bStart = size*4;
final int cStart = size*8;
final int vecSize = size/threads;

remainder = size % threads;
remainderStart = (size / threads) * threads;

for(int i=0; i<size; i++) {
   buffer.putFloat(i*4,i * 3.43 + 1);
   buffer.putFloat(i*4+bStart,i * 4.35 + 2);
}

Kernel kernelVec = new Kernel(){
  @Override public void run() {
     int vecSizeI = vecSize;
     //Add
     int gid = getGlobalId();
     if(gid < remainder){
        gid = gid*(++vecSizeI);
     }
     else{
        gid = gid*vecSizeI + remainder;
     }
     sub.addXFloatBuff(buffer,gid,bStart/4+gid,cStart/4+gid,vecSizeI);
  }
};

kernelVec.setExecutionMode(Kernel.EXECUTION_MODE.JTP);
kernelVec.execute(size/vecSize);
System.out.println(kernelVec.getExecutionTime());
kernelVec.dispose();
```

*Figure 20: Vectorized Add Kernel*

### 4.1.5 Vector Library Overheads

While the first implementation of the vector library simply passed the operand and destination arrays to the C++ library as arguments of a function, testing of this library quickly yielded that even though the calculation portion of the code was achieving significant speedups due to the SIMD operations, there was a data transfer overhead that overshadowed these speedups. Performance analysis showed that only 22% of a 1024 x 1024 matrix multiply was spent on the calculation piece inside the vector library and that the other 78% was spent in JNI and transfer overheads. This was not a viable option so other alternatives were explored.

The final vector library solution still involved using a C++ library called through JNI, but instead of passing all of the arguments as native elements, the arguments were passed through a NIO buffer. A Java NIO buffer allows for both Java and the C++ library to access the same area of memory eliminating the need for copying the arguments between the languages. This yielded a significant performance improvement over the basic C++ JNI library. The library now spent 98% of the time for the 1024 x 1024 matrix multiply in the computational phase and only 2% in overhead through the JNI call and data transfer. This enabled the SIMD performance gains to be transferred to the Java code without being offset by overhead costs.

Testing also revealed that optimal performance was achieved by packing all arguments passed to the vector library function into a single buffer. When passing the arguments to the vector library function in separate buffers there is a small overhead for each GetDirectBufferAddress() call. However, if all of the arguments are in a single buffer this overhead is only incurred once and offsetting pointers to different start places in the buffer takes a nominal amount of time compared to the GetDirectBufferAddress() load.

# Chapter 5

# Auto-Vectorization

Vectorization is the process of converting code with operations that act on a single pair of operands at a time and converting those into vector implementations that operate on multiple operands at a time. Traditionally this involves determining whether sections of an innermost loop can be run in parallel and therefore vectorized. The Loop Parallelization Theorem from Optimizing Compilers for Modern Architectures [13], states that any single-statement loop that has no dependence can be run completely in parallel and therefore is vectorizable. For loops with more than one statement, loop distribution can be used to split some of the statements into multiple loops with no internal loop dependencies. For example, Figure 21 below can be split into two loops that are both vectorizable. However, not all loops can be split this way. Figure 22 below illustrates as case where there are backward-carried dependencies and a loop independent dependence between the statements. This causes invalid results after loop distribution and therefore the code cannot be distributed or vectorized. [13].

In this work each Aparapi kernel is already determined to be independent and is treated as the innermost independent loop. Therefore removing the need for complicated dependency checks and allowing statements to be vectorized across kernels and, reducing the number of kernels that the processor needs to run. The auto-vectorization tool described below takes advantage of this fact to implement the vector library described in Chapter 4.

```
//before loop distribution
for(int i=0; i<N; i++){
    a[i+1] = b[i] + c;
    d[i] = a[i] + e;
}
//after loop distribution
for(int i=0; i<N; i++){
    a[i+1] = b[i] + c;
}
for(int i=0; i<N; i++){
    d[i] = a[i] + e;
}
```

*Figure 21:Valid transformation*

```
//before loop distribution
for(int i=0; i<N; i++){
    b[i] = a[i] + e;
    a[i+1] = b[i] + c;
}
//after loop distribution
for(int i=0; i<N; i++){
    b[i] = a[i] + e;
}
for(int i=0; i<N; i++){
    a[i+1] = b[i] + c;
}
```

*Figure 22: Invalid Transformation*

# 5.1 Auto-Vectorization Tool

This work presents and employs the use of an auto-vectorization tool that reads in Aparapi Java source code as an input and produces correctly vectorized Java source code using the vector library described above in Chapter 4 as an output. This allows for the end user to achieve the performance benefits of using the vectorization library without spending any effort on code modifications.

In order to parse and read the Java source code, this work takes advantage of the Java Parser toolkit [14]. This toolkit includes a basic API written in Java that allows the user to read in a Java source file and break it down into an Abstract Syntax Tree (AST) that is easy to work with. From there the AST can be modified as needed in order to add the appropriate vectorization instructions and then reassembled back into a Java source code file.

This work expands on the Java Parser to create a tool that can read in Aparapi Java source code, detect sections of the code that can be vectorized with the vectorization library, and generate properly vectorized source code as a result. This includes adding the extra include statements, generating the ByteBuffer in which data will be stored and transferred between Java and C, correctly populating the ByteBuffer before the kernel is executed, converting kernel instructions into their vector library counter-parts, and reading any necessary variables out of the ByteBuffer after the kernel execution has completed.

In order to complete this task the auto-vectorizer makes several passes over the AST. On the first pass through the AST the auto-vectorizer scans the code and searches for any kernels that are executed in the code as well as the size of the kernel to be executed. This is to make sure that only code executed inside of a valid kernel is vectorized as only data between kernels is proven to be completely independent and therefore vectorizable.

Knowing the number of kernels executed also allows the auto-vectorizer to determine the best vector size for the code. For two and three dimensional problems this is simple. The auto-vectorizer simply vectorizes along the last dimension. For example, a two dimensional image manipulation source code that would normally execute on a two dimensional range of (x,y) would instead execute on a one dimensional range of x and be vectorized along the second dimension y.

The last dimension is chosen because multi-dimensional programs are more likely to access array elements sequentially in the last dimension and any operation containing non-sequential accesses cannot be vectorized. When the auto-vectorizer detects operations on arrays it will only vectorize operations that act on sequential array elements for sequential kernels. For example the sample code below in Figure 23, is vectorizable because the vectorization library can load sequential elements into registers with only one instruction. However, the sample code below in Figure 24 is not vectorizable because the b array is accessing non-sequential elements. Since the AST breaks each equation down to its smallest element, the auto-vectorizer is able to detect these differences. It looks at the index of each array access in a possible vector instruction and checks for the presence of one and only one variable that varies with the global id in the vectorizable dimension and that the only operations on that variable are additions and subtractions.

```
int x = getGlobalId(0);
int y = getGlobalId(1);
a[x+y] = b[width*x+y] + c[y];
```

*Figure 23: Vectorizable Operation Example*

```
int x = getGlobalId(0);
int y = getGlobalId(1);
a[x*y] = b[width*y+x] + c[x+y];
```

*Figure 24: Un-Vectorizable Operation Example*

For a single dimensional problem, the task of determining the vector size is not as straight forward. The tool allows for the user to enter a vector size manually if the programmer has a specific desired vectorization length, but can also determine a good vector size on its own. From analysis of several benchmarks, described further in section 6.3, it was determined that optimal performance was achieved with the largest possible vector size that still evenly utilized all available threads. This is due to the fact that any cache performance penalty incurred by using large vector sizes is smaller than the overhead incurred by the additional JNI calls of using smaller vector sizes. For example, a program with 16384 kernels would be best mapped on a processor with 8 threads by running 8 vector kernels each with a vector length of 2048.

Once the auto-vectorizer knows the boundaries of the vectorizable regions and the vector size, it then is able to reduce the number of kernels to be executed by the vector size. This is accomplished by either replacing any Range creations associated with the kernel being vectorized

with a Range of the original size divided by the vector size or by replacing the arguments of the Kernel's execute call with arguments reduced by vector size.

In situations where the problem size is not a multiple of the vector size, individual kernels have their vector size increased to accommodate the remainder. For example, if there were 70 kernels to be divided among 8 vector kernels, the vector size of the first 6 vector kernels would be 9 and the last 2 vector kernels would have a vector size of 8. The C++ vector library will then vectorize the vector kernel operations it can and then perform the remaining operations individually. The reasoning for choosing this method to handle the extra kernels is explained further in section 6.4.

When reducing the number of kernels to be executed, the code inside the kernel is placed inside a for loop that loops from zero to the vector size and any call to getGlobalId() is replaced with getGlobalId() * vector size + vector loop iteration as illustrated below in Figure 25. This ensures the resultant code remains still yields the same end results.

```
@Override public void run(){
    for(int vecLoopIterator=0; vecLoopIterator<vecSize; vecLoopIterator++)
   {
        int gid = getGlobalId() * vecSize + vecLoopIterator;
        normal kernel code
   }
}
```

*Figure 25: Kernel Loops*

For some benchmarks this alone is enough to generate some speedup for kernels executed in the Java Thread Pool (JTP). When Aparapi code is written, the main goal is to run on a GPU and therefore the having more parallel kernels running at a time generally results in a faster run time. However, a CPU cannot run nearly as many concurrent threads as a GPU and is often limited to numbers around 4, 8 or 16 rather than the thousands than can be run on GPUs. Because of this, having more kernels than available threads can actually hinder performance as each kernel created has some non-negligible overhead. When kernels are condensed and run as one kernel per thread with loops executing multiple kernels worth of code, the JTP still can achieve its maximum parallelization, without incurring the additional startup overhead for each of the combined kernels.

If the auto-vectorizer detects that a line of code inside the loop can be vectorized, that line is split off into its own loop to later be vectorized. This is illustrated below in Figure 26. If the vectorizable instruction uses array variables, the auto-vectorization tool checks to see that the array is accessed sequentially for sequential kernels. There are three possible outcomes of this check. First if it is sequentially accessed, the array is marked as a buffer variable. Next, if the array accesses are independent of global id then the value is treated as a constant. Finally, if the array

accesses varies with global id, but not in a sequential way then the calculation is determined to not be vectorizable and left in the original, unseparated loop form.

Another possibility for a vectorizable instruction is that is uses primitive variables that vary within the loop. When a primitive variable is detected in a possibly vectorizable instruction, the tool first checks to see if the variable varies from kernel to kernel. If the variable varies, the auto-vectorizer marks it as a buffer variable and sets aside enough space in the buffer for the variable times the vector size times the number of threads that can be active at a time, see Figure 16. This ensures that the variable is properly stored for each instance in the vectorized operation as well as that concurrent threads do not overwrite each other's variables.

The auto-vectorizer also keeps track of when this process separates a non-buffer variable from the loop it was declared in. These variables are also added to the list of variables that must be stored into the ByteBuffer so that their value can be preserved throughout the kernel execution for when it is needed later.

Once the source code is split into the correct loops, see example in Figure 26 below, the auto-vectorization tool loops through and replaces any references to a buffer variable with code that reads associated variable from the ByteBuffer.

```
@Override public void run(){
    for(int vecLoopIterator=0; vecLoopIterator<vecSize; vecLoopIterator++)
   {
        int gid = getGlobalId() * vecSize + vecLoopIterator;
        normal kernel code
   }
   for(int vecLoopIterator=0; vecLoopIterator<vecSize; vecLoopIterator++)
   {
      vectorizable operation
   }
   for(int vecLoopIterator=0; vecLoopIterator<vecSize; vecLoopIterator++)
   {
      remaining kernel code
   }
}
```

*Figure 26: Vector Loops*

```
@Override public void run(){
    for(int vecLoopIterator=0; vecLoopIterator<vecSize; vecLoopIterator++)
    {
        int gid = getGlobalId() * vecSize + vecLoopIterator;
        normal kernel code
    }
    sub.vectorizableOperation(buffer, variable inputs, vecSize);
    for(int vecLoopIterator=0; vecLoopIterator<vecSize; vecLoopIterator++)
    {
        remaining kernel code
    }
}
```

*Figure 27: Vectorizable Loop Turned Into Vector Library Call*

Once the code is properly setup in loops and all of the buffer variables have correctly been replaced with the proper buffer accesses, the auto-vectorization tool enters the actual vectorization stage. In this stage the auto-vectorizer picks out all of the for loops within executed kernels that have a single instruction in their body. This statement is then passed on to checked to see if it is vectorizable. If it is then the entire for loop is replaced with the correct method call to the JNI vector method. Otherwise the loop is left alone, Figure 27.

To determine if the statement is vectorizable the auto-vectorization tool breaks down the statement to determine if a corresponding instruction exists in the vector library. If the statement is a binary expression (+, -, *, /, &, |, etc…), then the tool checks to see that the JNI vector library contains a function to not only handle that binary expression, but also with the correct variables (constant + constant, constant + buffer variable, buffer variable + buffer variable, etc…). Any array variables are also checked to ensure that their indexes are sequentially accessed by sequential kernels. If a corresponding vector instruction exists and the inputs are yield a vectorizable instruction, then this segment of the tool passes back the correct method call with the correct arguments including the variable name for constants and the correct buffer start position for buffer variables. Otherwise it leaves the code inside of its original loop.

The auto-vectorizer also has the option to check for calls to the Java Math library and vectorize them. If enabled the auto-vectorizer checks for these calls within a vector loop to determine if the Math call and corresponding argument types have a matching function in the JNI vectorization library. If so, then the loop is replaced with the correct JNI function.

However, since the hardware equivalent of some of the Java Math functions does not exactly match the implementation in the Java Math library [15], these vectorizations can be turned off in the auto-vectorizer. All of the benchmarks run in this paper had verification steps to ensure that the correct results were achieved with vectorization, and all of the benchmarks, including those with Java Math functions, passed with the Java Math functions vectorized.

Once all of the for loops have been traversed the tool checks to see if any vectorizations were made. If so, the tool traverses the code and populates the buffer before the kernel execution with any buffer variables that are set outside of the kernel but used within a vector instruction. This method also keeps track of where the first instance of one of the buffer variables populating the buffer so that the tool can create the buffer before this point. If no buffer variables are instantiated before the kernel is executed the auto-vectorizer places the buffer creation code just before the kernel execute call.

The auto-vectorization tool then loops through the source code after the kernel execution call to check for anywhere where a buffer variable is read from and replaces this variable access with a buffer.get corresponding to the variable accessed. In this pass, the auto-vectorizer also adds imports to the top of the Java source code file for the vector library and for ByteBuffer.

At this point the auto-vectorizer has completed the vectorization process and returns the vectorized Java source code. The vectorized Java source code should correctly compile and run just as the original source code would have been. Depending on the project setup, the user may have to link the pre-compiled shared object (.so) vector library file. An example of an Edge Detect kernel and a vectorized Edge Detect kernel are included below in Figure 29 and Figure 30 respectively. Figure 29 implements one instance of a kernel while the vectorized kernel in Figure 30 implements *vecSize* instances of the original kernel at once.

1. *Find executed kernels and determine their length*

2. *Determine the appropriate vec and loop sizes*

3. *Reduce the number of kernels executed by vec size or loop size*

4. *Convert executed kernel implementations into loops split by vectorizable operations (see Figure 26)*

5. *Keep track of all primitive and array variables that vary with the kernel id.*

6. *Keep track of all variables used in vector operations as well as those not used by vector operations but are separated from their declarations by loop splitting. These will become buffer variables*

7. *Replace all instances of internal buffer variables with their buffer access equivalent.*

8. *Replace single line vectorizable loops with their vector library equivalent (see Figure 27)*

9. *If vectorized, populate the buffer before the kernel with any buffer variables declared before the kernel and replace accesses to buffer variables after execution with the correct buffer accesses*

*Figure 28: Auto-Vectorizer pseudo-code*

```
@Override public void run(){
    int x = gidGlobalId(0) + 1;
    int y = getGlobalId(1) + 1;

    //GrayX Loop of Filter
    int grayX = lum[(x-1)*height+(y - 1)] * filter1[0];
    grayX += lum[(x-1)*height+(y)] * filter1[1];
    grayX += lum[(x-1)*height+(y + 1)] * filter1[2];
    grayX += lum[(x)*height+(y - 1)] * filter1[3];
    grayX += lum[(x)*height+(y)] * filter1[4];
    grayX += lum[(x)*height+(y + 1)] * filter1[5];
    grayX += lum[(x + 1)*height+(y - 1)] * filter1[6];
    grayX += lum[(x + 1)*height+(y)] * filter1[7];
    grayX += lum[(x + 1)*height+(y + 1)] * filter1[8];

    //GrayY Loop of Filter
    int grayY = lum[(x-1)*height+(y - 1)] * filter2[0];
    grayY += lum[(x-1)*height+(y)] * filter2[1];
    grayY += lum[(x-1)*height+(y + 1)] * filter2[2];
    grayY += lum[(x)*height+(y - 1)] * filter2[3];
    grayY += lum[(x)*height+(y)] * filter2[4];
    grayY += lum[(x)*height+(y + 1)] * filter2[5];
    grayY += lum[(x + 1)*height+(y - 1)] * filter2[6];
    grayY += lum[(x + 1)*height+(y)] * filter2[7];
    grayY += lum[(x + 1)*height+(y + 1)] * filter2[8];

    int truncVal = (int) Math.sqrt(grayX*grayX + grayY*grayY);
    if(truncVal > 255) {
        truncVal = 255;
    }
    else if(truncVal < 0) {
        truncVal = 0;
    }
    magnitude[x*width+y] = 255-truncVal;
}
```

*Figure 29: Edge Detect Kernel*

```
@Override public void run(){
    int grayXStart = getThreadId()*threadOffsetMul + initialThreadOffset;
    int grayYStart = grayXStart + vecSize;
    int temp1Start = grayYStart + vecSizeI;
    int temp2Start = temp1Start + vecSizeI;
    int gid = getGlobalId()*vecSize;
    int x = gid/VecSize + 1;
    //GrayX Loop of Filter
    sub.mulXIntBuff(buffer,filter1[0],(x-1)*height,grayXStart,vecSize);
    sub.mulAccumXIntBuff(buffer,filter1[1],(x-1)*height+(1),grayXStart,vecSize);
    sub.mulAccumXIntBuff(buffer,filter1[2],(x-1)*height+(2),grayXStart,vecSize);
    sub.mulAccumXIntBuff(buffer,filter1[3],(x)*height,grayXStart,vecSize);
    sub.mulAccumXIntBuff(buffer,filter1[4],(x)*height+1,grayXStart,vecSize);
    sub.mulAccumXIntBuff(buffer,filter1[5],(x)*height+2,grayXStart,vecSize);
    sub.mulAccumXIntBuff(buffer,filter1[6],(x+1)*height,grayXStart,vecSize);
    sub.mulAccumXIntBuff(buffer,filter1[7],(x+1)*height+1,grayXStart,vecSize);
    sub.mulAccumXIntBuff(buffer,filter1[8],(x+1)*height+2,grayXStart,vecSize);
    //GrayY Loop of Filter
    sub.mulXIntBuff(buffer,filter2[0],(x-1)*height,grayYStart,vecSize);
    sub.mulAccumXIntBuff(buffer,filter2[1],(x-1)*height+(1),grayYStart,vecSize);
    sub.mulAccumXIntBuff(buffer,filter2[2],(x-1)*height+(2),grayYStart,vecSize);
    sub.mulAccumXIntBuff(buffer,filter2[3],(x)*height,grayYStart,vecSize);
    sub.mulAccumXIntBuff(buffer,filter2[4],(x)*height+1,grayYStart,vecSize);
    sub.mulAccumXIntBuff(buffer,filter2[5],(x)*height+2,grayYStart,vecSize);
    sub.mulAccumXIntBuff(buffer,filter2[6],(x+1)*height,grayYStart,vecSize);
    sub.mulAccumXIntBuff(buffer,filter2[7],(x+1)*height+1,grayYStart,vecSize);
    sub.mulAccumXIntBuff(buffer,filter2[8],(x+1)*height+2,grayYStart,vecSize);
    //non-vectorizable region
    for(int vecLoopIterator=0; vecLoopIterator < vecSizeI; vecLoopIterator++) {
        int truncVal = buffer.getInt(y*4 + temp1Start*4);
        if(truncVal > 255) {
            buffer.putInt(y*4 + temp1Start*4, 255);
        }
        else if(truncVal < 0) {
            buffer.putInt(y*4 + temp1Start*4, 0);
        }
    }
    sub.subXIntBuff(buffer,255,temp1Start,magnitudeStart+gid,vecSize);
}
```

*Figure 30: Vectorized Edge Detect Kernel*

# 5.2 Auto-Vectorization Inhibitors

While each Aparapi kernel is independent, not all of them can be vectorized. Basic arithmetic calculations vectorize nicely, but when the kernels are full of control logic that varies from kernel to kernel or access non sequential array values, vectorization is no longer guaranteed.

For example, some two dimensional image manipulation benchmarks presented a challenge in that every pixel is not treated the same. Pixels along the outside of the image would undergo one set of transformations, while the inner pixels would undergo another as illustrated in Figure 31. The normal Aparapi model would distribute each pixel into a separate kernel as illustrated in Figure 32. This allows each kernel to select the appropriate set of calculations to be performed and execute normally. When vectorizing, the selection logic becomes complicated.

Traditional auto-vectorization logic would split the two dimensional problem into kernels along one dimension and vectorize along the other, as seen in Figure 33. This presents a challenge due to the fact that different pixels along the vectorization path need to perform different calculations. Vectorization can no longer occur along the entire dimension, but instead the first and last pixel must be handled separately while vectorizing the middle. This can be overcome by manual vectorization, Figure 34, but cannot currently be handled by the current auto-vectorizer. The current tool leaves the kernels split into groups but loops through the group to handle each pixel one at a time. The reasoning behind the group split is discussed in section 3.4.

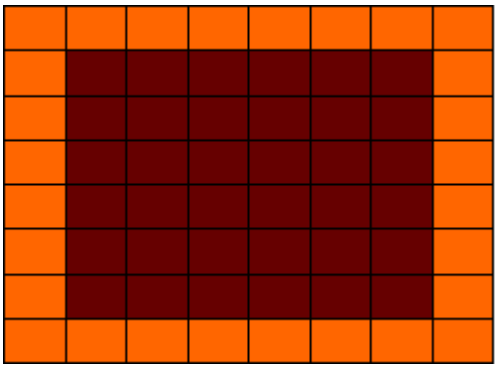

*Figure 31: Image*

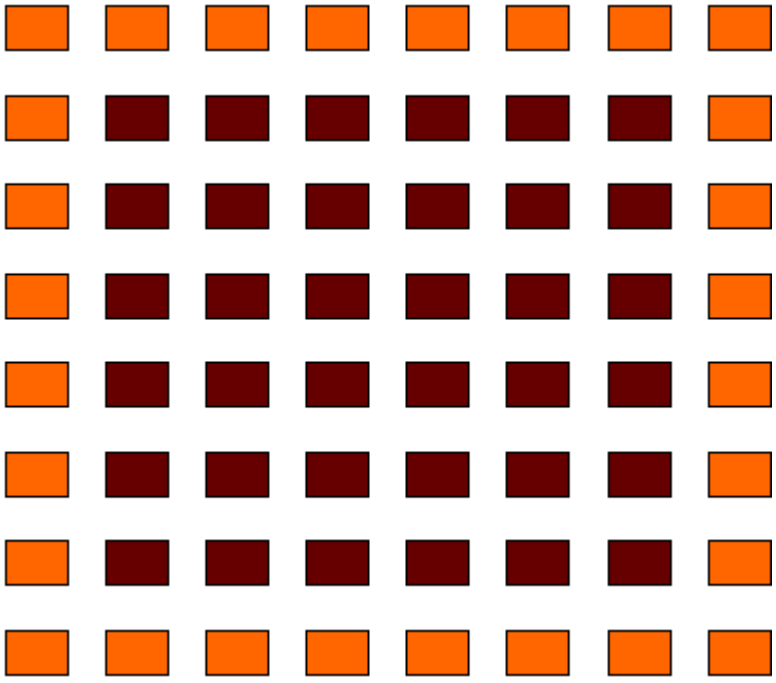

*Figure 32: Normal Aparapi Distribution*

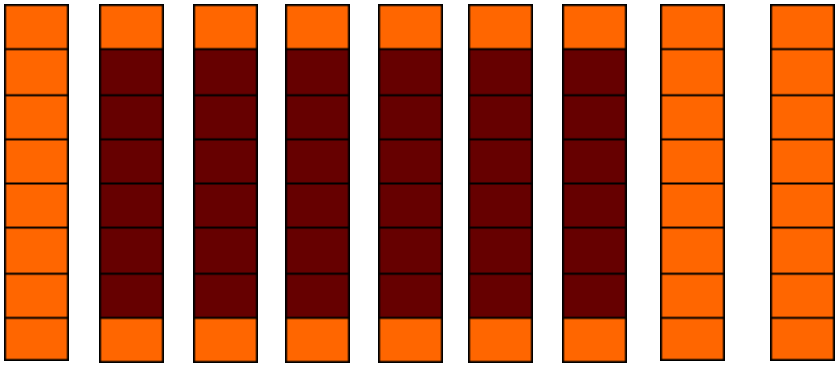

*Figure 33: Auto-Vectorized Groupings*

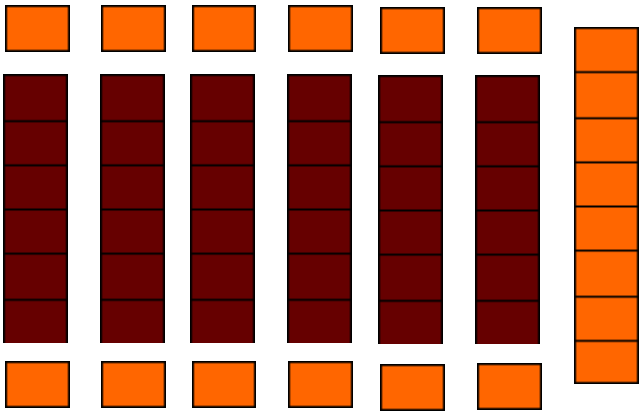

*Figure 34: Hand Vectorized Groupings*

# Chapter 6

# Results

In this chapter the performance of the vectorization library is compared against the Aparapi Java thread pool, looped Aparapi Java thread pool, Aparapi OpenCL on the GPU, and in some cases a library written in Java to try to enable the Oracle Hotspot JIT compiler to vectorize instructions. Being one of the first works to implement general purpose vectorization in Java this work relies on comparison against the non vectorized code for its main comparison. The only other true competitor to this work [32] had one similar benchmark which is compared in the Series Test section below. We evaluate performance on both a desktop computing environment as well as a server to ensure that the performance gains transfer across systems of multiple types.

## 6.1 Experimental Setup

All of the experiments were run on a desktop machine and a server. The desktop machine contained an 8 core AMD FX 8350 and a GeForce GT 610 with 48 CUDA cores. All 8 cores were clocked at the maximum 4GHz throughout the testing to ensure consistent results. The server consisted of 4 Opteron 6376 2.3 GHz 16-core processors for a total of 64 parallel processing cores. The server also contained an Nvidia GeForce GTX Titan GPU with a total of 2688 parallel CUDA cores.

All benchmarks were run on each machine for each kernel implementation:

- Java thread pool (JTP)
    - Stock Aparapi benchmark told to run in the Java thread pool
- vectorization library (VEC)
    - Stock Aparapi benchmark run through the auto-vectorization tool to produce Aparapi Java that utilizes the vectorization library.
- looped kernels on the Java thread pool (LOOP)

    - A reduced number of Aparapi kernels that each perform the work of multiple standard Aparapi kernels.
- Graphics Card (GPU).
    - Stock Aparapi benchmark told to run on the GPU.

Some benchmarks were also run with a vector library written in Java designed to isolate vectorizable regions so hopefully Java's JIT compiler would vectorize them without the need for JNI overhead. However, as results showed, the JIT was not vectorizing the library and performance numbers were never as good as the LOOP case it was most similar to. Because of this the full Java library was not implemented for all benchmarks. Also, non vectorizable benchmarks were still run on the JTP, LOOP, and GPU cases to determine and discuss the performance differences.

# 6.2 Benchmarks

Since Aparapi is still a work in progress, there are not a lot of compatible benchmarks to begin with. Aparapi contained Matrix Multiply, Blackscholes, Mouse Tracker, Mandel, and Life, but the other benchmarks were converted from other parallel Java or OpenCL benchmarking suites. Table 5 below provides an overview of the benchmarks used, if they were vectorizable across kernels, if vectorization was possible through the auto-vectorizer, and average case speedups. For more detailed results see the specific section for each benchmark.

| **Benchmark (Vector type)** | **Auto - Vectorizable** | **Average Speedup Across Both Machines** | | |
|---|---|---|---|---|
| | | Kernel Loop vs. Normal | Vectorized vs. Normal | Vect vs. Kernel Loop |
| **VecAdd (float)** | Yes | 1.7 | 3.6 | 2.0 |
| **VecCos (double)** | Yes | 1.1 | 4.2 | 3.9 |
| **MatMul (float)** | Yes | 5.3 | 6.9 | 1.3 |
| **EdgeDetect (int)** | Yes | 1.5 | 6.0 | 4.0 |
| **SeriesTest (Double)** | Yes | 1.0 | 12.4 | 12.4 |
| **MouseTracker (float)** | Yes | 1.0 | 1.3 | 1.3 |
| **Blackscholes (Float)** | Yes | 1.2 | 1.7 | 1.4 |
| **Mandel (N/A)** | No | 1.0 | - | - |
| **Life (N/A)** | No | 1.2 | - | - |
| **Kmeans (N/A)** | No | 2.2 | - | - |

*Table 5: Benchmark Vectorizability and Average Performance*

### 6.2.1 VecAdd

VecAdd is a simple vector addition calculation. It takes data stored in two, single dimensional arrays, adds the corresponding elements, and stores the result in a third single dimensional array. This was a very simple micro-benchmark meant to isolate the vector operations to get a direct comparison of the vector operations against their non-vectorized Aparapi counterparts. Example code for each implementation as well as the sequential java equivalent of the micro-benchmark can be found below in Figure 35 through Figure 38.

```
int[] a = new int[size];
int[] b = new int[size];
int[] c = new int[size];

//Populate a and b with data
...
for(int i=0; i<array.length; i++){
    c[i] = a[i] + b[i];
}
```

*Figure 35: Java Array Add*

```
Kernel kernel = new Kernel(){
  @Override public void run(){
    int gid= getGlobalId();
    result[gid]=a[gid]+b[gid];
  }
};
Range range = Range.create(result.length);

kernel.execute(range);
```

*Figure 36: Aparapi Array Add*

```
//Populate buffer for a and b
...
final int LoopSize = ...
final int remainder = result.length % LoopSize;

Kernel kernel = new Kernel(){
  @Override public void run(){
     int LoopSizeI = LoopSize;
     int gid = getGlobalId();
     if(gid < remainder){
          gid = gid*(++LoopSizeI);
     }
     else {
          gid = gid*LoopSizeI + remainder;
     }
     for(int vecLoop=0; vecLoop<LoopSizeI; vecLoop++){
         result[gid + i]=a[gid + i]+b[gid + i];
    }
  }
};
Range range = Range.create(result.length/vecLoop);

kernel.execute(range);
```

*Figure 37: Aparapi Loop Array Add*

```
//Populate buffer for a and b
…

Kernel kernel = new Kernel(){
  @Override public void run(){
    int VecSizeI = VecSize;
    int gid = getGlobalId();
     if(gid < remainder){
          gid = gid*(++VecSizeI);
     }
     else {
          gid = gid*VecSizeI + remainder;
    }
    addXIntBuff(buffer, aStart+i, bStart+i, cStart+i, VecSizeI);
  }
};
Range range = Range.create(result.length/VecSizeI);

kernel.execute(range);
//read from buffer at cStart instead of from c[]
```

*Figure 38: Aparapi Vector Add*

As you can see from the results below, once the vector library is able to overcome the JNI overhead costs it can provide significant speedups. These speedups continue to grow until a peak of a 14.6x speedup and for the desktop and a 4.8x speedup for the server, then begin to decrease back down to 2x. Figure 41 seeks to explain this peak behavior and illustrates that while the three CPU kernel types begin decreasing exponentially with an increase in size, the JTP and LOOP kernels begin to level off earlier which produces the large spikes around this section.

Speedups over the theoretical maximum vectorization speedup of 8x illustrate that at some sizes the vectorized kernel also achieves other performance benefits such as better cache efficacy. Both of these peaks come around problem sizes equivalent to the size of the L2 cache of both machines. The desktop has a 2048KB L2 cache meaning that 524288 Java float variables could fit inside, and the server has an L2 cache size of 16 MB meaning it can hold 4194304 Java float variables. Therefore by processing multiple sequential instances of the same instruction at once, the vectorized kernel can benefit from that data already being stored in the L2 cache.

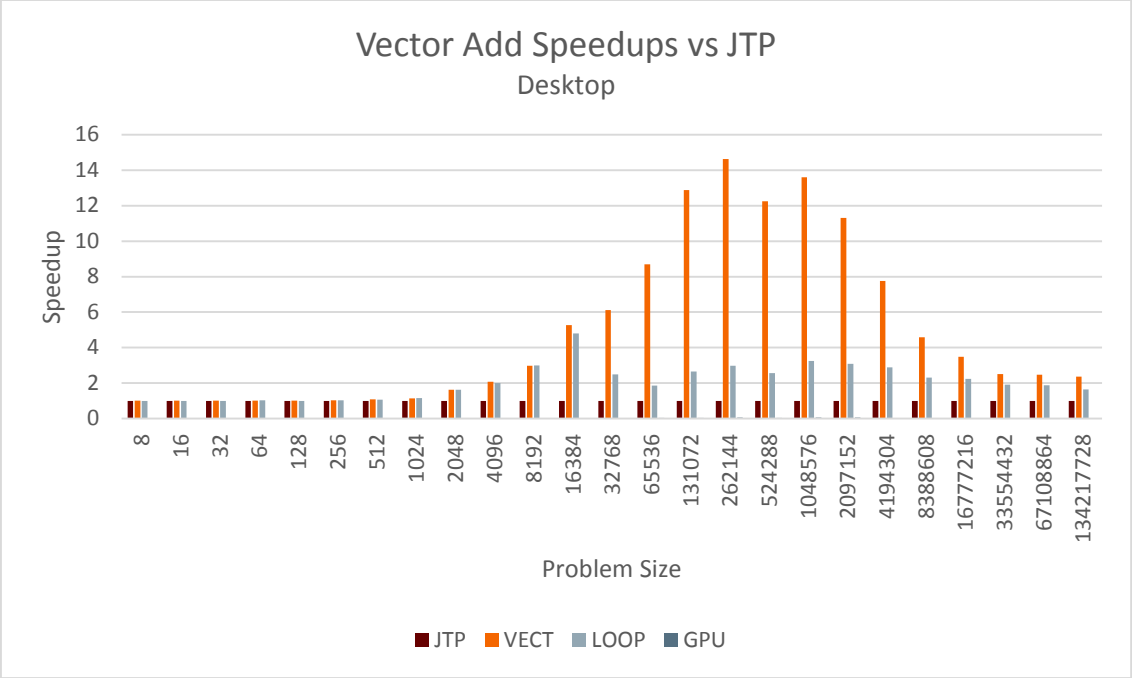


*Figure 39: Vector Add Speedups vs JTP on Desktop*

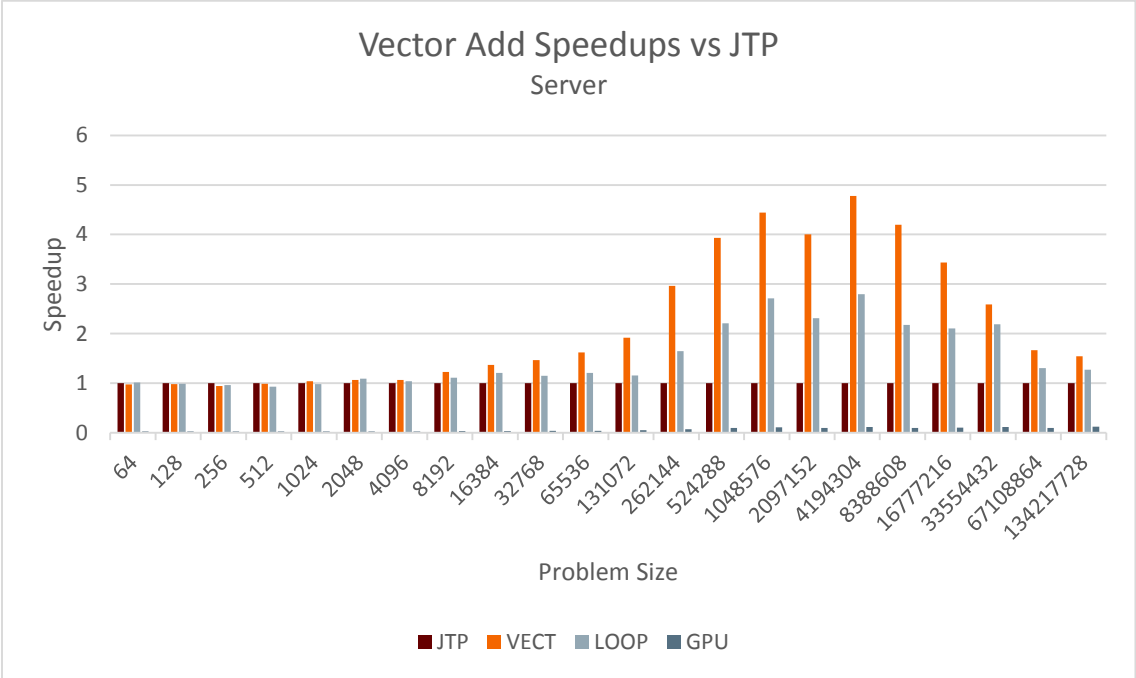


*Figure 40: Vector Add Speedups vs JTP on Server*

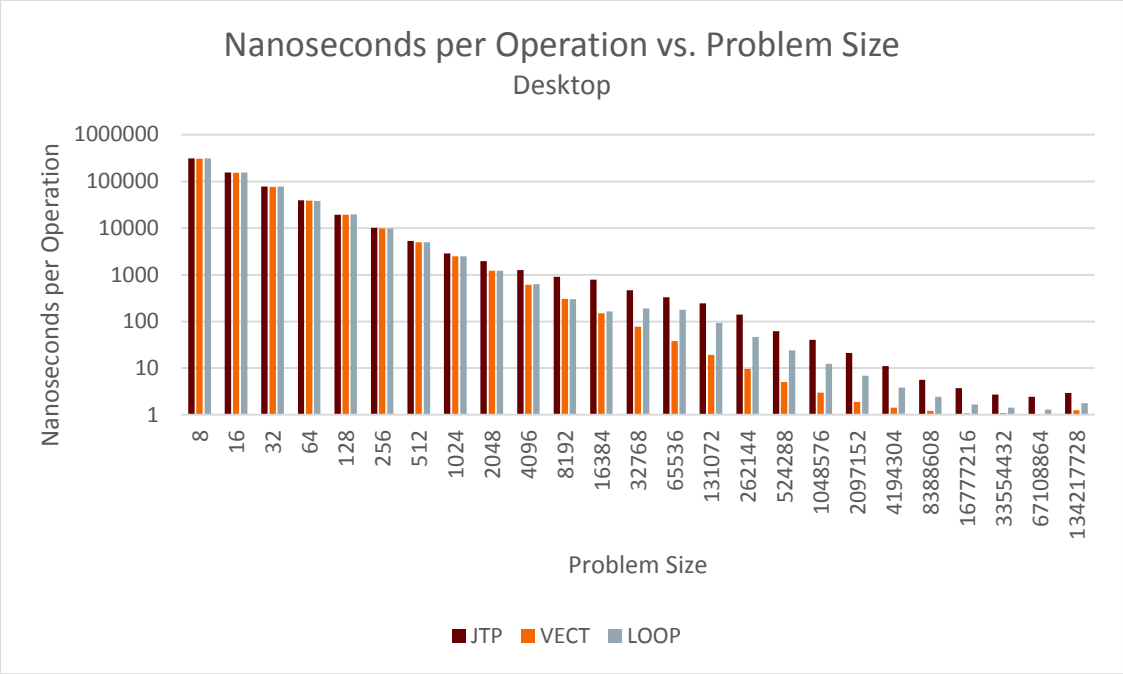


*Figure 41: Nanoseconds per Operation vs. Problem Size*

In the VecAdd case the GPU was not able to compete with the CPU. VecAdd only performs one operation on each piece of data in the three arrays and as such requires three large data transfers: two to the GPU and one back from the GPU relative to the number of parallel operations performed. Due to these data transfer overheads plus the initial startup overhead the GPU was not able perform as well as the CPU for the simple VecAdd.

## 6.2.2 VecCos

VecCos is very similar to VecAdd, except instead of performing an addition on two corresponding data entries and storing the data in a third, VecCos performs a cos operation on each element of an array and stores the result in a target array. VecCos was made into a separate benchmark because it is not only comparing the speed benefits vectorized code against its non-vectorized counterpart but it also compares the speed of hardware cosine operations against Java's Math library which does some of the calculations in software [4]. As with every other benchmark the results of the vectorized code were compared with the non-vectorized to ensure their validity.

As you can see in Figure 42, and Figure 43 below, VecCos follows the same initial trends as VecAdd. The vector case grows strong speedups with increased size, but unlike VecAdd, VecCos maintains its maximum speedups of around 12x for the desktop and 5.5x for the server for the remainder of the problem sizes. This is because as the rest of the kernel is JIT compiled to native code, the cos operation becomes a more prominent factor in the overall runtime and therefore accentuates the performance difference between hardware and software implementations

of the cos operation and allows the vector case that utilizes the hardware implementation to maintain such large speedups.

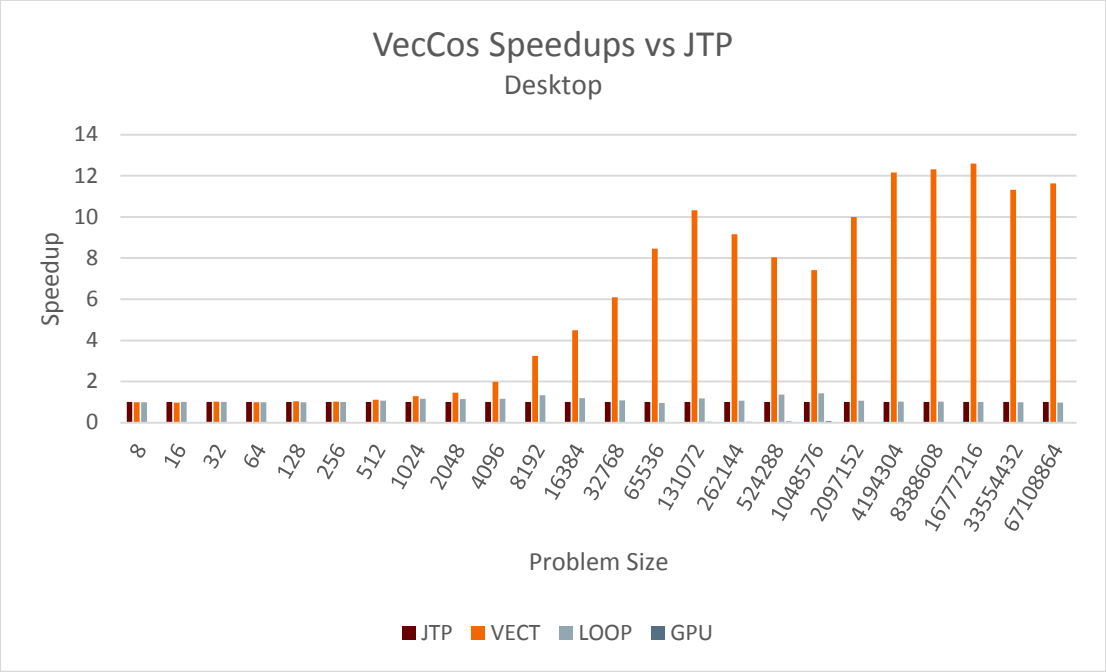


*Figure 42: Vec Cos Speedup vs JTP on Desktop*

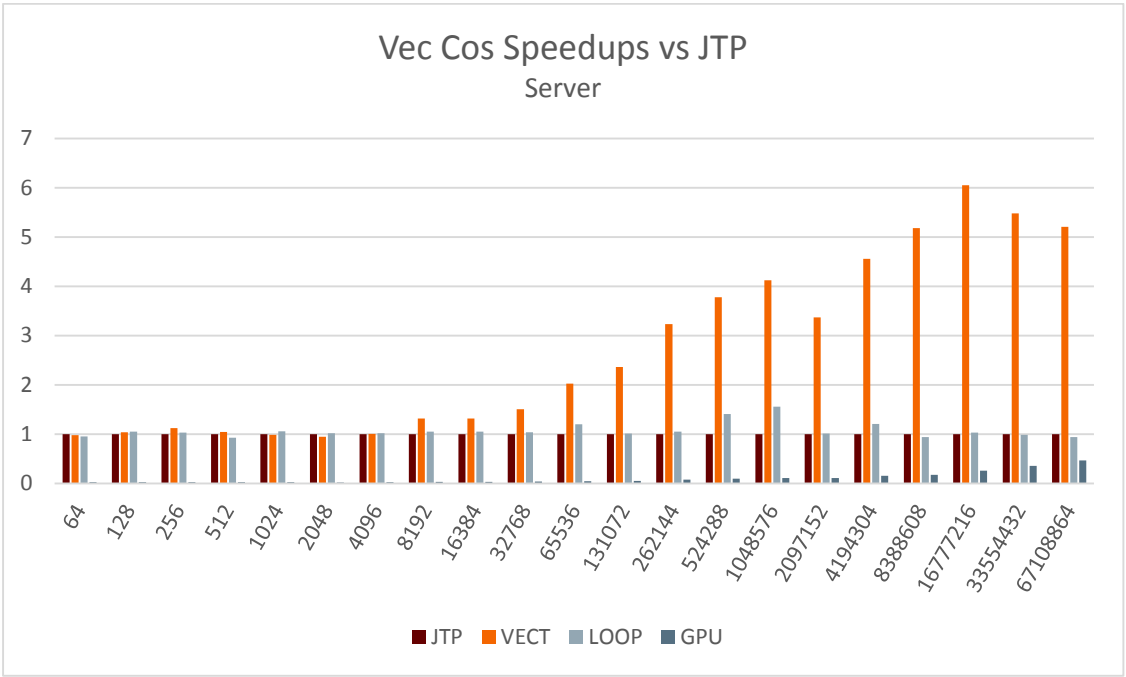


*Figure 43: Vec Cos Speedup vs JTP on Server*

### 6.2.3 MatMul

Matrix multiply was implemented as a basic two dimensional square matrix multiplication, Figure 44. In the creation of Aparapi kernels it was treated as a two dimensional problem with $i$ and $j$ being dimensions and each kernel still looping through $k$, illustrated below in Figure 45. Since vectorization occurs along the last dimension of a two dimensional problem, the vectorized implementation is a one dimensional Aparapi problem with kernels for each $i$, vectorized along $j$ and still performing the $k$ loop inside the kernel, as illustrated in Figure 46.

In order to test a wide range of sizes, matrix multiplies were performed for 8x8, 16x16, 32x32, 64x64, 128x128, 256x256, 512x512, 1024x1024, 2048x2048, and 4096x4096 and across all types of Aparapi kernels. Analysis of this benchmark revealed that cache performance became more important of a speedup factor than vectorization on the CPU. As you can see in the results below, the JTP case was easily outperformed by all of the other CPU cases as a result of poor memory access patterns. The JTP implementation performed each operation of the $k$ loop before moving on to the next iteration of the j loop while the vectorized case performed all of the $j$ loop for each iteration of the $k$ loop. This results in the JTP case jumping around through the $b[]$ array resulting in cache misses while the vectorized case resulted in sequential accesses of both the $a[]$ and $b[]$ arrays. Because of this the LOOP case for matrix multiply was handled specially. It implemented the extra kernels loop inside the k loop to simulate the data access patterns of the vectorized case in order to isolate the speedups from the vectorizations.

Matrix multiply was the first micro-benchmark in which the fully Java vector library was implemented. The JTP kernel was reduced just like in the VEC kernel, but instead of operating in buffers and transferring the data to a C++ library, the two arrays to be multiplied and the length of the multiplication were sent to a simple java function. This function performed the multiplication operation in the same loop that was written in C++ and compiled to vector instructions by GCC's optimizations. The goal was that Java's JIT compiler would recognize the loop as vectorizable instructions and vectorize the operations. However, analysis of the JIT code yielded different results. All of the operations were JIT compiled down to sequential operations and therefore gained no performance gains over the LOOP case with the same variable access patterns.

As you can see from the results shown below in Figure 48 the VEC kernel was still able to achieve speedups over the LOOP case of peaking around 4.5x and averaging 2.1x in the desktop environment, but the low powered GPU failed execution of the largest problem size and was unable to keep up with the CPU kernels on the desktop. For the desktop environment matrix multiply is best performed on the CPU with the VEC implementation with a fallback of the LOOP implementation if the vector library is unavailable at runtime.

While the server showed similar results for the four CPU cases, the high powered GPU finally got its chance to shine. Matrix multiply is a massively parallel problem with a low enough data transfer overhead that allowed the GPU to achieve speedups peaking at nearly 80x over the

64 core server CPUs. In this situation, with a high powered GPU, matrix multiply is best performed on the GPU with a fallback path of the VEC kernel implementation.

```
for(int i=0; i<size; i++){
  for(int j=0; j<size; j++){
    float sum = 0;
    for(int k=0; k<size; k++){
      sum += a[i][k] * b[k][j];
    }
    c[i][j] = sum;
  }
}
```

*Figure 44: Matrix Multiply Code*

```
Kernel kernel = new Kernel(){
  @Override public void run(){
    int gid = getGlobalId();
    int i = gid / size;
    int j = gid % size;
    float sum = 0;
    for(int k=0; k<size; k++){
      sum += a[i*size+k] * b[k*size+j];
    }
    c[gid] = sum;
  }
};
```

*Figure 45: Matrix Multiply Kernel*

```
Kernel vKernel = new Kernel(){
  @Override public void run() {
    int VecSizeI = VecSize;
    int gid = getGlobalId();
     if(gid < remainder){
          gid = gid*(++VecSizeI);
     }
     else {
          gid = gid*VecSizeI + remainder;
     }
    for(int k=0; k<size; k++){
     sub.mulAccumXIntConst1Buff(buffer, a[i*size+k],bStart+k*size,cStart+i*size, vecSize);
    }
  }
};
```

*Figure 46: Vectorized Matrix Multiply*

```
Kernel vKernel = new Kernel(){
  @Override public void run() {
    int VecSizeI = VecSize;
    int gid = getGlobalId();
     if(gid < remainder){
          gid = gid*(++VecSizeI);
     }
     else {
          gid = gid*VecSizeI + remainder;
    }
   for(int k=0; k<size; k++){
    int gid = getGlobalId();
    for(int VecLoop=0; VecLoop<size; VecLoop++){
      int i = gid / size;
      int j = gid % size;
      c[gid] = a[i*size+k] * b[j*size+k];
      gid++;
    }
   }
  }
};
```

*Figure 47: LOOP Kernel*

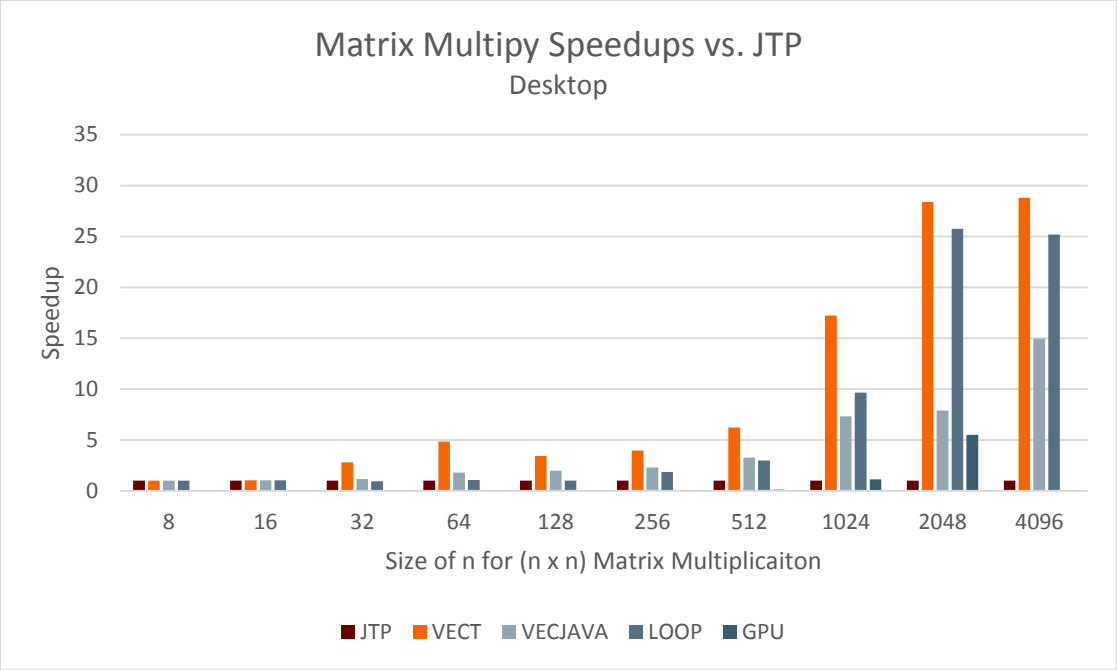


*Figure 48: Matrix Multiply Speedups vs. JTP for Desktop*

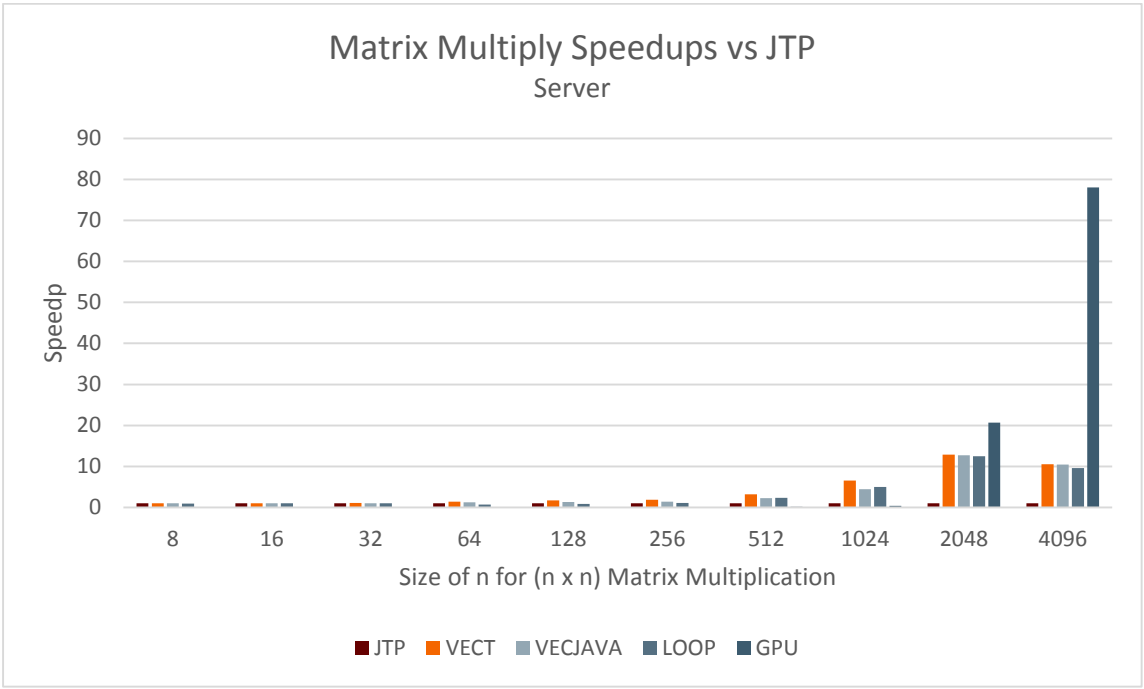


*Figure 49: Matrix Multiply Speedups vs JTP on Server*

## 6.2.4 EdgeDetect

EdgeDetect was based off an Edge Detection Java application written by Robert Sedgewick and Kevin Wayne at Princeton University [27]. Their implementation was converted into Aparapi Java for use in this work. The edge detection algorithm loops through each pixel and checks how different it is from its neighboring pixels to try and determine where the edges in a picture are. An example of starting and resulting pictures for one of the EdgeDetect cases is shown below in Figure 50. Converting the algorithm to Aparapi was simple since all of the calculations for each pixel are independent and therefore assigned their own kernel to operate on. As a two dimensional problem the LOOP and VEC implementations have kernels along the height of the image and loop/vectorize along the width.

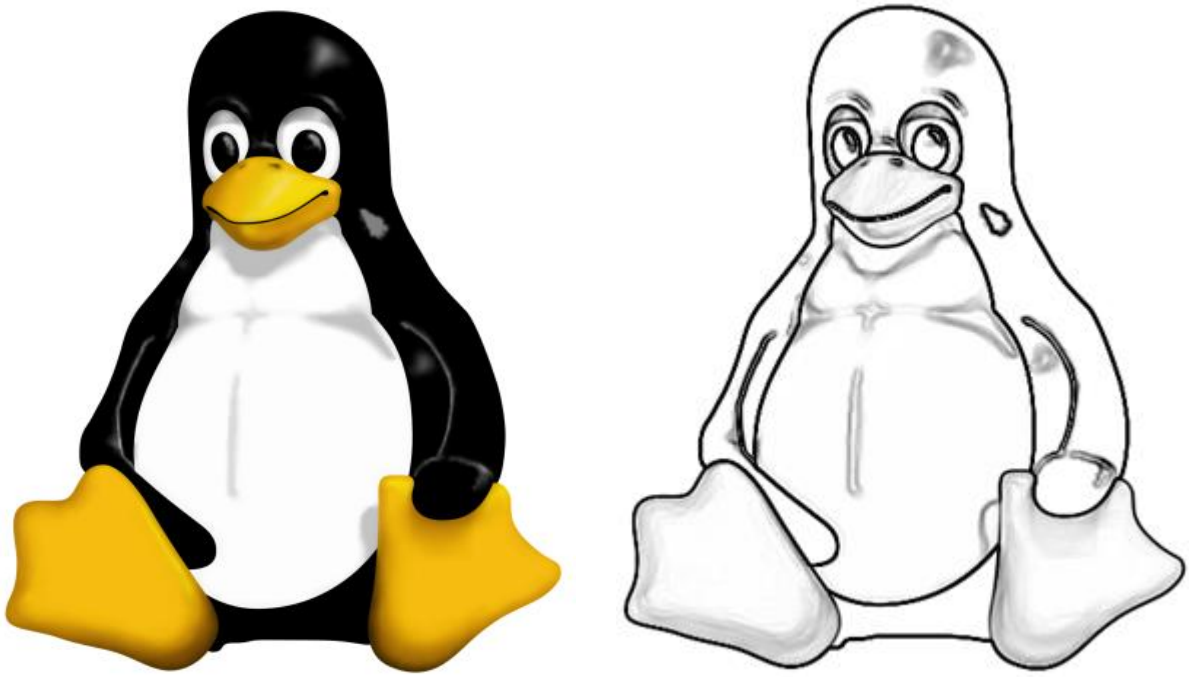

*Figure 50: Edge Detect Example*

Edge detect is another benchmark that benefits from vectorization. Since the benchmark consists of completely vectorizable operations, the vectorized kernel is able to achieve speedups along all operations. The vectorized kernel also is able to achieve an added cache performance benefit. Since it operates on several array elements at once it can capitalize on the array being loaded into the cache and operate on sequential elements already loaded before they gets flushed to make room for the next operations operand and destination arrays. The JTP and LOOP implementations do not get this luxury. They access elements one at a time causing sequential array elements to often be flushed from cache by loading operands of sequential operations before they can be accessed.

Edge detection is another benchmark where the fully Java vector library was implemented. The hope was that by isolating each of the operations into vectorizable loops the JIT compiler could vectorize them. However, analysis of the JIT output code yielded this not to be the case. The functions were JIT compiled down without using SIMD instructions and yielded worse performance than both the LOOP and JTP kernels.

As with VecAdd and VecCos there is too much data transfer and initial overhead for the GPU to be able to capitalize on the parallelization of the problem. The GPU is does gain ground with each problem size increase but peaks at a speedup of 0.3x for the largest problem size. Therefore EdgeDetect is best performed on a vectorized kernel with the loop kernel as a fallback path if the vectorization library is not present.

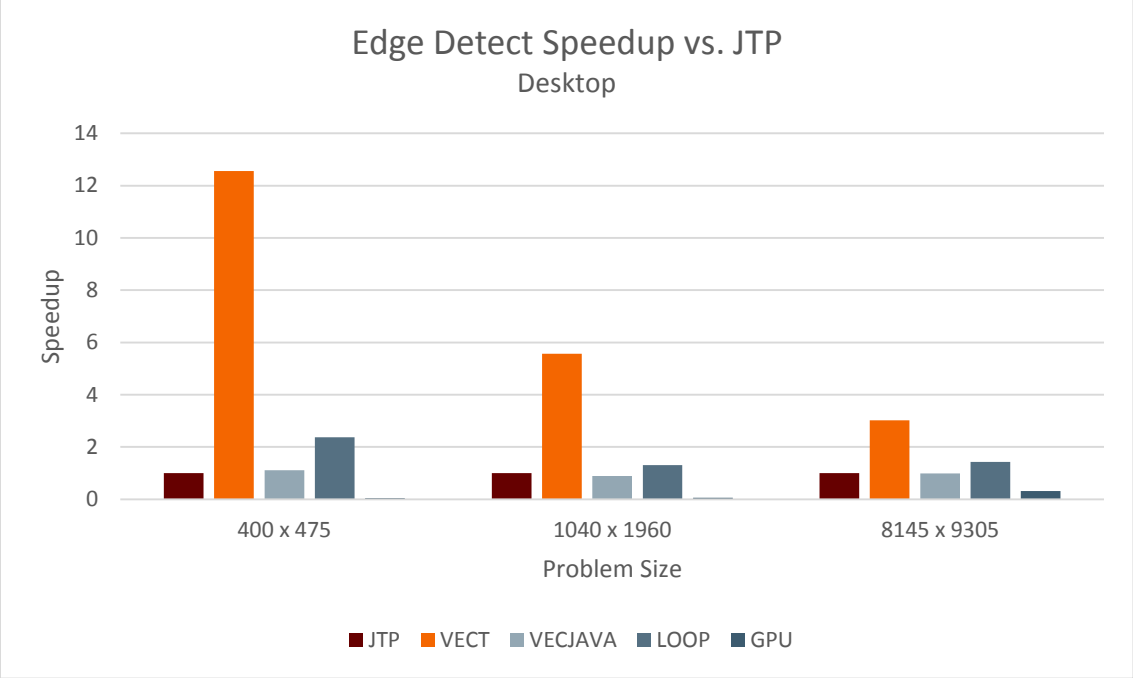


*Figure 51: Edge Detect Speedup vs JTP on Desktop*

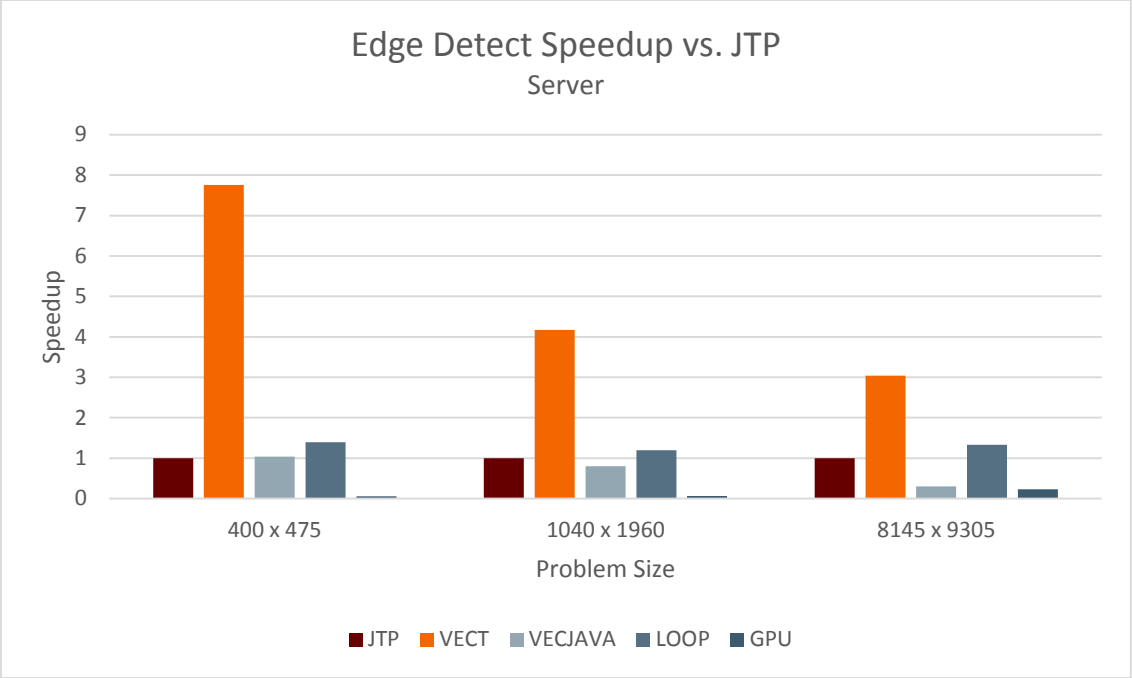


*Figure 52: Edge Filter Speedups vs JTP on Server*

## 6.2.5 Series Test

Series Test comes from the Java Grande Benchmarking Suite's multi-threaded benchmarks [28] and has been converted to Aparapi Java for use in this work. The benchmark calculates the first 1,000,000 Fourier coefficients of the function $f(x) = (x+1)^x$ on the interval (0,2) with each of the Fourier coefficients calculated in a separate kernel. Within each kernel, the benchmark relies heavily on the trigonometric functions to obtain the Fourier coefficients.

The heavy reliance on trigonometric functions enables the vectorized kernel to achieve speedups well beyond those from the SIMD instructions alone. Just like in the VecCos micro-benchmark above, the Series Test benchmark benefits from the hardware implementations of the trigonometric functions. This enables the vector implementation to achieve a 13.4x speedup on the desktop implementation and an 11.4x speedup on the server system.

Due to the large size of the benchmark, the GPU is able to overcome its overheads and surpass the non vectorized CPU kernel implementations in terms of performance. However, due to the large dependence on trigonometric functions the VEC kernel implementation is still able to beat the GPU implementation resulting in a benchmark best run with the vector implementation

with a fallback path of the GPU. As all of the remaining benchmarks consist of a single problem size, all of the remaining results are presented with Desktop and Server results on the dame graph.

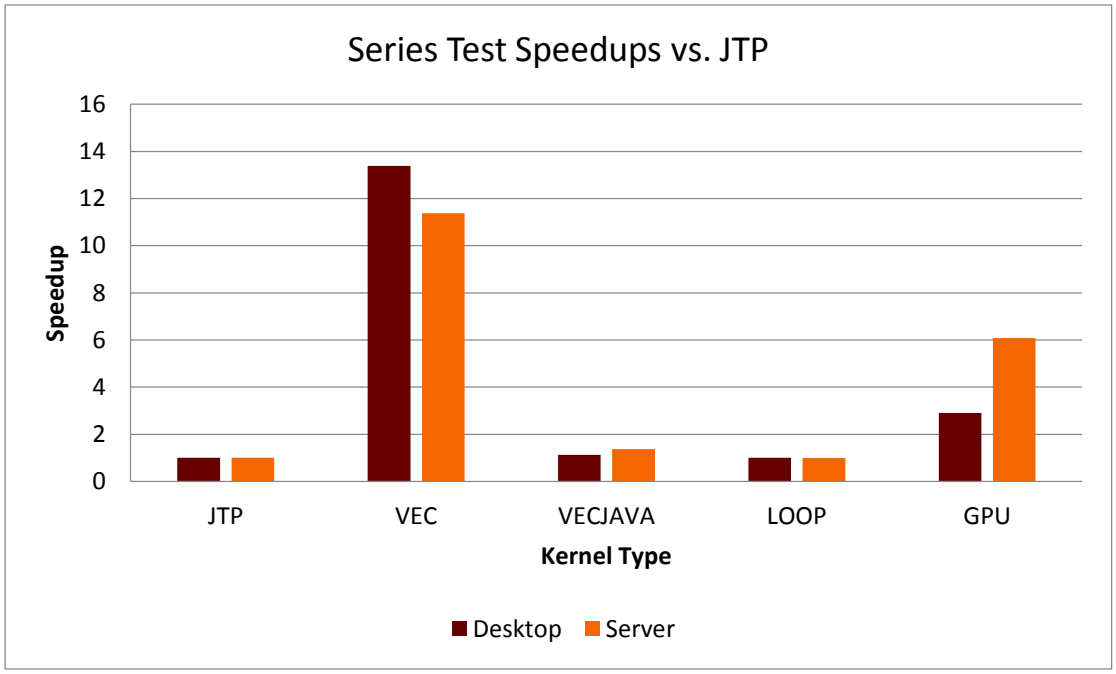


*Figure 53: Series Test Speedups vs. JTP*

A similar work on Java vectorization [32] also implemented a fast Fourier transform benchmark. Theirs was slightly different and came from the SPECjvm2008 benchmark. While the work claims 1.55x and 1.45x speedups on their desktop and server implementations these speedups are only for a small input size and 4 threads on the desktop and 2 threads on the server. Across all sizes and on 4 threads (their desktop max) the desktop only averages a 1.22x speedup across the three input sizes. For the maximum 8 thread server implementation the speedups only average 1.09x with the default and large size implementations seeing almost no speedup at all.

These results are much lower than those presented in this work. Part of the difference comes from the ability of the vector library presented in this work to take advantage of hardware trigonometric SIMD instructions. This allows for part of the 11x to 13x speedup in Figure 53 above. However, even benchmarks presented in this work that do not use the trigonometric functions still see better and more consistent performance numbers.

## 6.2.6 Mouse Tracker

Mouse Tracker was another simple image manipulation benchmark that came with the Aparapi source code. It starts out with a black frame of size 1024 x 1024. When it detects mouse movement, it grabs the x and y pixels of the mouse's location and redraws the picture with a rainbow circle around where the mouse was detected. The frame keeps the last 64 mouse track circles on the image. To do this each kernel performs the calculations to compute the color for one pixel based on its distance from the all of the points in the mouse pointer trail. If it is in a circle radius, it grabs the color from a lookup table associated with its distance from the mouse. Otherwise the pixel is black.

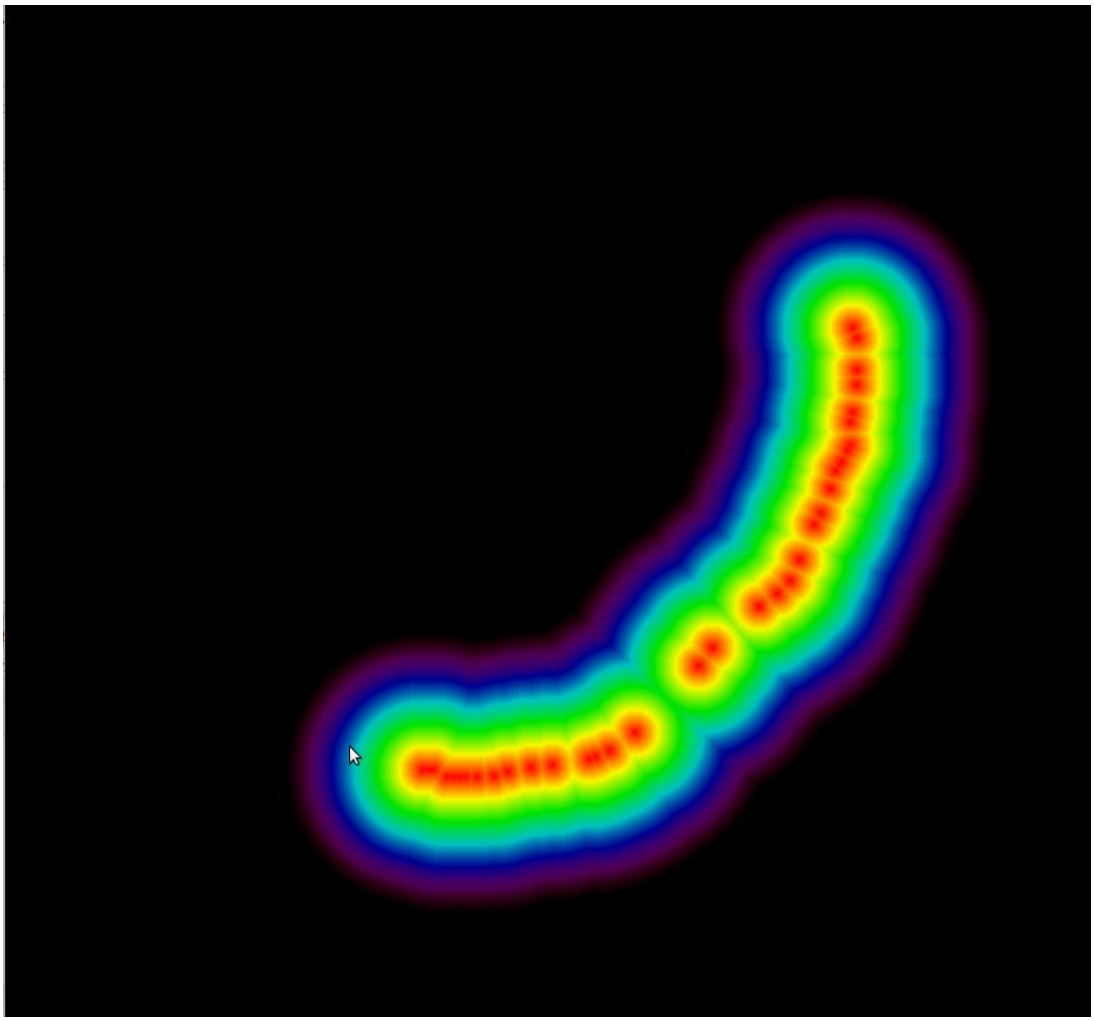

*Figure 54: Mouse Tracker Image*

In order to make this benchmark consistent across multiple runs, the mouse movement detection was removed and replaced with random floats. As the calculation cost for all points within the square is even a random number still provides consistent results without the need to store a long constant array for the path. The mouse tracker was set to calculate 100 points and then report an average execution time for calculating the frame pixels. The speedups versus the standard Java thread pool are reported below in Figure 55.

As this benchmark required little data transfer relative to the number of parallel calculations performed, the GPU implementation came out on top with speedups of 3.1x and 3.9x for the desktop and server implementations respectively. As not all of the internal instructions were vectorizable the vectorized implementation was not able to achieve its usual speedup, but was still able to achieve a 1.4x speedup on the desktop and 1.2x on the server system. This results in the best implementation being the GPU and the fallback path being the VEC implementation.

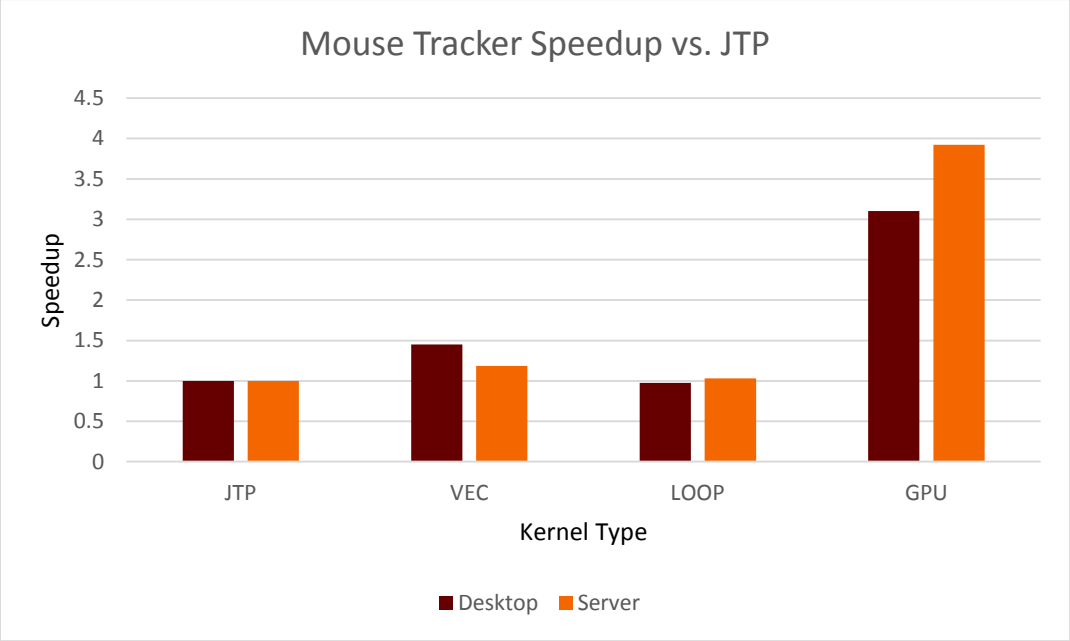


*Figure 55: Mouse Tracker Speedups vs. JTP*

## 6.2.7 Blackscholes

The Black Scholes benchmark also came with the Aparapi source code and is a mathematical model of a financial market calculating a theoretical estimate for the price of European-style financial options. This parallel calculation is run across $2^{20}$ kernels over 5 iterations. After the calculation, the results are verified and then the average execution time across

all iterations is printed. Figure 56 below reports the speedups achieved by each kernel type with respect to the base Java Aparapi thread pool.

While Black Scholes is highly parallel the GPU came up with a slowdown. This is due to the GPU's initial startup overhead. The JTP implementation is able to complete each iteration in around 70 ms and the initial overhead for the first iteration of the GPU implementation is around 780 ms on the desktop implementation. On the CPU side the VEC implementation was able to achieve almost a 2x speedup on the desktop and almost a 1.5x speedup on the server since almost all of the kernel operations were vectorizable calculations. This results in a benchmark with the best implementation being the VEC kernel and a fallback path of the LOOP kernel.

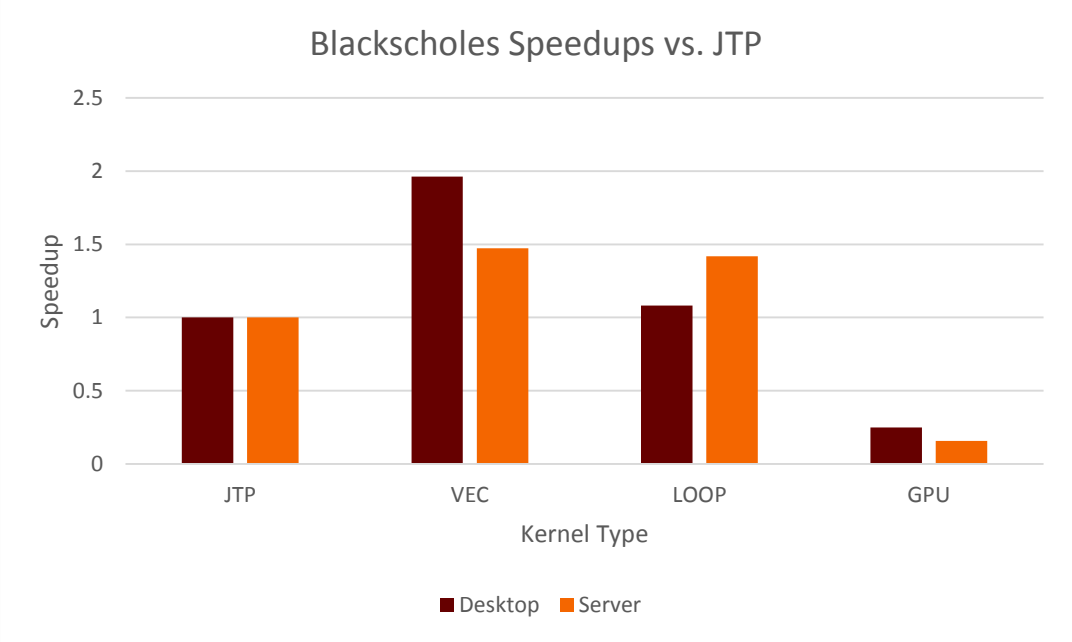


*Figure 56: Blackscholes Speedups vs. JTP*

## 6.2.8 Mandelbrot

Mandelbrot is another benchmark provided by the Aparapi source code. It generates a Mandelbrot fractal and then zooms in and out on any point of the fractal that is clicked. To turn this into a repeatable benchmark, mouse click portion of the code was replaced with the coordinates of a constant point to zoom in and out on. While this benchmark was well suited for Aparapi and the GPU and other works have cited Mandelbrot as a good example of vectorization speedup [1], the majority of the benchmark, is not vectorizable across kernels. Most of the computation time for the Mandelbrot fractal is spent inside a while loop whose length varies from pixel to pixel. Since the loop will execute a different number of times for each kernel, code within this loop

cannot be vectorized across kernels. The auto-vectorizer tool will vectorize a the instructions outside of this loop, but since the majority of the execution time in each kernel is in an un-vectorizable while loop, the majority of the vector kernel is no different from the loop kernel and as such yielded almost identical results. However, the benchmark was still used as an example of a benchmark that was saw speedups by offloading the calculations to the GPU.

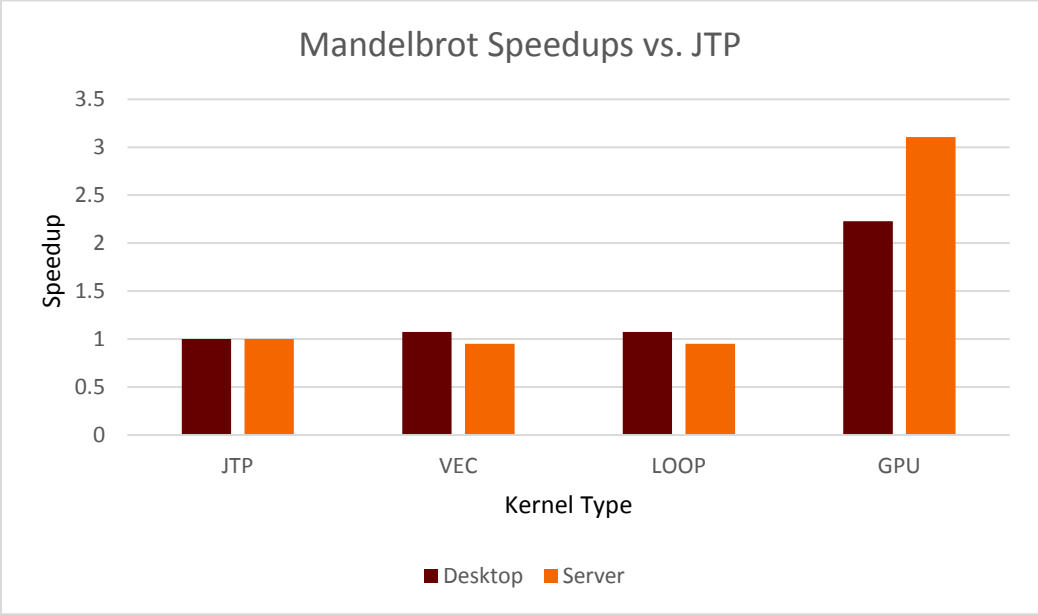


*Figure 57: Mandelbrot Speedups vs. JTP*

## 6.2.9 Life

The Aparapi source code also provided an implementation of the game of life in an Aparapi source kernel. The benchmark initially draws a line across the middle of the screen and these pixels are deemed alive and the rest are dead. From there each pixel counts the number of neighboring pixels that alive. If that number is 3, or the number is 2 and the pixel was alive to start with the pixel is considered alive for the next generation. Otherwise the pixel is considered dead. Edge pixels are treated differently and their states remain constant from generation to generation. In the Aparapi implementation, each pixel is calculated with its own kernel.

The auto-vectorizer tool was unable to vectorize the life benchmark due to it treating different pixels. Because of this, the tool could not guarantee that all kernels implemented in a vector underwent the same calculations and therefore could not vectorize the kernel. For a more detailed explanation see section 5.3. However, like Mandelbrot, this benchmark was still included in this work to show another Aparapi benchmark that gains speedups from offloading calculations to the GPU and to show that even though the auto-vectorizer was unable to vectorize the kernel it was still able to achieve a performance improvement over the default JTP implementation.

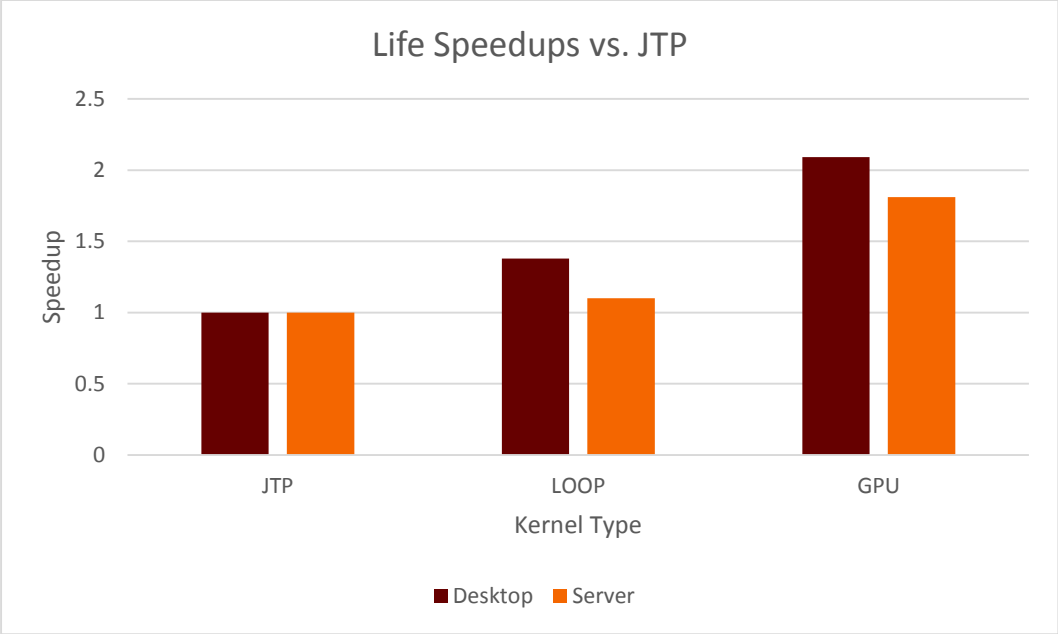


*Figure 58: Life Speedups vs. JTP*

## 6.2.10 Kmeans

Kmeans is a well-known benchmark that was converted from an OpenCL implementation in the Rodinia benchmark suite [31] to an Aparapi benchmark for this work. While this benchmark is highly parallel, it ended up not being a good fit for either the VEC or GPU cases. Although a majority of the operations within a kmeans kernel are on arrays, the elements of the array accessed are not sequential for sequential kernels. This inhibits the operations from being vectorized. Kmeans also contains control flow logic that causes different kernels to implement different calculations further disrupting the vectorizability of the kernel. In the end the auto-vectorizer was unable to vectorize any of the calculations and the output of the auto-vectorizer was the same as the loop kernel. The GPU implementation was also unable to compete due to the small runtime of the problem and the large use of control logic.

This benchmark was included to illustrate that even when the auto-vectorizer is unable to vectorize a kernel, it still outputs a looped implementation that is often better than the normal JTP implementation alone. For this particular benchmark the best results were achieved when the kernel was run in a LOOP configuration due to its reduced kernel overhead.

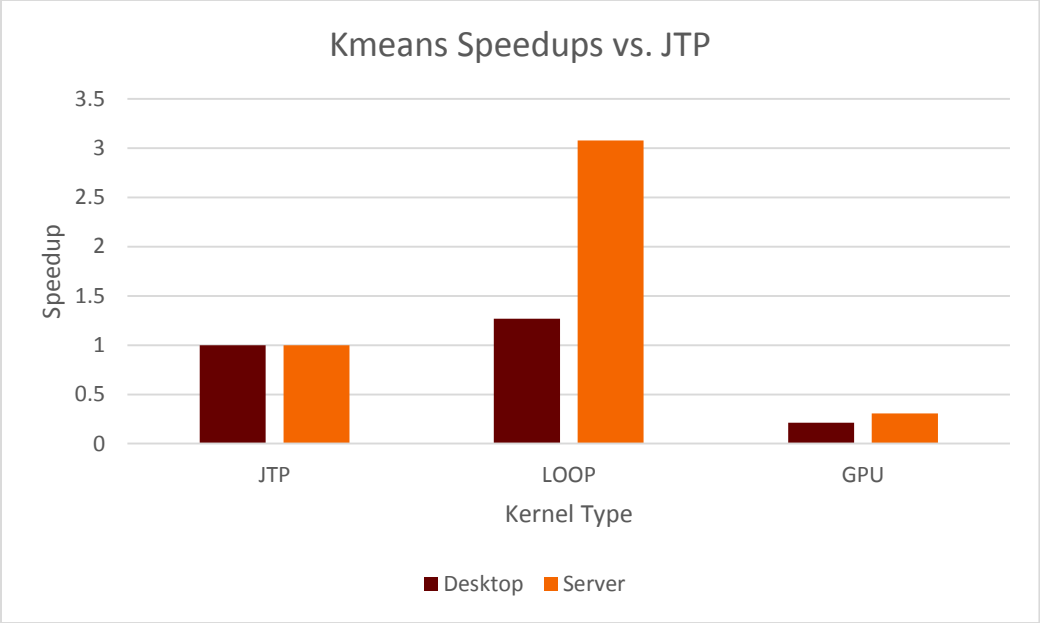


*Figure 59: Kmeans Speedup vs. JTP*

# 6.3 Vector & Loop Size

There were multiple factors that came into play when determining the optimal vector size and loop size. For both the VEC and LOOP kernel implementations a balance had to be met between overhead and cache performance. For vector size it was a balance of minimizing the JNI overhead by increasing the vector size and therefore reducing the number of JNI library calls with finding the optimal vector size to enable good cache performance. However, after numerous tests on varying vector sizes across several benchmarks, all of them yielded better performance for larger vector sizes. It turns out that the overhead from the JNI library call is a much larger factor. As Figure 60 illustrates, the performance continues to rise until a vector size of 8192. At this point the overhead per operation has been reduced to a point where it begins to become nominal compared to the cost of each operation.

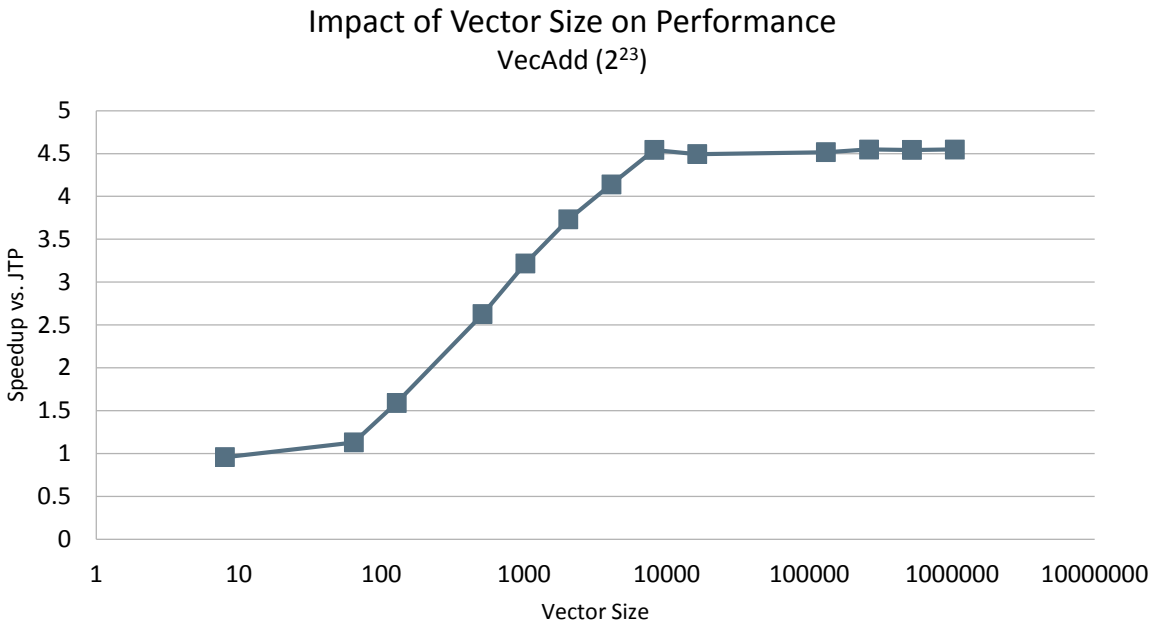


*Figure 60: Impact of Vector Size on Performance of Vec Add*

However, the optimal loop size did come at a balance point between reducing overhead and maintaining good cache performance. The shape in Figure 61 was common among all of the benchmarks with each benchmark achieving similar performance results up until the maximum possible loop size for the benchmark. VecAdd's speedup curve was chosen to show below because the benchmark could implement loop sizes large enough to illustrate the full performance curve of varying loop sizes.

For the beginning of the curve, the LOOP implementation achieves better performance over the JTP implementation due to two factors. First each kernel undergoes less and less kernel creation overhead per iteration as more and more iterations are removed from separate kernels and added to the loop of the LOOP kernels. Second each kernel is writing to areas of memory that are farther and farther apart. As each kernel is responsible for a larger section, it becomes less and less likely that two kernels will be writing to the same area of memory at once therefore reducing the overhead of keeping the cache coherent between the cores. However there is a limit to this performance improvement. As the overhead per iteration decreases and the number of iterations each kernel is responsible for grows cache performance becomes a factor. The peak comes at a loop size of 4096 which is the number of floats or integers that can be stored in the 16KB L1 data cache of the desktop processor. Because of this peak, the auto-vectorizer was set to output loop sizes as close to 4096 as possible without going over whenever it was unable to produce a vectorized implementation. However, this limitation is a constant defined at the top of the file that can be easily modified for operations on different processors.

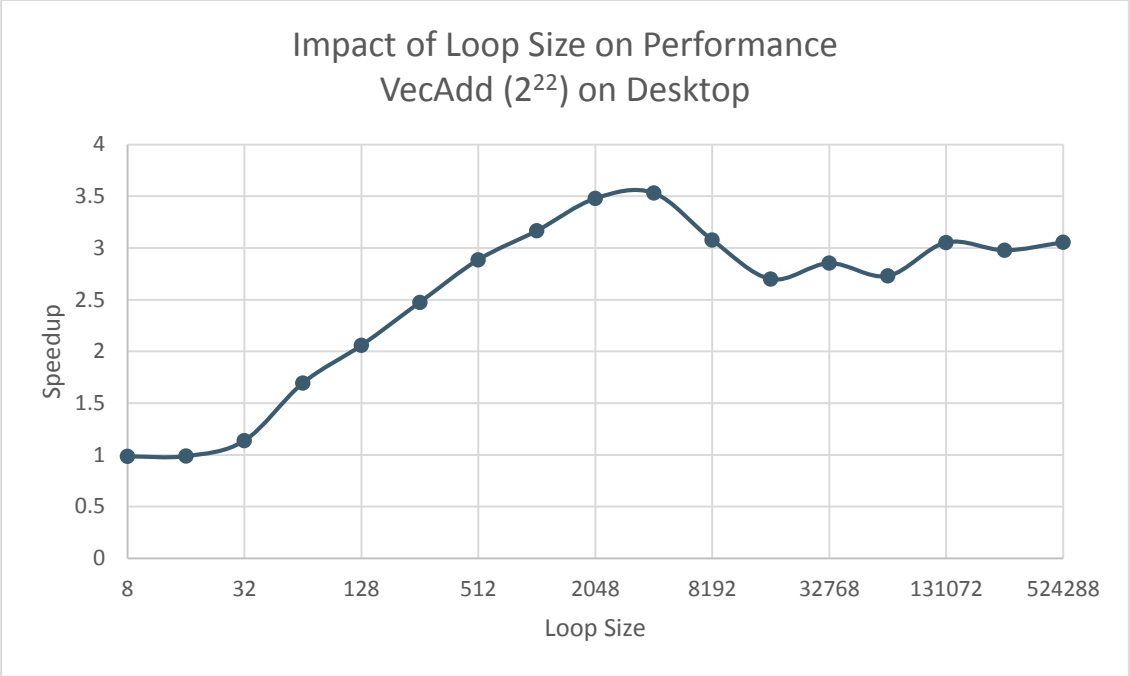


*Figure 61: Impact of Loop Size on Performance for VecAdd ($2^{22}$)*

# 6.4 Handling Remainder Kernels

When reducing kernels either for vectorization or for looping, not all kernels reduce nicely. If the kernel size is a prime number or is not a multiple of the desired vector or loop size, then something must be done with the remaining kernels. In order to handle this task, this work looked at two different approaches. The first of which involved calculating the kernels that were an even multiple of loop or vector size first and then running the remaining kernels as individual JTP kernels after. This method is represented in Figure 62 below as Method 1. As you can see the speedups obtained through vectorization are quickly lost when remainder kernels need to be executed. This is because that while the vector or loop kernels gain the performance benefits of being JIT compiled due to multiple executions, the remainder kernels are executed at most 7 times on the desktop machine. This means that these 7 kernels will be completely interpreted and run much slower than the JIT compiled code, see Figure 41 for the exponential decrease in exectution time as the code is JIT compiled.

Method 2 seeks to remedy this problem by eliminating the need for extra kernels. Instead the vector or loop size for the first remainder kernels is increased by 1. For example if there are 8,195 kernels to execute on 8 threads, the vector or loop size of the first 3 kernels will be 1025 and the remaining 5 kernels will have a vector or loop size of 1024. This lets the C++ vector library handle vectorizing the iterations of the loop possible and calculating the remaining iterations on

their own for the vector case. As can be seen in Figure 62 below, this yielded much better performance. The speedup lost was greatly reduced and in most cases was smaller than the average variation of the benchmark times. Because of this, Method 2 was selected and implemented in the auto-vectorization tool and all of the benchmarks in this work.

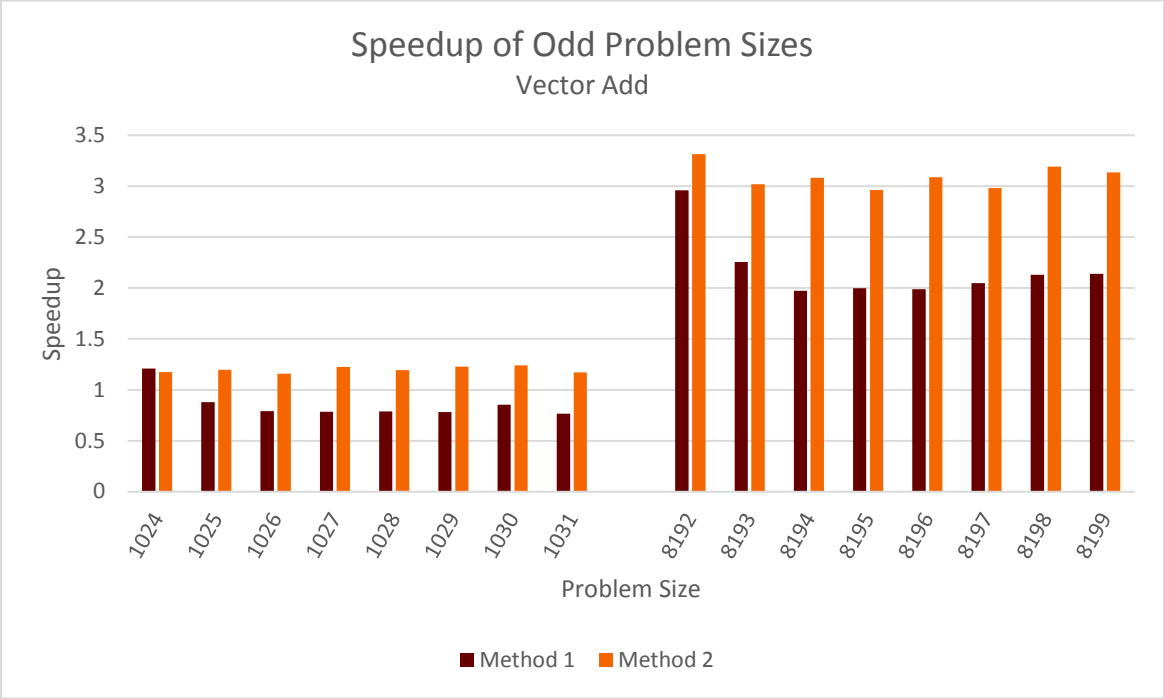


*Figure 62: Speedup of Odd Problem Sizes*

# Chapter 7

# Conclusions

In this thesis, we presented a vector library that enables Java to take advantage of SIMD instructions as well as an auto-vectorizer that automatically utilizes this vector library in existing Aparapi Java source code. In creating the vector library, it was discovered that GCC's auto-vectorizing optimizations are just as good as hand coded assembly. This enabled the vector library to be written in C++ allowing for easier implementation as well as a simpler library for anyone else to read and understand, all while still maintaining full SIMD performance enhancements.

In implementing some of the more complex SIMD functions such as sin, and cos, this work found out that Java's Math library does not fully utilize hardware equivalent operations. In the Java Math library documentation it states that it often relies on software implementations of these functions. This allowed the SIMD vector library to achieve even greater performance enhancements as its SIMD hardware implementations of these functions are competing against software implementations that consist of several more instructions. This work did not find any cases where the results of these two differed, but left these functions as optional in the auto-vectorizer in case a later user found a case where they differed enough to affect their program.

In implementing the vector library this work also discovered the huge overhead costs of variable data transfers between Java and C++. Passing variables back and forth between Java and C++ creates more overhead than can be overcome by SIMD performance enhancements and the first implementation of the vector library was unable to achieve any speedups. However, through the use of a NIO buffer these overheads can be almost completely eliminated. Both C++ and Java can access data in a NIO buffer in the same amount of time it takes either of them to access an array and there is only a slight overhead penalty paid on the C++ side as it loads the buffer. This enabled the final implementation of the vector library to achieve desirable performance enhancements for all vectorizable operations.

In this work, we explored the impact of loop and vector size on the performance of their associated Aparapi kernels. From this analysis we were able to determine that vector size should

always be maximized, but loop size has a peak optimal size. This size is dependent on the L2 cache and occurs when loop size is equivalent to the number of variables that fit into the L2 cache.

This work also presents an auto-vectorizer for Aparapi code that enables SIMD performance enhancements through the vector library to be utilized without any additional effort by an Aparapi coder. The speedups achieved by the tool are often related to the number of vectorizable options in the Aparapi kernel. For kernels that are fully vectorizable, the auto-vectorizer tool is able to achieve speedups of between 4x and 5x on the desktop implementation and 1.5x to 2x on the server implementation. However, not all kernels are completely vectorizable. Some kernels have control flow logic that can reduce the number of vectorizable operations or even eliminate the possibility of auto-vectorization altogether.

However, the looped implementation, output by the auto-vectorizer when vectorization was not possible, was still able to beat the default JTP kernel in all 10 of the desktop benchmarks and 9 out of 10 of the server benchmarks. The only server benchmark that did not experience some sort of speedup with the auto-vectorizer outputted implementation, was Mandelbrot which was only slowed down to 0.98x of the original execution time. The vectorization speedups were even good enough to beat the GPU implementation in 6½ out of the 10 benchmarks. Matrix multiply on the desktop implementation performed best under the VEC implementation but performed best with the GPU implementation on the server and was therefore awarded a ½ victory.

In conclusion this work further improves Java and the Aparapi programming extensions in three main ways. First this work brings one of the first major implementations of SIMD instructions in Java. This enables Java programmers to take advantage of these performance data parallel instructions for the first time. Second through the use of the Aparapi API and the auto-vectorization tool, these performance enhancements can be had by an Aparapi programmer without any additional programming effort. Third this work improves Aparapi's default Java fallback path of the Java thread pool with both the looped and vectorized kernel implementations. Experimentation even revealed the loop or vectorized implementation even outperformed the GPU kernel and provides the fastest implementation of the kernel for relatively short programs (less than 2 seconds), programs with high data transfers and low computational complexity, or programs that involve a large number of Java Math library functions.

# Chapter 8

# Suggestions for Future Work

While this work does provide the fastest implementation of several of the benchmarks through the vector library, there is room for improvements. The current vector library implementation requires the use of JNI instructions which add additional overhead. This overhead requires at least 1000 operations to be conducted by the vector library function before the overhead could be overcome and any significant performance gain could be achieved. In order to avoid these overheads the JIT compiler could be rewritten to understand these function calls and JIT compile them directly to SIMD instructions within the Java application itself instead of calling a separate C++ library function. This would also reduce the need to perform multiple of these operations and the performance benefit could be seen in as little as 4 vectorizable operations. However, reconfiguring a JIT compiler is a very large engineering task and would require the modified SIMD capable JIT at every runtime in order to achieve these performance benefits. Therefore this work focused on proving the performance gain possibilities of implementing SIMD instructions in Java and left JIT integration for future work. While the SIMD capable JIT is being created and until SIMD instructions become common in all Java JIT compilers, this work can be used as a fallback for any non-SIMD enabled JIT compilers.

The current implementation of the vector library also does not allow for chaining of vector operations. Since the vector library implements SIMD operations as separate functions, each instruction must read data into the special SIMD registers before computation and back to memory after. An ideal implementation implemented directly in the JIT would recognize back to back vectorizable instructions and be able to leave data in the special registers and therefore reducing the loading and storing overheads and further enhancing the speedups possible through SIMD instructions.

Another area for future work is within the auto-vectorizer itself. Currently it cannot vectorize code within conditional statements whose result varies from kernel to kernel. Discussed in detail in section 5.3, there are instances where code that is vectorizable is left un-vectorized by

the auto-vectorizer. The auto-vectorizer could be improved to spot these sections and split them into vectorizable and non-vectorizable sections just like a hand vectorized implementation would do.

# References


[1] I. Landwerth, ".NET Framework Blog," Microsoft, 7 April 2014. [Online]. Available: http://blogs.msdn.com/b/dotnet/archive/2014/04/07/the-jit-finally-proposed-jit-and-simd-are-getting-married.aspx. [Accessed 8 April 2014].

[2] "JDK-6340864 : Implement vectorization optimizations in hotspot-server," Oracle, 12 November 2013. [Online]. Available: http://bugs.java.com/view_bug.do?bug_id=6340864. [Accessed 27 February 2014].

[3] J. P. Lewis and U. Neumann, "Performance of Java versus C++," University of Sourhtern California, January 2004. [Online]. Available: http://www.scribblethink.org/Computer/javaCbenchmark.html. [Accessed 28 January 2014].

[4] J. Valdivia, "Java vs C++ Performance Comparison, JIT Compilers, Java Hotspot & C++ Native Compiler," codex[pi], 7 January 2014. [Online]. [Accessed 4 February 2014].

[5] G. P. Nikishkov, Y. G. Kikishkiv and V. V. Savchenko, "Comparison of C and Java performance in finite element computations," *Computers and Structures,* vol. 81, pp. 2401-2408, 2003.

[6] C. Felde, "C++ vs Java performance: It's a tie!," 27 June 2010. [Online]. Available: http://blog.cfelde.com/2010/06/c-vs-java-performance/. [Accessed 4 February 2014].

[7] J. M. Bull, L. A. Smith, L. Pottage and R. Freeman, "Benchmarking Java against C and Fortran for Scientific Applications," in *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, Palo Alto, California, USA, 2001.

[8] D. Rosenberg, "Hadoop breaks data-sorting world record," Cnet, 15 May 2009. [Online]. Available: http://www.cnet.com/news/hadoop-breaks-data-sorting-world-records/. [Accessed 13 April 2014].

[9] Intel, "Multicore: The Software View," 2007. [Online]. Available: http://download.intel.com/corporate/education/emea/event/af12/files/cownie.pdf. [Accessed 13 April 2014].

[10] S. El-Shobaky, A. El-Mahdy and A. El-Nahas, "Automatic vectorization using dynamic compilation and tree pattern matching technique in Jikes RVM," *Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS '09)*, pp. 63-69, 2009.

[11] J. Nie, B. Cheng, S. Li, L. Wang and X.-F. Li, "Vectorization for Java," *Network and Parallel Computing*, vol. 6289, pp. 3-17, 2010.

[12] Nvidia, "Tesla GPU Accelerators for Servers," Nvidia, 2014. [Online]. Available: http://www.nvidia.com/object/tesla-servers.html. [Accessed 15 March 2014].

[13] NVIDIA, "What is CUDA?," NVIDIA, 2014. [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html. [Accessed 18 April 2014].

[14] I. Buck and P. Hanrahan, "Data Parallel Computation on Graphics Hardware," 2003.

[15] Oak Ridge Natinoal Laboratory, "Introducing Titan," Oak Rudge National Laboratory, 2012. [Online]. Available: https://www.olcf.ornl.gov/titan/. [Accessed 18 April 2014].

[16] Khronos Group, "OpenCL: The open standard for parallel programming of heterogeneous systems," Khronos Group, 2014. [Online]. [Accessed 18 April 2014].

[17] Nvidia, "Cuda C Programming Guide," Nvidia, 13 February 2014. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. [Accessed 21 April 2014].

[18] R. Tsuchiyama, T. Nakamura, T. Iizuka, A. Asahara, J. Son and S. Miki, "The OpenCL Programming Book," FixStars, 26 April 2012. [Online]. Available: http://www.fixstars.com/en/opencl/book/. [Accessed 18 April 2014].

[19] "What is Aparapi?," 19 May 2014. [Online]. Available: https://code.google.com/p/aparapi/. [Accessed 8 April 2014].

[20] E. R. Altman, J. S. Auerbach, D. F. Bacon, I. Baldini, P. Cheng and S. J. Fink, "The Liquid Metal Blokus Duo Design," in *International Conference on Field-Programmable Technology*, 2013.

[21] IBM, "Liquid Metal," IBM, 2014. [Online]. Available: http://researcher.watson.ibm.com/researcher/view_project.php?id=122. [Accessed 28 April 2014].

[22] A. Stromme, R. Carlson and T. Newhall, "Chestnut: A GPU Programming Language for Non-Experts," in *International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM '12)*, 2012.

[23] P. C. Pratt-Szeliga, J. W. Fawcett and R. D. Welch, "Rootbeer: Seamlessly Using GPUs from Java," *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*, pp. 375-380, 2012.

[24] G. Frost, "Aparapi: An Open Source tool for extending the Java promise of 'Write Once Run Anywhere' to include the GPU," *Open Source Convention (OSCON)*, 2012.

[25] AMD, "Aparapi," AMD, 2014. [Online]. Available: http://developer.amd.com/tools-and-sdks/heterogeneous-computing/aparapi/. [Accessed 21 April 2014].

[26] P. Pratt-Szeliga, "Rootbeer," Github, 8 December 2013. [Online]. Available: https://github.com/pcpratts/rootbeer1. [Accessed 8 December 2013].

[27] T. D. Han and T. S. Abdelrahman, "Reducing Branch Divergence in GPU Programs," *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, pp. 1-8, 2011.

[28] R. Allen and K. Kennedy, Optimizing Compilers for Modern Architectures, San Diego, CA: Academic Press, 2002.

[29] Oracle Corporation, "Class Math," Oracle Corporation, 2014. [Online]. Available: http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html. [Accessed 20 April 2014].

[30] Oracle Corporation, "JSR 51: New I/O APIs for the Java Platform," Oracle Corporation, 2014. [Online]. Available: https://www.jcp.org/en/jsr/detail?id=51. [Accessed 16 February 2014].

[31] Oracle Corporation, "New I/O APIs," Oracle Corporation, 2010. [Online]. Available: http://docs.oracle.com/javase/1.5.0/docs/guide/nio/index.html. [Accessed 15 March 2014].

[32] J. V. Gesser, "Java 1.7 parser and Abstract Syntax Tree," 13 April 2014. [Online]. Available: https://github.com/matozoid/javaparser.

[33] J. Gosling, "Transcendental Meditation," James Gosling: on the Java Road, 27 January 2005. [Online]. Available: https://blogs.oracle.com/jag/entry/transcendental_meditation. [Accessed 16 March 2014].

[34] R. Sedgewick and K. Wayne, "EdgeDetector," Princeton, 9 February 2009. [Online]. Available: http://introcs.cs.princeton.edu/java/31datatype/EdgeDetector.java.html. [Accessed 12 December 2013].

[35] EPCC, "The Java Grande Forum Multi-threaded Benchmarks," [Online]. Available: http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/threads.html. [Accessed 15 January 2014].

[36] University of Virginia, "Rodinia:Accelerating Compute-Intensive Applications with Accelerators," 13 August 2013. [Online]. Available: https://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Main_Page. [Accessed 20 January 2014].

[37] E. Holk, W. Byrd, N. Mahajan, J. Willock, A. Chauhan and A. Lumsdaine, "Declarative Programming for GPUs," in *International Conference on Parallel Computing (parCo 2011)*, 2011.

[38] S. Audet, "Do any JVM's JIT compilers generate code that uses vectorized floating point instructinos," Stackoverflow, 22 June 2013. [Online]. Available: http://stackoverflow.com/questions/10784951/do-any-jvms-jit-compilers-generate-code-that-uses-vectorized-floating-point-ins. [Accessed 2 December 2013].