Applying Source Level Auto-Vectorization to Aparapi Java

Curt Albert Alastair Murray Binoy Ravindran

Dept. of Electrical and Computer Engineering,
Virginia Tech, Virginia, USA
{falbert9,alastair,binoy}@vt.edu

Abstract

Parallelism dominates modern hardware design, from multi-core CPUs to SIMD and GPGPU. This bring with it, however, a need to program this hardware in a programmer-friendly manner. Traditionally, managed languages like Java have struggled to take advantage of data-parallel hardware, but projects like Aparapi provide a programming model that lets the programmer easily express the parallelism within their code, while still programming in a high-level language.

This work takes advantage of this programmer-specified parallelism to perform source-level auto-vectorization, an optimization that is rarely considered in Java compilation. This is done using a source-to-source auto-vectorization transformation on the Aparapi Java program and a JNI vector library that is pre-compiled to take advantage of available SIMD instructions. This replaces the existing Aparapi fallback path, for when no OpenCL device exists or if that device has insufficient memory for the program.

We show that for all ten benchmarks tested the auto-vectorization tool produced an implementation that was able to beat the default Aparapi fallback path by a factor of 4.56x or 3.24x on average for a desktop and a server system respectively. In addition it was found that this improved fallback path even outperformed the GPU implementation for six of the ten benchmarks.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

General Terms Languages, Performance

Keywords Auto-Vectorization, Aparapi, Java, GPGPU, OpenCL, SIMD, Parallel

1. Introduction

As individual cores on a processor have reached their clock frequency limits, chip makers and programmers have had to turn to parallelism to find performance improvements. These parallel enhancements, however, require programmers and toolchains to support programming across multiple cores, to exploit single-instruction multiple-data (SIMD) or vector instructions, or even to go as far as using hundreds of cores in massively data-parallel general-purpose GPU (GPGPU) architectures.

PPPJ '14, September 23 - 26 2014, Cracow, Poland.
Copyright © 2014 ACM 978-1-4503-2926-2/14/09…$15.00.
http://dx.doi.org/10.1145/2647508.2647519

1.1 Language Support for Data Parallel Computations

Neither expecting the programmer to manually specify how to use available parallelism, nor expecting the toolchain to automatically extract parallelism from serial code has been successful. Modern parallel architectures are most commonly exploited by using parallel programming languages or language extensions. Multi-core parallelism can be programmed through high-level parallel extensions such as OpenMP, Intel Threading Building Blocks or Cilk. SIMD/vector instructions can be automatically exploited by auto-vectorizing compilers with no programmer intervention, but accessing greater levels of data-parallelism on GPGPU architectures currently requires the programmer to use low-level parallel programming models such as OpenCL [8] or CUDA [11].

OpenCL and CUDA enable a programmer to offload parallel calculations to a GPGPU, but require complex, time-consuming and error-prone programming techniques. Therefore these languages have seen limited acceptance outside the realm of high performance computing. Much of modern programming happens using managed languages such as Java or C#, there has been recent work abstracting away the complexity of GPGPU programming by by extending these managed languages. For example, Aparapi [2] and Rootbeer [17] both bring GPGPU programming to the Java language and Habanero-Java [6] extends the Java language with parallel constructs.

SIMD performance enhancements have been easier for most programmers to take advantage of. Many compilers have adopted SIMD instructions in their normal optimization process. Compilers for natively compiled languages, such as C and C++, already utilize them as part of their standard optimizations. Managed languages, however, generally do not exploit these instructions except in the most minimal ways. Both the runtime JIT-compiled nature of these languages and the high-level structure of programs written in them have meant that auto-vectorization optimizations have not been implemented. The optimizations are expensive to perform, making them unfriendly in a JIT-compiled scenario. Also, they only successfully apply to naturally data-parallel programs, so they often failed to provide any benefit for high-level programs that are heavily abstracted from the underlying architecture.

This work, however, takes Java code that has been written to the Aparapi API and exploits this programmer provided parallel structure at the source-level to successfully exploit SIMD instructions without JIT modification.

1.2 Aparapi

Aparapi's ("A PARallel API") API [2] was developed as a way to express parallel sections of code in Java so that they can be run on any OpenCL capable device. This extends Java's write-once run-anywhere model to include any OpenCL capable hardware.

Aparapi is programmed through the use of a kernel method that will be instantiated thousands of times in parallel. This kernel can be inline with the rest of the source code, it just has to meet the constraints of the Aparapi API. At runtime the Aparapi library checks to see if an OpenCL device is available to run the kernel. If there is no such device then Aparapi falls back to running the kernel in parallel using a pool of Java threads.

1.3 Contributions

This work brings the data parallel performance enhancements of SIMD instructions to Java through a source-level auto-vectorizer and a simple vector library. To handle the automatic utilization of the vector library, this work presents an auto-vectorizer that works with the high-level Java Aparapi API. Aparapi brings the performance benefits of massively parallel GPU hardware to the Java programmer without forcing them to learn to write 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 so that they may run on a GPU. The auto-vectorizer takes advantage of this knowledge of independence to aid it in determine sections of code that are safe to vectorize, and to correctly utilize the vector library without any additional effort from the programmer.

As Java JIT compilers cannot reliably take advantage of the performance of SIMD instructions, this work presents a C++ library that is accessed by Java through JNI calls to perform SIMD calculations. The auto-vectorization tool uses this library to enable access to SIMD hardware without JIT-compiler support.

This work is not only evaluated against the vanilla Aparapi Java Thread Pool and OpenCL implementations, but against a partial implementation of our technique that bundles parallel work together but without vectorization. We find that while performing this bundling improves performance over the Java Thread Pool, the addition of vectorization makes our technique competitive with the OpenCL implementation running on a GPU for many benchmarks. Even when auto-vectorization is not able to out-perform a GPU, our technique always has better performance than the Java Thread Pool fallback path.

2. Related Work

Although previous work has not looked at the intersection of Java auto-vectorization and using high-level languages to program GPGPUs, both of these topics have been investigated separately.

2.1 JIT Auto-vectorization

Nuzman et al. [10] have previously described an effective approach to introducing auto-vectorization into JIT compilers through a split offline/online approach. Like this work, they use GCC's auto-vectorizer offline. They, however, produce vector bytecode instructions that will be translated into SIMD instructions by the online JIT component. This means that the bytecode can be run on newer JITs and exploit instructions that did not exist at the time of compilation, but the JIT must be updated for every new architecture. In contrast, our approach only requires recompiling a C++ library to target a new architecture, but does not provide a mechanism for targeting future hardware within an existing binary. The main advantage of our approach over that of Nuzman et al. is that we are specifically exploiting the parallel structure of Aparapi, rather than being limited to traditional loop-based vectorization.

Nuzman et al. [10] targeted C code running on the Mono CLR, however there has been other works that directly target auto-vectorization in Java. Shobaky et al. [3] target subword-SIMD functionality within Java. They do 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 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.

Nie et al. [9] avoid the small data size limitations of sub-word SIMD. They create an automatic vectorizer that is built into the Harmony JVM's JIT compiler: Jitrino. The auto-vectorizer is able to vectorize Java loops of any size, unlike the previous work's [3] single instruction loop limitation. The work also supports a much larger selection of SSE and AVX instructions to bring true vectorization to Java. The work sees speedups of 2x on sub-benchmarks within SPECjvm2008 and even. The work achieves 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 drastically as the number of threads and data input size increases. This means that a multi-threaded approach can often provide a larger performance gain than a single-thread SIMD approach.

2.1.1 High-level GPGPU Languages

Programming GPGPUs often involves complex low-level data transfers and code complexity that the average programmer does not wish to deal with. In order to make general purpose GPU programming more accessible, extensions to current languages and entirely new domain specific languages and have been created.

Chestnut [19] and Harlan [7] are both new domain specific languages created just for this reason. They seek to abstract all of the complexities associated with GPGPU computing away from the programmer and provide a simple to use parallel API that allows programmers to write high-level parallel code that can be run across a variety of different parallel architectures.

However, since new domain specific languages require a programmer to port existing code to a new language, this work focuses on extensions to Java that allow parallel programming models to be implemented inside existing code allowing for minimal code modification. Aparapi [2] and Rootbeer [17] are both extensions to the Java programming model that are designed to provide the programmer with a simple API to declare certain sections of their code parallel. The extension runtime then takes these sections of code and compiles them down to low level code that can run on a GPU, Aparapi uses OpenCL and Rootbeer uses CUDA. Habanero-Java [6] provides a deeper level of expressibility by providing language extensions to support parallelism and to safely preserve Java exception semantics when the underlying GPGPU hardware follows a different execution model.

Because Habanero-Java syntactically and semantically extends the language it is more difficult to integrate its use into an existing project, Aparapi and Rootbeer merely provide a new API. Of these last two, Aparapi's use of OpenCL means that it can run on any OpenCL device, including CPUs and most GPGPUs, whereas Rootbeer's use of CUDA means that it can only run on NVIDIA GPUs. Therefore Aparapi was used in this work, as it provides the most heterogeneous flexibility.

3. 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 1 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.

Normal Instruction	Vector Instruction			
101010	0...101101	0...101100	0...101011	0...101010
+ 101010	+ 0...101101	0...101100	0...101011	0...101010
1010100	0...1011010	0...1011000	0...1010110	0...1010100

Figure 1. SIMD example.

3.1 Vector Library

Java is still at the early stages of supporting SIMD instructions. Although the Hotspot JIT compiler has the capability to vectorize extremely simple loops [14], in practice anything more complicated than a sequential loop containing an array addition is not vectorized. Any time 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.

3.1.1 Vector Library Implementation

As a first step to bringing SIMD instruction support 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 each vector operation was written as a basic C++ loop compiled to SIMD instructions through GCC's auto-vectorization compiler optimizations. The GCC compilation handles odd sized problems by vectorizing the loop iterations that fit into vectors and then individually calculating the remaining elements. Writing the library in C++ obviously limits the natural portability of Java, but by using generic C++ rather than specialized vector intrinsics this library can be compiled for any architecture with a C++ compiler.

Due to JNI data transfer overheads, 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 has 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 in Figure 3. 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 3 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.

3.1.2 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 `java.lang.StrictMath` which implements the function in software to ensure accuracy of the result [15]. 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.

3.1.3 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 [16]. 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 [16].

In the vector library implementation, all of the variables are passed through the use of a single `java.nio.ByteBuffer`. Figure 2 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 2 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.

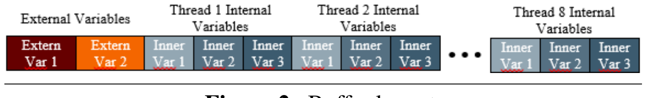


Figure 2. Buffer layout.

Packing all data into a single buffer minimizes NIO overhead. When passing the arguments to the vector library function in separate buffers there is a small overhead for each `GetDirectBuffer-Address()` 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 mininal amount of time compared to the `GetDirectBufferAddress()` load.

```
/******************************************************************************
* Name: addXIntBuff
* Inputs: JNIenv * env, and jobject obj are handled by Java and required for JNI Functions
* Inputs: aStart (offset of the first input array from the start of the buffer)
* Inputs: bStart (offset of the second input array from the start of the buffer)
* Inputs: cStart (offset of the output 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 3. Vector library: C++ `addXIntBuff` function.

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 in Figure 4.

Using NIO, the vector library spent 98% of the time for a 1024 x 1024 matrix multiply in the computational phase and only 2% in overhead through the JNI call and data transfer. This enables the SIMD performance gains to be transferred to the Java code without being offset by overhead costs.

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.

4. Auto-Vectorization

Auto-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 [1].

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, reducing the number of kernels that the processor needs to run. The auto-vectorization tool takes advantage of this fact to implement the vector library described in Section 3.1.

4.1 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 Section 3.1 as an output. This allows for the end user to achieve the performance benefits of using the vectorization library without spending any effort on manual code modifications.

In order to parse and read the Java source code, this work takes advantage of the Java Parser toolkit [5]. 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 uses 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-

```
...
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 4. Code snippet illustrating the use of the `ByteBuffer`.

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.

4.2 Vectorization Dimensions

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 in Figure 5, is vectorizable because the vectorization library can load sequential elements into registers with only one instruction. However, the sample code in Figure 6 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 5. An example vectorizable operation.

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

Figure 6. An example unvectorizable operation.

For a single dimensional problem, the task of determining the vector size is not as straightforward. The tool allows for the programmer to enter a vector size manually if there is a specific desired vectorization length, but it can also determine a good vector size on its own. From analysis of several benchmarks, described further in section 5, 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 cores by running 8 vector kernels each with a vector length of 2048.

4.3 Restructure Loops

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.

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 in Figure 7. This ensures the resultant code remains still yields the same end results.

For some benchmarks this alone is enough to generate some speedup over 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.

4.4 Vectorization

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 in Figure 8. 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 ac-

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

Figure 7. Aparapi kernel rewritten to perform `vecSize` operations.

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

 for(int vecLoopIterator=0; vecLoopIterator<vecSize; vecLoopIterator++) {
   /* vectorizable operation */
 }

 for(int vecLoopIterator=0; vecLoopIterator<vecSize; vecLoopIterator++) {
   /* remaining kernel code */
 }
}
```

Figure 8. Aparapi kernel from Figure 7 rewritten to split each operation into a separate kernel.

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

 /* vectorizable operation */
 sub.vectorizableOperation(buffer, variable inputs, vecSize);

 for(int vecLoopIterator=0; vecLoopIterator<vecSize; vecLoopIterator++) {
   /* remaining kernel code */
 }
}
```

Figure 9. Aparapi kernel from Figure 8 rewritten to perform vectorizable operations using the vector library.

cessed, 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 2. 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 8, the auto-vectorization tool iterates over all the created loops and replaces any references to a buffer variable with code that reads associated variable from the `ByteBuffer`.

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 o checked to see if it is vectorizable. If it is then the entire loop is replaced with a call to the correct JNI vector method. Otherwise the loop is left alone, Figure 9 continues the example from Figure 8, showing the vectorizable loop replaced with a call to the JNI vector library.

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

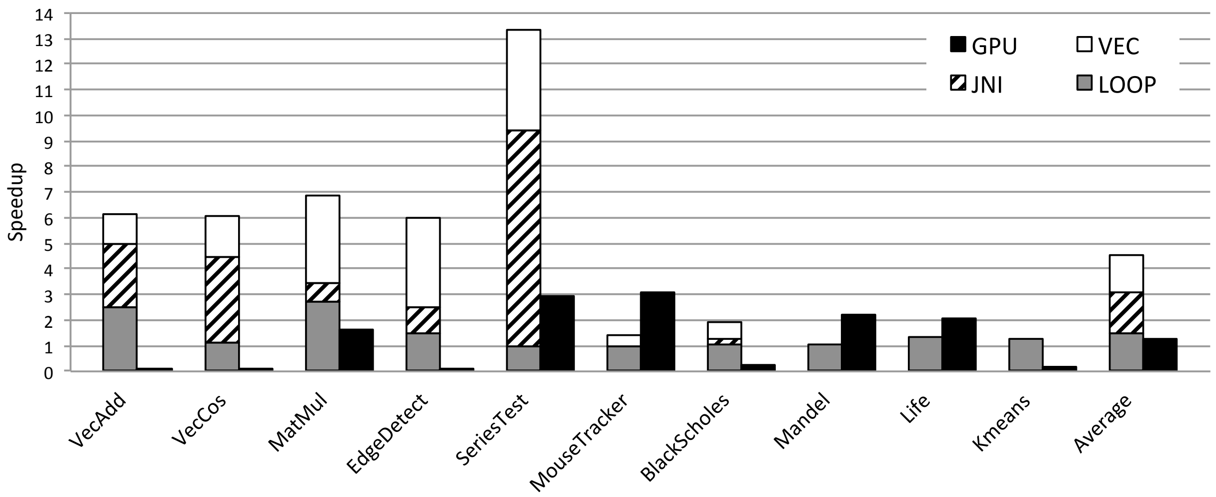


Figure 10. Desktop system results. Speedup compared to initial Java Thread Pool implementation.

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 indices are sequentially accessed by sequential kernels. If a corresponding vector instruction exist 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 to be executed as regular Java code.

4.5 Vectorizing Math Library Calls

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 do not exactly match the implementation in the Java Math library [15], these vectorizations can be disabled 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.

4.6 Buffer Creation

Once all of the 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.

5. Evaluation

In this section 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 one case a library written in Java to try to enable the Oracle Hotspot JIT compiler to vectorize instructions.

Java thread pool (JTP)
Stock Aparapi implementation using a Java thread pool, all speedup performance numbers are normalized to JTP.

Looped kernels on the Java thread pool (LOOP)
Version of the benchmark produced by the auto-vectorization tool but with no vectorization performed. All vectorizable operations, however, are bundled into their own Java loop.

C++ Library (JNI)
Version of the benchmark with vectorizable work-items replaced with calls to an *unvectorized* C++ library via JNI.

Vectorization library (VEC)
Fully auto-vectorized benchmark produced by the presented technique, with calls to the pre-vectorized C++ library via JNI.

Graphics Card (GPU)
Stock Aparapi benchmark ran on the GPU via OpenCL.

5.1 Experimental Setup

All of the experiments were run on a desktop system and a server system. The desktop system contains an 8-core AMD FX 8350 and a GeForce GT 610 with 48 CUDA cores. The server system consists of 4 Opteron 6376 16-core processors for a total of 64 CPU cores. The server also contained an NVIDIA GeForce GTX Titan GPU with a total of 2688 parallel CUDA cores. All experiments used the maximum available parallelism per architecture.

5.2 System Results

Figures 10 and 11 show the full set of results for the desktop and server systems respectively. The LOOP, JNI and VEC results are

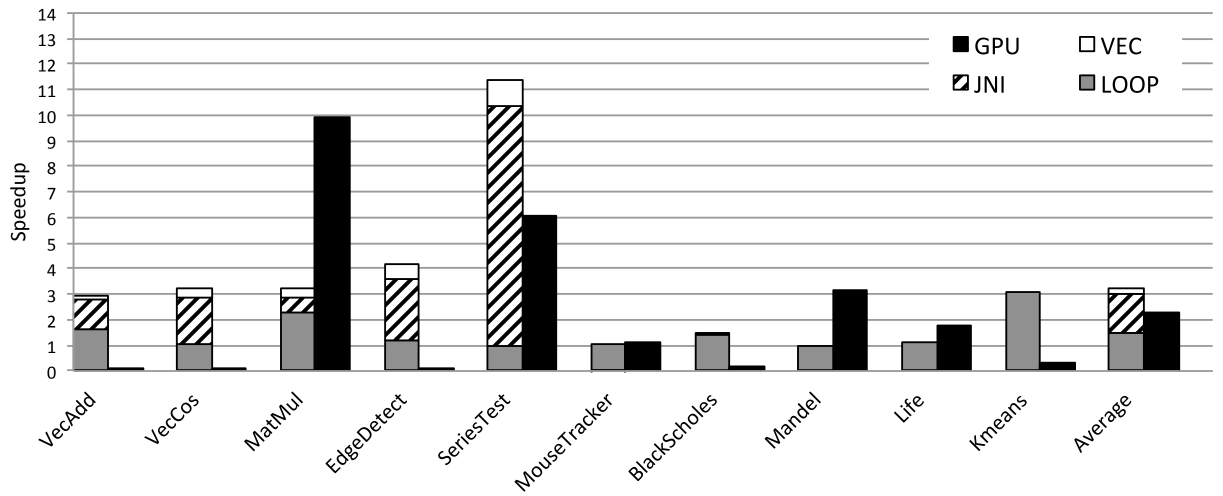


Figure 11. Server system results. Speedup compared to initial Java Thread Pool implementation.

presented as a stacked bar as each stage builds on the performance of the last. In Figure 10 it can be see that LOOP alone gives a reasonable performance improvement of 1.46x on average over Aparapi's Java thread pool (JTP). This is largely due to having to create far fewer kernel objects within Aparapi as each object executes many kernel instances. This does introduce the possibility of extending execution time for programs with highly uneven kernel execution time, as one thread may take longer to complete than the others. However, this program structure would also perform poorly with OpenCL and does not result in slow-downs for any of our benchmarks. The use of the non-vectorized C++ library (JNI) extends this speedup to 3.08x on average for the desktop system, due to reducing JIT overhead. Finally, the use of the vectorized library (VEC) brings the final average speedup to 4.56x for the desktop system. The step from JNI to VEC is entirely due to the use of native SIMD instructions.

Figure 11 shows the same trends on the server system for LOOP and JNI, but not for VEC. The step from JNI to VEC only takes the average speedup from 3.01x to 3.24x. This is because on the 64-core system the JIT-savings that the JNI implementation provides are comparatively larger than on the 8-core desktop as the higher core count means that the system can complete a larger portion of the benchmark before the baseline JTP implementation can complete its JIT compilation. This, combined with less work available per-core, means that the vectorized implementation can only provide a modest improvement for these benchmarks.

5.3 Benchmarks

Since Aparapi is still a work in progress, there are currently few benchmarks written using it. Aparapi provides a few sample programs: Matrix Multiply, Blackscholes, Mouse Tracker, Mandel, and Life. To supplement this extra programs were added, these are described and evaluated below.

5.3.1 VecAdd

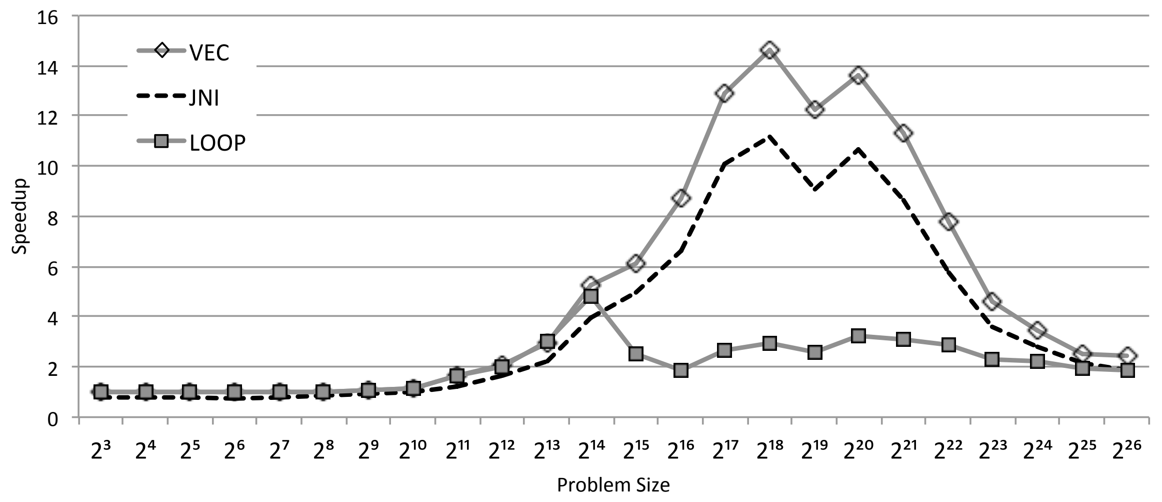
VecAdd is a simple vector addition calculation. It takes data stored in two single dimensional arrays, adds the corresponding elements, and stores the result in a third single dimensional array. This was a very simple micro-benchmark meant to isolate the vector operations to get a direct comparison of the vector operations against their non-vectorized Aparapi counterparts. This benchmark performs very poorly on the GPU as the trivial $O(n)$ computation can be completed on the CPU in less time than it takes to transfer the data to and from the GPU.

The simple nature of VecAdd makes it a natural choice to explore the vectorizer more deeply, this is done in Figures 12 and 13. Figure 12 shows the performance of the CPU-based implementations on different data sizes. As can be seen in Figure 12(a), 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 12(b) seeks to explain this peak behavior. It illustrates that while all three CPU kernel implementations see improved performance on larger data-sizes, the JTP implementation is slower to realize those benefits due to JIT-compiler warm-up time. At a problem size of 2^{14} elements performance of LOOP levels out until the JIT'ed version is ready. Thus the vectorizer sees the largest speedups for medium sized data-sets, but provides speedups on data-sizes as small as 2^9 elements and as large as 2^{26}.

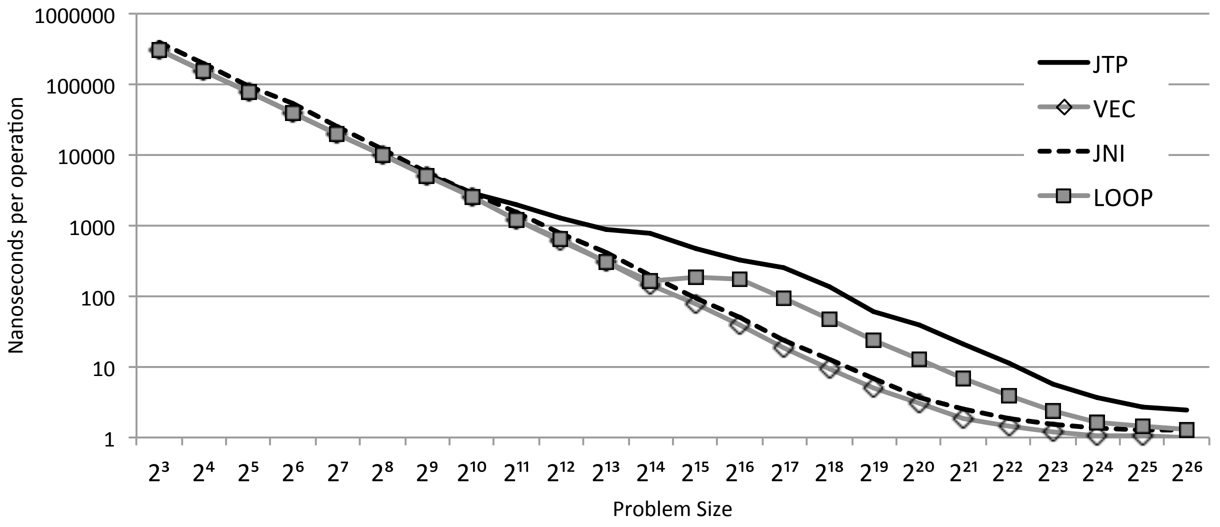
Figure 13 investigates the effect of vectorization sizes and work-item distribution sizes. Figure 13(a) varies the vectorization size parameter, i.e. how many calculations should be performed by a single call to the vectorization library. Larger sizes provide an obvious gain through reducing JNI overhead, though beyond a certain point no further gains can be made. A simple policy of choosing the largest possible vectorization size will provide optimal performance. Choosing a loop-size parameter, however, is not as simple. This parameter controls how many Aparapi work-items are bundled together into a single loop. Although small bundles do not provide optimal performance due to thread-object creation overhead, large bundles also do not provide optimal performance as they tend to have poor temporal locality in the cache due to performing long linear sequences of operations. It also increases the probability that at the end of a kernel execution some threads will be idling waiting on the last few threads to complete their large bundles. The peak seen in figure 13(b) relates to L1 data cache size – the best performance is found by maximally utilizing the data cache, but not exceeding its capacity.

5.3.2 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. This benchmark

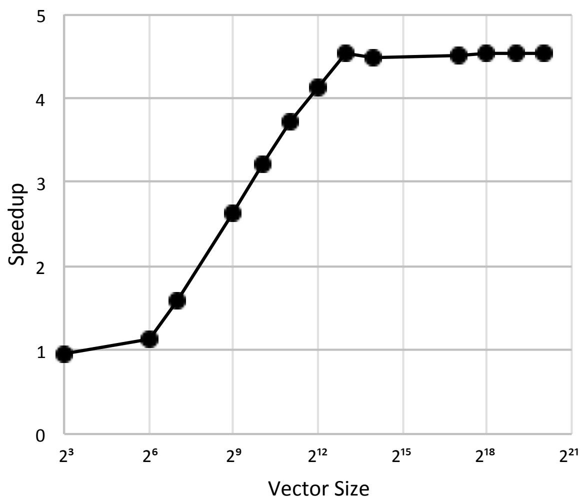


(a) Speedups over JTP vs problem size

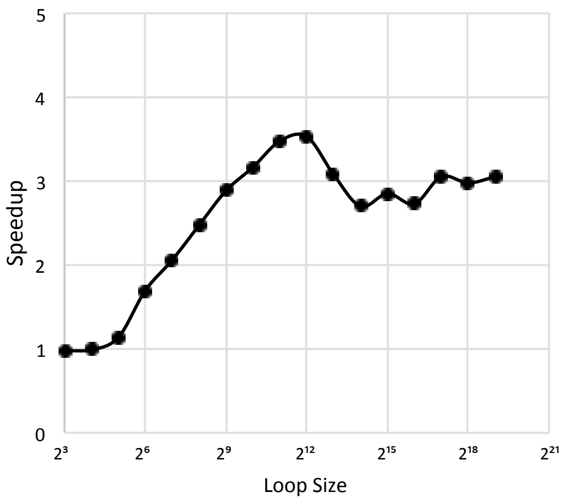


(b) Nanoseconds per operation vs problem size

Figure 12. Performance of CPU-based implementations of Vector Add (VecAdd) over various problem sizes on desktop system.

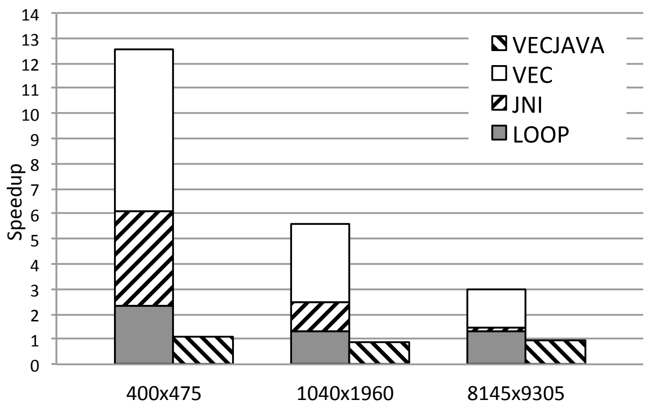


(a) Impact of varying vector size on performance

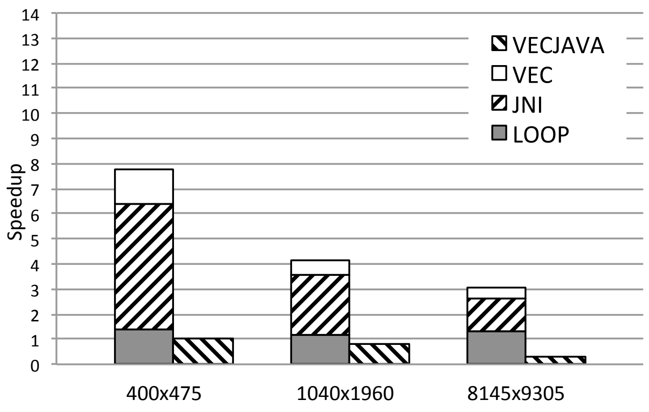


(b) Impact of varying loop size on performance

Figure 13. Effects of varying vectorization parameters for Vector Add (VecAdd) on various problem sizes for the desktop system.



(a) Edge detect speedups vs JTP on desktop system



(b) Edge detect speedups vs JTP on server system

Figure 14. Comparison of various CPU-based implementations of edge-detect for several image sizes.

has the same compute to data-transfer ratio problem that caused vector-add to perform poorly on the GPU, a result that is repeated here. VecCos was benchmarked because it compares the speed of hardware cosine operations against Java's Math library which does some of the calculations in software [15]. As with every other benchmark the results of the vectorized code were compared with the non-vectorized to ensure their validity. Although VEC provides approximately the same speedups for the VecCos benchmark as was achieved on VecAdd, the distribution of which stages provide those gains is different. LOOP provides less of an advantage as it is still using the Java Math library, JNI however provides a larger speed-up by using the optimized implementation and finally VEC improves on that result using SIMD instructions.

5.4 Matrix Multiply

Matrix multiply represents a benchmark that is ideal for a GPU as it has a low data-transfer to compute ratio and is "embarrassingly parallel". The server GPU was able provide an 80x speedup over JTP for a 4096 x 4096 matrix multiplication, illustrating the true potential of GPGPU computing when the conditions are ideal. However, for the 512x512 matrix multiply used in Figures 10 and 11 the GPU in the desktop struggled to overcome the initial startup overhead, with VEC outperforming the GPU. On the server system, however, the massively parallel GPU significantly outperforms the vectorized CPU implementation.

5.4.1 Edge Detect

EdgeDetect was ported to Aparapi from an existing Java implementation [18]. 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 JNI and vectorized kernels are also able to achieve an added cache performance benefit. Since they operate on several array elements at once they can capitalize on the array being loaded into the cache and operate on sequential elements already loaded before they get flushed. 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.

Figure 14 investigates the performance of edge-detect on several different image sizes, but it also provides results for a pure-Java version of the vectorization library. The hope was that by restructuring the code into obvious vector components the Hotspot JIT-compiler's limited vectorization capabilities [14] would be effective. Unfortunately this is not the case, inspecting the assembly produced by the Hotspot JIT-compiler showed that it did not vectorize any of the restructured loops. Even worse, this restructuring reduces performance due to increased call overhead and limiting the other optimizations that the JIT-performs. Using the pre-vectorized C++/JNI library, however, is able to provide enough performance to overcome these overheads.

5.4.2 Series Test

Series Test comes from the Java Grande Benchmarking Suite's multi-threaded benchmarks [4] 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, the Series Test benchmark benefits from the hardware implementations of the trigonometric functions. This can be seen in the JNI results that provide considerable speed-ups even without vectorization. With the addition of SIMD instructions VEC achieves 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.

5.5 Mouse Tracker

Mouse Tracker was another simple image manipulation benchmark that came with the Aparapi source code. It was modified to not rely on user-input, but to follow a pre-programmed sequence.

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

5.6 Black Scholes

The Black Scholes benchmark also came with Aparapi and is a mathematical model of a financial market. While Black Scholes is highly parallel, the GPU provides a slowdown. This is due to the GPU's initial startup overhead. On the desktop system the JTP implementation is able to complete each iteration in around 70ms and the initial overhead for the first iteration of the GPU implementation is around 780ms 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.

5.7 Mandelbrot, Life, and Kmeans

Mandelbrot, Life, and Kmeans were not able to be vectorized across kernels because either the majority of their code was within kernel dependent control flow, or they accessed non-sequential elements of arrays and therefore were unvectorizable across kernels. Work-item bundling, i.e. "LOOP", was still able to provide small speedups however.

6. Conclusions

In this work, we presented an auto-vectorizer that allows Aparapi Java programs to utilize SIMD instructions even though the Java JIT compiler does not. This can be done with no extra effort from the programmer. The speedups achieved by the tool are 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 system and 1.5x to 2x on the server system. However, not all kernels are completely vectorizable. Some kernels have control flow logic that can reduce the number of vectorizable operations. The vectorizer was still able to improve performance in these cases by bundling together multiple work items.

In investigating the parameters of auto-vectorization we found that although the performance of the Java thread pool increases on longer-running workloads, due to amortizing JIT compiler overhead, it never matches the performance of the JNI implementation. We found that when distributing Aparapi work-items to threads the size should be limited based on CPU cache size, but that each bundle of work-items should be processed with the maximum vectorsize to reduce JNI overhead.

For initial future work the obvious path is to try to create a JIT-compiler auto-vectorizer that is able to vectorize the pure-Java vector library used in Figure 14. A more interesting path, however, may be to try to exploit the same Aparapi-aware knowledge used in this paper directly within the JIT-compiler. API specific optimizations are generally only provided for heavily used APIs, but the Graal project [13] provides a path to insert this type of optimization.

7. Acknowledgments

This work is supported by the US Office of Naval Research under Contract N00014-12-1-0880.

References

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

[2] Aparapi. "What is Aparapi?". 2014. `https://code.google.com/p/aparapi/`.

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

[4] EPCC. "The Java Grande Forum Multi-threaded Benchmarks". `http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/threads.html`

[5] J. V. Gesser. "Java 1.7 parser and Abstract Syntax Tree". 2014. `https://github.com/matozoid/javaparser`

[6] A. Hayashi, M. Grossman, J. Zhao, J. Shirako and V. Sarkar. "Accelerating Habanero-Java programs with OpenCL generation" in Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages, and Tools (PPPJ '13), pp 124-134, 2013.

[7] Eric Holk, William Byrd, Nilesh Mahajan, Jeremiah Willcock, Arun Chauhan, and Andrew Lumsdaine. "Declarative Parallel Programming for GPUs" in Proceedings of the International Conference on Parallel Computing (ParCo '11), 2011.

[8] Khronos Group "The OpenCL Specification. Version 1.0". 2009.

[9] J. Nie, B. Cheng, S. Li, L. Wang and X.-F. Li, "Vectorization for Java" in Proceedings of the 2010 IFIP international conference on Network and Parallel Computing (NPC '10), pp 3-17, 2010.

[10] D. Nuzman, S. Dyshel, E. Rohou, I. Rosen, K. Williams, D. Yuste, A. Cohen and A. Zaks, "Vapor SIMD: Auto-Vectorize Once, Run Everywhere" in Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11), pp 151-160, 2011.

[11] NVIDIA, "What is CUDA?". 2014. `http://www.nvidia.com/object/cuda\_home\_new.html`.

[12] NVIDIA, "Cuda C Programming Guide". 2014. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[13] OpenJDK. "Graal Project". 2014. `http://openjdk.java.net/projects/graal/`

[14] Oracle, "JDK-6340864 : Implement vectorization optimizations in hotspot-server". 2013 `http://bugs.java.com/view_bug.do?bug_id=6340864`.

[15] Oracle Corporation. "Class Math". 2014. `http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html`.

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

[17] P. C. Pratt-Szeliga, J. W. Fawcett and R. D. Welch. "Rootbeer: Seamlessly Using GPUs from Java" in Proceedings of High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS '12), pp 375-380, 2012.

[18] R. Sedgewick and K. Wayne, "EdgeDetector". 2009. `http://introcs.cs.princeton.edu/java/31datatype/EdgeDetector.java.html`

[19] A. Stromme, R. Carlson and T. Newhall. "Chestnut: A GPU Programming Language for Non-Experts," in Proceedings of the International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM '12), pp 156-167, 2012.