Safe Acceleration of Habanero-Java Programs with OpenCL Generation

Akihiro Hayashi
Department of Computer Science
Rice University
Houston, TX, USA
ahayashi@rice.edu

Max Grossman
Department of Computer Science
Rice University
Houston, TX, USA
jmg3@rice.edu

Jisheng Zhao
Department of Computer Science
Rice University
Houston, TX, USA
jisheng.zhao@rice.edu

Jun Shirako
Department of Computer Science
Rice University
Houston, TX, USA
shirako@rice.edu

Vivek Sarkar
Department of Computer Science
Rice University
Houston, TX, USA
vsarkar@rice.edu

Abstract

The initial wave of programming models for general-purpose computing on GPUs, led by CUDA and OpenCL, has provided experts with low-level constructs to obtain significant performance and energy improvements on GPUs. However, these programming models are characterized by a challenging learning curve for non-experts due to their complex and low-level APIs. Looking to the future, improving the accessibility of GPUs and accelerators for mainstream software developers is crucial to bringing the benefits of these heterogeneous architectures to a broader set of application domains. A key challenge in doing so is that mainstream developers are accustomed to working with high-level managed languages, such as Java, rather than lower-level native languages such as C, CUDA, and OpenCL.

The OpenCL standard enables portable execution of SIMD kernels across a wide range of platforms, including multi-core CPUs, many-core GPUs, and FPGAs. However, using OpenCL from Java to program multi-architecture systems is difficult and error-prone. Programmers are required to explicitly perform a number of low-level operations, such as (1) managing data transfers between the host system and the GPU, (2) writing kernels in the OpenCL kernel language, (3) compiling kernels & performing other OpenCL initialization, and (4) using the Java Native Interface (JNI) to access the C/C++ APIs for OpenCL.

In this paper, we present compile-time and run-time techniques for accelerating programs written in Java using automatic generation of OpenCL as a foundation. Our approach includes (1) automatic generation of OpenCL kernels and JNI glue code from a parallel-for construct (`forall`) available in the Habanero-Java (HJ) language, (2) leveraging HJ's `array view` language construct to efficiently support rectangular, multi-dimensional arrays on OpenCL devices, and (3) implementing HJ's phaser (`next`) construct for all-to-all barrier synchronization in automatically generated OpenCL kernels. Our approach is named HJ-OpenCL. Contrasting with past approaches to generating CUDA or OpenCL from high-level languages, the HJ-OpenCL approach helps the programmer preserve Java exception semantics by generating both *exception-safe* and *unsafe* code regions. The execution of one or the other is selected at runtime based on the `safe` language construct introduced in this paper.

We use a set of ten Java benchmarks to evaluate our approach, and observe performance improvements due to both native execution of OpenCL code and parallelism. On an AMD APU, our results show speedups of up to 36.7× relative to sequential Java when executing on the host 4-core CPU, and of up to 55.0× on the integrated GPU. For a system with an Intel Xeon CPU and a discrete NVIDIA Fermi GPU, the speedups relative to sequential Java are 35.7× for the 12-core CPU and 324.0× for the GPU. Further, we find that different applications perform optimally in JVM execution, in OpenCL CPU execution, and in OpenCL GPU execution. The language features, compiler extensions, and runtime extensions included in this work enable portability, rapid prototyping, and transparent execution of JVM applications across all OpenCL platforms.

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

General Terms Languages, Design.

Keywords GPGPU, OpenCL, Java, Habanero-Java.

PPPJ'13,, September 11–13, 2013, Stuttgart, Germany..

http://dx.doi.org/10.1145/2500828.2500840

1. Introduction

While the performance and energy benefits of heterogeneous computing are now well established, the adoption of heterogeneous computing has still been slow in many domains that can benefit significantly from the use of accelerators. The primary reason for this is the high complexity of developing and maintaining software for heterogeneous systems.

The two leading programming models for heterogeneous computing, OpenCL [15] and CUDA [18], give software developers

fine-grain control over the hardware platform, transfer of data, and execution of kernels. The verbosity of the OpenCL and CUDA APIs allows HPC specialists to use advanced, hardware-specific tuning techniques. However, this fine-grained control is a double-edged sword and adds a non-trivial amount of application code to write, maintain, and optimize across multiple architectures. While OpenCL offers functional portability, it is often the case that different OpenCL variants of the same kernel have to be written to obtain the best performance on different hardware devices.

As a result, much attention has been given to using OpenCL and CUDA as foundations for higher level programming models. For instance, consider OpenACC [9]. OpenACC uses OpenMP-like compiler directives to identify parallel regions of code that should be offloaded to an accelerator. It supports a number of directives related to data motion and kernel execution, but is only accessible from C and Fortran, two low-level languages. Additionally, while OpenACC minimizes the code changes required to add heterogeneity to a program, it also reduces the control a programmer has in optimizing that heterogeneous execution, reduces the flexibility of the system for arbitrary kernels, and hides a number of important features from the application developer which often impact performance significantly.

On the other hand, using high-level managed languages, such as Java, often has many benefits. These benefits include a high-level syntax, massive library support, and garbage collection. However, the performance of an application can often suffer when interpreted, garbage-collected languages are used. Using heterogeneous systems to accelerate applications in these high-level languages is also a difficult and error-prone task. Accessing OpenCL's C/C++ API from Java requires the use of the Java Native Interface (JNI) API, immediately removing many of the programmability benefits of Java software development.

In this work, we use language features of the Habanero-Java (HJ) [4] parallel programming language to support implicit heterogeneous execution from the JVM using OpenCL. Habanero-Java with OpenCL generation (HJ-OpenCL) is an extension to the parallel Habanero Java programming language which enables execution of parallel `forall` loops on any heterogeneous processor in an OpenCL platform without any code change to the original HJ source. In addition, the programmer may use the `next` language construct for all-to-all barrier synchronization, and the `safe` language construct to preserve Java exception semantics. These extensions do not interfere with existing HJ constructs for other types of parallelism, such as asynchronous tasks, futures and actors, but are instead intended to complement them. As a result, an application can be written entirely in the high-level HJ programming language and take advantage of existing Java libraries, frameworks, and tools while accelerating computationally-intensive parallel loops using the multi-threaded HJ runtime or many-threaded OpenCL kernels. The main contributions of this work include:

1. Auto-utilization of multi-core CPUs and many-core GPUs using OpenCL from the Habanero-Java language, while maintaining Java exception semantics with the `safe` construct introduced in this paper.
2. Auto-generation of bridge code between the JVM and OpenCL. This includes OpenCL device discovery, OpenCL platform initialization, data movement between the JVM and OpenCL devices, OpenCL kernel execution, and barrier synchronization within kernels.
3. Auto-generation of OpenCL kernels from Java bytecode with extensions to the APARAPI [1] open-source translation framework.

```
1  public class ArraySum {
2    public static void main(String[] args) {
3      int[][] arrays = new int[N][M];
4      int[] result = new int[N];
5      ... arrays initialization ...
6      for(int i=0; i < N; i++) {
7        result[i] = 0;
8        for(int j=0; j < M; j++) {
9          result[i] += arrays[i][j];
10       }
11     }
12   }
13 }
```

Figure 1: Original Array-Sum Java Code. Note that while this code is short and simple, it is also O(N*M) where N is the number of arrays to sum and M is the length of each array.

4. Performance evaluation of HJ-OpenCL on multiple heterogeneous platforms with performance comparison between sequential Java, sequential HJ, multi-threaded HJ, and HJ-OpenCL on multi-core CPUs and many-core GPUs.

2. Background

A well-known approach from past work on accelerating Java programs with heterogeneous accelerators is exemplified in the Rootbeer system [23]. Rootbeer is built on top of CUDA. It uses multiple threads and reflection to rapidly serialize Java objects to a native format compatible with CUDA's C/C++ API, a translation framework to auto-generate CUDA kernels from Java bytecode at compile-time, and a runtime to manage execution of CUDA kernels and the transfer of input and output data. Rootbeer supports "all features of the Java Programming Language except dynamic method invocation, reflection, and native methods" [23]. In this section we will study Java, Rootbeer, and HJ-OpenCL implementations of an application similar to the Array-Sum example in [23].

The Array-Sum example studied in this section takes as input a matrix of size $N \times M$ (`arrays`), and returns a vector of size N (`result`) for which the i^{th} element is the sum of the elements contained in row i of the input matrix. This is a data-parallel application which we would expect to perform well on GPUs for large input sizes.

First, let us consider the sequential Java implementation in Figure 1. Nested for-loops are used to sum each input array into its corresponding output element. While the implementation is simple, it is single-threaded and takes $O(N \times M)$ time.

Now consider the Rootbeer implementation. In Rootbeer, any computation to be executed on a CUDA device must be encapsulated in a class which implements Rootbeer's *Kernel* interface. *Kernel* contains a *gpuMethod* method which is used as the entry point for the computation kernel and is executed by each CUDA thread. The work to be performed by CUDA in a Rootbeer program is specified by passing a vector of *Kernel* objects to a *Rootbeer* object's *runAll* method. Each of these *Kernel* objects represents a thread of execution to be run on the CUDA device.

The Rootbeer implementation of Array-Sum is shown in Figure 2. It is clear that using Rootbeer dramatically increases code length (approximately 2x) and decreases readability by separating the computational kernels from application code. The parallelism in this program is not readily apparent, hidden by a sequential loop adding *Kernel* objects to the *jobs* list. While Rootbeer provides impressive support for executing Java programs on CUDA devices, the programming model significantly decreases programmability relative to the original sequential Java implementation. It is also important to note that using Rootbeer eliminates Java exception se-

```
1  public class ArraySum {
2    public static void main(String[] args) {
3      int[][] arrays = new int[N][M];
4      int[] result = new int[N];
5      ... arrays initialization ...
6      List<Kernel> jobs =
7          new ArrayList<Kernel>();
8      for(int i = 0; i < N; i++) {
9        jobs.add(new ArraySumKernel(arrays[i],
10           result, i);
11     }
12     Rootbeer rootbeer = new Rootbeer();
13     rootbeer.runAll(jobs);
14   }
15 }
16
17 class ArraySumKernel implements Kernel {
18   private int[] source;
19   private int[] ret;
20   private int index;
21   public ArraySumKernel(int[] source,
22       int[] ret, int i) {
23     this.source = source;
24   this.ret = ret;  this.index = i;
25   }
26   public void gpuMethod() {
27     int sum = 0;
28     for(int i = 0; i < source.length; i++) {
29       sum += source[i];
30     }
31     ret[index] = sum;
32   }
33 }
```

Figure 2: Rootbeer Implementation.

mantics, and common errors such as out-of-bounds or null pointers may not be reported.

Finally, we compare the Java and Rootbeer implementations to the HJ-OpenCL implementation in Figure 3. The nested for-loop structure in the Java sequential version is retained in the HJ-OpenCL benchmark, without the added encapsulation and separation required for Rootbeer. In the HJ-OpenCL implementation, the outer for loop is replaced by a parallel HJ `forall` loop which executes every iteration in parallel. There are other two modifications from the Java sequential version. First, an HJ array view object *arraysView* is declared at line 6. This array view object provides two-dimensional indexing for the one-dimensional *arrays* array. Translating *arrays* to a one-dimensional array supports efficient serialization to OpenCL devices. Now, the *arraysView* is referenced in the parallel `forall` loop in place of the *arrays* object backing it. The second change, the addition of a `safe` block in line 8, is used to preserve Java exception semantics. A `safe` statement takes a boolean condition, and indicates to the compiler that no exceptions are thrown inside the body of that block if that condition is true. The compiler then generates exception-safe and unsafe versions of the same block. These HJ language constructs will be covered in more detail later in Section 3.

It is important to note that the implementation in Figure 3 can be executed without modification on the multi-threaded HJ runtime, in native threads on a multi-core CPU, or in native threads on a many-core GPU.

3. Habanero Java Language

This section describes features of the Habanero-Java (HJ) parallel programming language which are added or supported by HJ-OpenCL.

```
1  public class ArraySum {
2    public static void main(String[] args) {
3      int[] arrays = new int[N*M];
4      int[] result = new int[N];
5      ... arrays initialization ...
6      int[.] arraysView = new arrayView(arrays, 0, [0:N
         -1, 0:M-1]);
7      boolean isSafe = ...;
8      safe(isSafe) {
9        forall(point [i] : [0:N-1]) {
10         result[i] = 0;
11         for(int j=0; j<M; j++) {
12           result[i] += arraysView[i,j];
13         }
14       }
15     }
16   }
17 }
```

Figure 3: HJ-OpenCL Implementation.

3.1 Overview of HJ language

The Habanero Java (HJ) parallel programming language [4] provides an execution model for multicore processors that builds on four orthogonal constructs:

1. Lightweight *dynamic task creation and termination* using *async* and *finish constructs* [10].
2. *Locality control* with task and data distributions using the *place* construct [5].
3. Mutual exclusion and isolation among tasks using the *isolated* construct [16].
4. Collective and point-to-point synchronization using the *phasers* construct [26].

Since HJ is based on Java, the use of certain primitives from the Java Concurrency Utilities [21] is also possible in HJ programs.

HJ-OpenCL only supports OpenCL execution of a subset of the parallel language features of Habanero-Java. Work to date on HJ-OpenCL focuses on supporting `forall` parallel loops and `next` barrier statements for OpenCL execution, without affecting the use of other language features targeting the HJ runtime:

1. **forall:** The statement "`forall(point p :` *region*`)` ⟨*stmt*⟩" indicates a parallel loop whose iteration space is defined by a *region*. The region can be one- or multi-dimensional space, e.g., `[0:M-1,0:N-1]` for a 2-D iteration space. Each iteration instance executes the loop body ⟨*stmt*⟩ given a different *point* in the iteration space. All `forall` loops end with an implicit barrier (finish scope).
2. **next:** The statement `next` represents an all-to-all barrier synchronization [1] point with both signal and wait semantics in a parallel `forall` loop.

3.2 The `safe` Construct

This work adds the `safe` language construct to Habanero-Java. In HJ-OpenCL, programmers use the `safe` language feature to specify conditions which ensure no exception is thrown in a code region. The "`safe` (boolean cond) ⟨*stmt*⟩" construct indicates that ⟨*stmt*⟩ is not expected to throw an exception if `cond` is true.

To illustrate this, let us again consider the ArraySum example from Section 2. To check the non-nullness of *result*, *arrays* and

[1] HJ-OpenCL only supports the all-to-all barrier synchronization in phasers due to OpenCL constraints. (See Section 4.2)

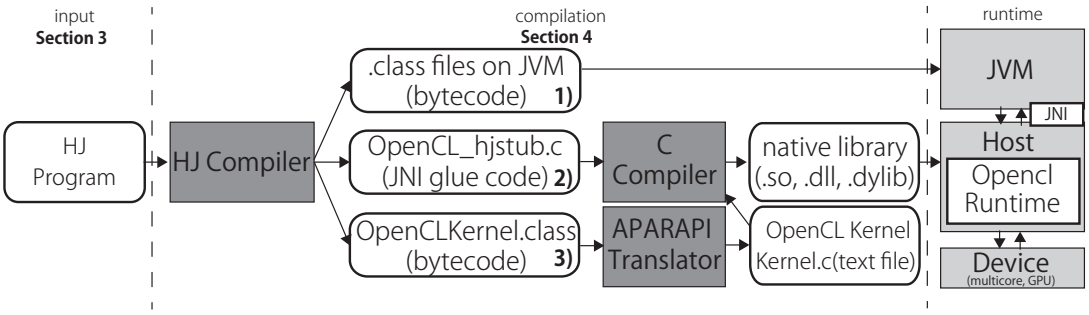

Figure 4: Compilation Flow and Runtime

arrayViews, a programmer would put *isSafe = (result != null && arrays != null && arraysView != null)*. It would also be necessary to add *N - 1 ≤ result.length && N*M-1 ≤ arrays.length* for array bounds checking. The HJ-OpenCL compiler and runtime can then use these conditions to safely generate and execute code on OpenCL devices. Section 4.1 describes the detail of code generation for the `safe` constructs.

3.3 Supporting Multi-Dimensional Arrays

The OpenCL specification[14] does not support regular or irregular multi-dimensional arrays in OpenCL kernels. However, computationally heavy Java programs often make use of regular multi-dimensional arrays as a way of organizing data, such as multi-dimensional matrices. Therefore, it is important for HJ-OpenCL to provide software developers with language support for regular multi-dimensional arrays while enabling efficient translation and transfer of those arrays for use in OpenCL kernels.

In the HJ-OpenCL language, HJ's array views are used to provide the software developer with what syntactically appears to be a multi-dimensional array but which is actually backed by a one-dimensional buffer. The statement "*dataType*`[.] v = new arrayView(`*baseArr*`,`*offset*`,`*region*`)` declares a view `v` on a one-dimensional array *baseArr* with starting offset *offset*. The created view `v` can then be referenced with syntax similar to multi-dimensional Java arrays, with bounds defined by *region*. The details of supporting array views at compile- and run-time in HJ-OpenCL are discussed in Section 4.3.

4. OpenCL Generation

While HJ-OpenCL provides language features to allow the programmer to identify parallel loop kernels in their Java code, mapping these kernels to OpenCL devices requires compile-time and run-time support. This section introduces the code generation techniques used at compile-time which take JVM bytecode as input and produce OpenCL kernels and the required glue code (i.e. the JNI stub) that invoke kernels and perform data transfer.

The HJ-OpenCL compiler leverages APARAPI[1], a comprehensive, open-source framework for executing computational kernels from Java applications on OpenCL devices. For this work we extended the APARAPI component that generates OpenCL code from Java bytecode. Our major extension to APARAPI has been to enable static translation from Java bytecode to OpenCL, whereas the publicly available APARAPI framework only works in a dynamic compilation mode.

In addition to OpenCL kernels, glue code must be automatically generated to transfer execution and data from the JVM to the OpenCL device and back. This functionality is provided internally

```
1  public class ArraySum {
2    static { System.loadLibrary("libCalc"); }
3    public static native void openCL_Kernel0(...);
4    public static void main(String[] args) {
5    ...
6    isSafe = ...;
7    if (isSafe) {
8      /* Safe Region */
9      openCL_Kernel0(arrays, result); // JNI call
10   } else {
11     /* Unsafe Region, run in JVM */
12     forall (point [i]:[0:N-1]) {
13       result[i] = 0;
14       for(int j=0; j<M; j++) {
15         result[i] += arraysView[i,j];
16       }
17     }
18   }
19 }
```

Figure 5: Generated code by the HJ compiler

by the HJ-OpenCL compiler, and includes the generation of JNI functions, OpenCL API calls, and transformed bytecode.

The rest of this Section is organized as follows: Section 4.1 introduces the steps of code generation. Section 4.2 discusses the solution to barrier synchronization support. Section 4.3 discusses the HJ array view construct which enables efficient support of multi-dimensional arrays in parallel loops translated to OpenCL.

4.1 Compilation Flow

Figure 4 illustrates the compilation flow and how the compiler interacts with the HJ-OpenCL runtime. The HJ-OpenCL compiler takes a HJ program (see Section 3) as inputs, and produces the following outputs:

1. Java CLASS files intended for execution in JVM.
2. JNI glue code between the JVM and OpenCL kernels, contained in the file *OpenCL_hjstub.c*.
3. A Java CLASS file named *OpenCLKernel.class* which contains all bytecode that needs to be translated to OpenCL kernels by the APARAPI bytecode translator.

To explain the compilation work flow, let us again consider the ArraySum example from Section 2. The ArraySum implementation from Figure 3 is translated to the Java code shown in Figure 5. During code generation, the HJ-OpenCL compiler identifies the input and output data relationships of the `forall` loop, and replaces the original `forall` loop with a native method

```
1  cl_context       context;
2  cl_command_queue command_queue;
3  JNIEXPORT void  JNICALL Java_ArraySum_initGPU(JNIEnv * env, jclass cls) {
4     cl_int           ret;
5     /* Get Device */
6     clGetPlatformIDs(...);
7     clGetDeviceIDs(...);
8     /* OpenCL create Context and Queue */
9     context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &ret);
10    command_queue = clCreateCommandQueue(context, device_id, 0, &ret);
11 }
12 extern char *HjKernel0; // from Kernel.c
13 JNIEXPORT void JNICALL Java_ArraySum_openCL_1Kernel0(JNIEnv * env, jclass cls, jintArray arrays, jintArray result,
       ... ) {
14    void *aptr = (env)->GetPrimitiveArrayCritical(arrays, 0);
15    void *rptr = (env)->GetPrimitiveArrayCritical(result, 0);
16    ...
17    /* Create Buffer */
18    cl_mem Aobj = clCreateBuffer(context, ...);
19    cl_mem Robj = clCreateBuffer(context, ...);
20    /* Host to Device Communication */
21    clEnqueueWriteBuffer(command_queue, Aobj, ..., aptr, ...); // transfer aptr
22    clEnqueueWriteBuffer(command_queue, Robj, ..., rptr, ...); // transfer rptr
23    /* Kernel Compilation */
24    clCreateProgramWithSource(context, 1, (const char **) &HjKernel0, ...);
25    clCreateKernel();
26    clSetKernelArg(...);
27    /* Invoke Kernel */
28    clEnqueueNDRangeKernel(command_queue);
29    /* Device to Host Communication */
30    clEnqueueReadBuffer(command_queue); // transfer rptr
31    /* Clean up */
32    clFinish(command_queue);
33    ...
34    (env)->ReleasePrimitiveArrayCritical(arrays, aptr, 0);
35    (env)->ReleasePrimitiveArrayCritical(result, rptr, 0);
36 }
```

Figure 6: Generated JNI and OpenCL code (*OpenCL_hjstub.c*)

named *openCL_Kernel0()* (line 9). This native method invokes the OpenCL API to execute computations on an OpenCL device. Its definition is contained in a library, *libCalc* (line 2). These transformations translate the original HJ program with `forall` constructs to JVM bytecode with JNI stubs for OpenCL kernel invocation.

A powerful feature of Java is runtime exceptions, thrown when some safety constraints are violated. For example, the array access *result[i] = 0;* in Figure 3 might throw "ArrayIndexOutOfBoundsException" or "NullPointerException". Running natively in OpenCL removes all safety guarantees from Java exceptions. HJ-OpenCL code generation maintains Java exception semantics by generating two versions of the same `forall` loop: one to run when the programmer guarantees no exceptions will be thrown, and one to run when the exception-safe conditions do not hold. The code region to generate these two versions is specified by the *s*afe language construct, which takes user-specified conditions as input (see Section 3.2). If these conditions hold at runtime, HJ-OpenCL assumes no exception will be thrown inside the *s*afe block. For example, in Figure 3 the programmer has set the value of *isSafe*. This definition is kept unchanged through to the transformed version in Figure 5. If the exception-safe condition is met, the runtime uses *openCL_Kernel0()* to launch OpenCL execution. Otherwise the computation is conservatively run in the JVM to preserve Java exception semantics. These changes are only visible in the emitted bytecode. In the case that programmers do not provide a `safe` region around a parallel loop but have provided compiler flags indicating that OpenCL execution is desired, the HJ-OpenCL compiler and runtime trusts that no exception is thrown.

OpenCL_hjstub.c in Figure 6 is the auto-generated bridge between the JVM and OpenCL devices. It uses JNI to invoke the OpenCL API to initialize the OpenCL device, transfer data, compile and launch the OpenCL kernel, and return data and control to the JVM. *OpenCL_hjstub.c* and *Kernel.c*, which contains the OpenCL kernels translated from HJ `forall` loops, are generated by the HJ-OpenCL compiler and built into a library which the JVM links to at runtime.

The first function in Figure 6, *Java_ArraySum_initGPU*, checks for OpenCL devices and creates an OpenCL context and command queue. The HJ-OpenCL compiler inserts an asynchronous invocation of this function at the very beginning of the input HJ program to reduce the overhead of OpenCL device initialization.

The second function, *Java_ArraySum_openCL_1Kernel0*, performs the following steps to run the ArraySum example in OpenCL:

1. The JNI call *GetPrimitiveArrayCritical* is used to retrieve a bare pointer to the contents of Java arrays (line 14, 15).
2. The OpenCL API is invoked to transfer scalar and vector inputs from the JVM into the OpenCL address space (line 17 - 22).
3. The ArraySum kernel is compiled and executed on an OpenCL device (line 23-28).
4. The OpenCL API is invoked to retrieve results from the device if they are marked as output (line 29-32).
5. The JNI call *ReleasePrimitiveArrayCritical* is used to release access to the Java arrays after copying the results into them and before returning to JVM execution (line 34, 35).

```
1  public class OpenCLKernel
2          extends com.amd.aparapi.Kernel {
3      int[] result, array;
4      int OpEnCl_size;
5      public OpenCLKernel();
6      public void run() {
7          int gid = this.get_global_id(0);
8          if (gid >= OpEnCl_size) { return; }
9          for (int j = 0; j < M; j++) {
10           result[gid] += array[gid*M + j];
11         }
12     }
13 }
```

Figure 7: Example code of OpenCLKernel.class

The paragraphs above have covered the generation of all bytecode and source code intended for execution in the host system. To generate the OpenCL kernel code, the APARAPI translator is passed a single CLASS file, *OpenCLKernel.class*. The HJ-OpenCL compiler generates *OpenCLKernel.class* by extracting the body of parallel `forall` loops and translating them into a format which the APARAPI translator can process. Figure 7 shows the equivalent Java of the bytecode generated in *OpenCLKernel.class*. Line 1 shows that *OpenCLKernel* extends *com.amd.aparapi.Kernel*, as required by the APARAPI translator. The body of the original `forall` loop is placed in the *run* method (lines 6-11). The APARAPI translator automatically identifies the *run* method of any class extending *com.amd.aparapi.Kernel* as the entry point of an OpenCL kernel, and can be used to emit an equivalent OpenCL kernel. The generated OpenCL kernel is expressed as array of characters(*HjKernel0*) which is stored in *Kernel.c* and later compiled with *OpenCL_hjstub.c*(see line 12, 24 in Figure 6).

4.2 Barrier Synchronization

HJ-OpenCL also supports all-to-all barrier synchronization across parallel iterations of a `forall` loop. The programmers can specify synchronization points in a loop using the *next* statement.

Consider the HJ-OpenCL barrier example in Fig 8. Each iteration of the `forall` loop calls *method1* and *method2*. Following execution of *method1(i, j)* or *method2(i, j)* each `forall` iteration is blocked until all other iterations also complete the corresponding calls in their own execution streams. Because OpenCL does not support all-to-all barrier synchronization as a kernel language feature, the HJ-OpenCL compiler partitions the `forall` loop body into blocks separated by synchronization points. Rather than performing a single kernel invocation for a single `forall` loop in *OpenCL_hjstub.c*, the HJ-OpenCL compiler must enqueue multiple kernels for execution, where each kernel implements a different block of the original `forall` loop body. This takes advantage of OpenCL's implicit all-to-all synchronization between kernel launches in the same command queue.

Fig 9 shows the Java equivalent of the generated bytecode in *OpenCLKernel.class* for the example in Figure 8, which will eventually be translated to an OpenCL kernel. *OpenCLKernel.class* declares *passId*, which identifies a block of the body of the original `forall` loop. Fig 10 shows the sequence of OpenCL API calls for this barrier example. Note that the value of *passId* passed to the OpenCL kernel is incremented before each kernel launch, leading to execution of a different block of code for each kernel invocation as the kernel progresses through synchronization points.

4.3 Multidimensional Array Issues

In Java, a multi-dimensional array is represented as an array of arrays. This permits irregularly shaped arrays where elements at the same dimension have different lengths. This implies that, given an array `int[10][10] A = new int[10][10]`, it would be necessary to perform 10 calls to OpenCL's transfer functions, *clEnqueueWriteBuffer* and *clEnqueueReadBuffer*, for transferring `A` to and from an OpenCL device. This leads to the high communication overheads between host and device.

As discussed in Section 3.3, we instead use HJ array-views[13] to provide multi-dimensional syntax backed by a single-dimensional array. In Figure 3, line 3 shows the declaration of a one-dimensional array. Line 6 in Figure 3 shows the declaration of an array-view `arraysView` which maps a two-dimensional region of size [0:N-1,0:M-1] to the one-dimensional array`arrays`. The array-view `arraysView` is referenced with multi-dimensional syntax in the body of *forall* as shown in line 12. The HJ compiler internally transforms `arraysView[i, j]` into `arrays[i*M + j]`. APARAPI is then able to translate this statement to an OpenCL kernel. APARAPI does not support multi-dimensional arrays.

```
1  forall (point [i]:[0:n-1]) {
2    method1(i, j);
3    // synchronization point 1
4    next;
5    method2(i, j);
6    // synchronozation point 2
7    next;
8  }
```

Figure 8: HJ-OpenCL Barrier Example.

```
1  public class OpenCLKernel
2      extends com.amd.aparapi.Kernel {
3      ...
4      int passId;
5      public void run() {
6        // For synchronization point 1
7        if (passId == 0) {
8            method1(gid, j);
9        }
10       // For synchronization point 2
11       if (passId == 1) {
12           method2(gid, j);
13       }
14     }
15 }
```

Figure 9: Barrier Example in OpenCLKernel.class

```
1  cl_int passid;
2  // For method1
3  passid = 0;
4  ret = clSetKernelArg(..., (void *) &passid);
5  ret = clEnqueueNDRangeKernel(...);
6  clEnqueueBarrier(command_queue);
7  // For method2
8  passid = 1;
9  ret = clSetKernelArg(..., (void *) &passid);
10 ret = clEnqueueNDRangeKernel(...);
11 clEnqueueBarrier(command_queue);
```

Figure 10: Generated kernel invocation code of barrier example by the HJ-Compiler

Benckmark	Summary	Data Size	ArrayView?	Next?
Blackscholes	Data-parallel financial application which calculates the price of European put and call options	16,777,216 virtual options	No	No
Crypt	Cryptographic application from the Java Grande Benchmarks[12]	Size C with N= 50,000,000	No	No
MatMult	A standard dense matrix multiplication: $C = A.B$	1024×1024	Yes	No
Doitgen	Multi-resolution analysis kernel from PolyBench[22], ported to Java	128×128×128	Yes	No
MRIQ	Three-dimensional medical benchmark from Parboil[20], ported to Java	large size(64×64×64)	No	No
Syrk	Symmetric rank-1 computation: $C = \alpha.A.A^T + \beta.C$ from PolyBench[22], ported to Java	2048×2048	Yes	No
Jacobi	1-D Jacobi stencil computation from PolyBench[22], ported to Java	T = 50 and N = 134,217,728	No	No
SparseMatmult	Sparse matrix multiplication from the Java Grande Benchmarks [12]	Size C with N = 500,000	No	No
Spectral-norm	Eigenvalue computation using the power method from the Computer Language Benchmarks Game	N = 2,000	No	Yes
SOR	Successive over relaxation from the Java Grande Benchmarks [12]	Size C with N = 2,000	Yes	Yes

Table 1: Details on the benchmarks used to evaluate HJ-OpenCL

5. Performance Evaluation

This section presents experimental results for HJ-OpenCL on two platforms.

The first platform is an AMD *A10-5800K* APU. This APU includes an AMD Radeon HD 7660D GPU with 6 Streaming Multiprocessors(SMs). The CPU of the *A10-5800K* includes 4 cores, 16KB of L1 cache per core, and 32MB of L2 cache. Each SM in the GPU has exclusive access to 32 KB of local scratchpad memory. The CPU and GPU can each directly access system memory, but share bandwidth. While physical memory is shared, it is partitioned between devices such that the CPU has 6GB and the GPU has 2GB. We conducted all experiments on this system using the Java SE Runtime Environment (build 1.6.0_21-b06) with Java HotSpot 64-Bit Server VM (build 17.0-b16, mixed mode).

The second platform has two hexacore Intel X5660 CPUs and two NVIDIA Tesla M2050 discrete GPUs connected over PCIe. There are 4GB of RAM per CPU core, with a total 48GB of ran inside a single node. Each GPU also has approximately 2.5GB of global memory. Only 1 of the 2 available GPUs was used at a time to evaluate this work. In this platform, we used the Java SE Runtime Environment (build 1.6.0_25-b06) with Java HotSpot 64-Bit Server VM (build 20.0-b11, mixed mode).

The ten benchmarks shown in Table 1 were used in our experiments. Four of these benchmarks use array views to perform multi-dimensional array accesses, and two of these benchmarks use the `next` construct for synchronization.

Each benchmark was tested in the following execution modes: sequential Java, sequential HJ, and parallel HJ. The sequential HJ version is equivalent to the Java version except for the use of array views for multi-dimensional array accesses. This is intended as a test of any overhead from the HJ environment. The parallel HJ version employs the `forall` construct to mark parallel loops which can be run in the HJ runtime or on OpenCL devices. The parallel HJ version was executed on three different platforms: 1) on HJ's parallel work-sharing runtime in the JVM using HJ's standard code generation for SMP multicore, 2) on OpenCL CPUs using HJ-OpenCL's code generation and runtime, and 3) on OpenCL GPUs using HJ-OpenCL's code generation and runtime. In the following sections, these three variants are referred to as HJ parallel, HJ OpenCL CPU, and HJ OpenCL GPU, respectively.

Performance is measured by retrieving elapsed nanoseconds from the start of a parallel loop to the completion of all iterations of that loop. The Java system call *System.nanoTime()* was used. This includes the overhead of task spawning and joining for parallel HJ. For OpenCL execution, this includes the overhead of evaluating the `safe` condition and any overhead from JNI or OpenCL API calls (such as communicating data to and from the device). Thus, any performance difference between HJ OpenCL execution and a completely native C++/OpenCL implementation would be from overheads incurred to pass through JNI, which are negligible. Our measurements do not include OpenCL initialization time for creation of OpenCL contexts and command queues.

```
1  boolean isSafe = true;
2  if(M >= row.length) isSafe = false;
3  if(isSafe) {
4    for(int id = 0; id < M; id++) {
5      int row_begin = row[id];
6      int row_end = row[id+1];
7      for(int i =0 ; i < (row_end-row_begin); i++) {
8        if(row_begin + i < 0 || row_begin + i >= val.
             length ||
9           row_begin + i < 0 || row_begin + i >= col.
                length ||
10          col[row_begin+i] < 0 || col[row_begin+i] >=
                x.length) {
11          isSafe = false;
12          break;
13       }
14     }
15     if(!isSafe) break;
16   }
17 }
```

Figure 12: Array Bounds Checking Code for SparseMatmult

5.1 Performance on A10-5800K

Figure 11 shows the speedup numbers on the AMD *A10-5800K* APU relative to the sequential Java version. On the AMD APU system, Blackscholes, Crypt, MRIQ and Jacobi performed as expected across all versions and platforms. By expected, we mean that we see performance improvements for these data-parallel benchmarks from sequential, to parallel HJ, to native OpenCL CPU, to native OpenCL GPU. The result shows speedup of up to 36.71 × for HJ OpenCL CPU and 55.01 × for HJ OpenCL GPU for Jacobi, relative to sequential Java. [2] Doitgen, Syrk, Spectral-norm, and SOR each experience slowdown on HJ OpenCL GPU relative to sequential Java and sequential HJ. Due to runtime overhead of array views sequential HJ is slower than sequential Java for MatMult and Doitgen, both of which each contain multi-dimensional array accesses.

Table 2 shows the performance of each benchmark with exception checking using `safe` relative to without checking (not us-

[2] Some result shows super linear speedups because running natively in OpenCL incurs lower overhead than executing inside the JVM

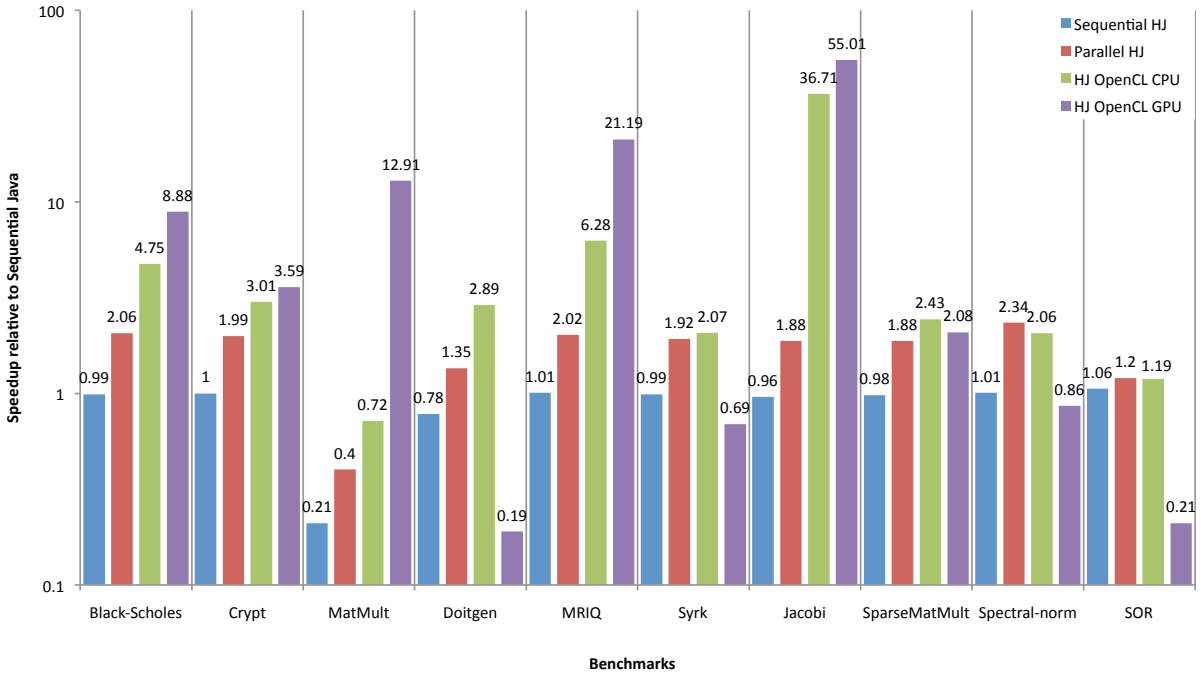

Figure 11: Performance improvents over sequential Java on *A10-5800K*

Platform	Blacksholes	Crypt	MatMult	Doitgen	MRIQ	Syrk	Jacobi	SparseMatmult	Spectral-Norm	SOR
HJ OpenCL CPU	0.99	0.99	1.00	1.04	1.03	0.99	1.00	0.94	0.98	0.98
HJ OpenCL GPU	1.02	0.99	1.00	1.00	1.00	1.00	0.97	0.91	1.00	1.00

Table 2: Slowdown for exception checking on the *A10-5800K*

Platform	Sectral-Norm	SOR
HJ OpenCL CPU	0.10	0.08
HJ OpenCL GPU	0.19	0.29

Table 3: Slowdown for without `next` construct on the *A10-5800K*

ing `safe`) on the *A10-5800K*. Some benchmarks shows performance difference between with checking and without checking due to the overhead of evaluating the `safe` condition. For example, SparseMatMult demonstrates the largest overheads from performing exception checking on both HJ OpenCL CPU (0.94×) and HJ OpenCL GPU (0.91×) due to indirect array accesses in its kernel. Indirect accesses require checking all index values contained in arrays to guarantee no out-of-bounds array accesses. To illustrate this point, Fig. 12 shows the programmer-written array bounds checking code for SparseMatmult.

Spectral-norm and SOR both make use of `next` constructs. Both benchmarks were written in the style shown in Figure 8, a `forall` loop containing barriers. However, an alternative to using all-to-all barrier synchronizations in this style would be to use multiple `forall` loops and have the implicit barrier at the end of each take the place of `next`, as shown in Figure 13. However, this approach

```
1  for (int j = 0; j < iter; j++) {
2    forall (point [i]:[0:n-1]) {
3      method1(i, j);
4    }
5    // synchronization point 1
6    forall (point [i]:[0:n-1]) {
7      method2(i, j);
8    }
9    // synchronozation point 2
10 }
```

Figure 13: HJ-OpenCL Barrier Example without `next` construct

leads to high overhead when using HJ-OpenCL. Multiple `forall` loops would also require multiple calls into JNI. Each call to JNI includes compilation of the OpenCL kernel, transfer of data to

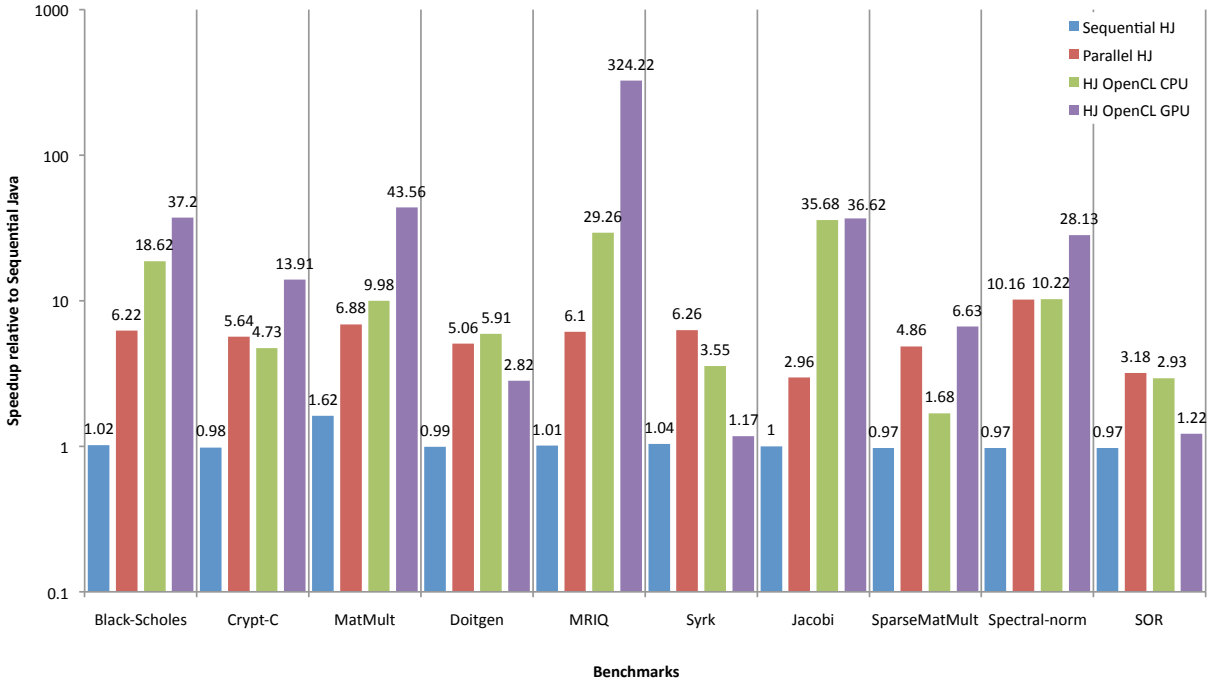

Figure 14: Performance improvents over sequential Java on *Westmere*

Platform	Blacksholes	Crypt	MatMult	Doitgen	MRIQ	Syrk	Jacobi	SparseMatmult	Sectral-norm	SOR
HJ OpenCL CPU	0.98	0.98	0.98	0.99	1.00	1.00	1.00	0.97	1.00	1.02
HJ OpenCL GPU	0.95	0.94	0.99	1.00	0.98	1.00	0.99	0.68	0.99	1.00

Table 4: Slowdown for exception checking on the *Westmere*

Platform	Sectral-Norm	SOR
HJ OpenCL CPU	0.14	0.06
HJ OpenCL GPU	0.35	0.19

Table 5: Slowdown for without `next` construct on the *Westmere*

and from the device, and execution of the kernel. On the other hand, using `next` rather than multiple `forall` loops only incurs the overhead of additional kernel launches. As part of our investigation, we implemented Spectral-norm and SOR in this alternative style. Table 3 shows the relative performance of Spectral-norm and SOR without the `next` relative to with the `next` on the *A10-5800K*. Both benchmarks without `next` are significantly slower (3.4 × - 12.5 ×) than with `next`.

5.2 Performance on Westmere

Figure 14 shows the speedups on the *Westmere* platform with two NVIDIA Tesla GPUs and two hexacore Intel CPUs. For our evaluation on the Intel-NVIDIA system, Blackscholes, Crypt, MatMult, MRIQ, Jacobi and Spectral-norm show expected performance across all versions and platforms. The results show speedup of up to 35.68 × for HJ OpenCL CPU on Jacobi and 324.22 × for HJ OpenCL GPU on MRIQ relative to sequential Java. For the benchmarks Doitgen, Syrk, and SOR HJ OpenCL GPU is slower than HJ OpenCL CPU due to insufficient parallelism.

Table 4 shows the performance of each benchmark with exception checking with the `safe` construct relative to without checking on *Westmere*. Like the previous results on the AMD APU, all benchmarks except SparseMatMult demonstrate low overhead from exception checking using `safe`.

Table 5 shows the performance of Spectral-norm and SOR without the `next` construct relative to with the `next` construct on the *Westmere*. Like the previous results on the AMD APU, both benchmarks demonstrate high overhead when not using `next`.

One obvious conclusion from these results is that benchmarks show very different performance across different platforms due to the architectural features of those platforms. As an example, consider the results on the AMD APU system. Of the ten benchmarks,

two perform best on parallel HJ, four perform best on HJ OpenCL CPU, and four perform best on HJ OpenCL GPU. Parallel HJ provides a load-balancing runtime and avoids added overheads incurred when running in OpenCL. HJ OpenCL CPU has the advantage of native execution and the flexibility of the CPUs architecture. HJ OpenCL GPU also provides native execution, as well as the massive parallelism of GPUs. It is important to note that the HJ application code requires no modifications to switch between these platforms. This enables rapid prototyping and testing across a wider variety of platforms than existing programming models support.

6. Related Work

6.1 GPU code generation from high-level language

GPU code generation is supported by several high-level language compilation systems.

Lime[8] is a JVM compatible language which generates OpenCL code automatically. Lime provides language extensions to Java which express coarse grain tasks and SIMD parallelism. Its compiler generates Java bytecode, JNI glue code, and OpenCL kernels. To enable barrier synchronization, Lime users must partition tasks at each synchronization point.

RootBeer[23] compiles Java bytecode to CUDA by including a data-parallel code region in a method named *gpuMethod* declared inside a class which implements *Kernel*. The RootBeer compiler translates the *gpuMethod()* method to a CUDA kernel (details of this translation has been discussed in Section 2). Similar to Lime, the programmer must partition their kernel to enable barrier synchronization.

X10[7] and Chapel[6] extend their locality controls to target the GPU. Their compilers generate CUDA code when the user explictly specifies a task is to be scheduled on a GPU.

Sponge[11] generates highly optimized CUDA code from the data flow streaming language StreamIt[27]. Once an application is written, programmers can delegate non-trivial optimizations for a wide variety of GPU targets to the compiler.

Firepile[19] performs runtime translation of JVM bytecode from Scala programs to OpenCL kernels. It does this by generating syntax trees from bytecode inspection at runtime, producing OpenCL kernels from traversing these trees similar to how APARAPI performs OpenCL code generation. Firepile is also a library-oriented approach which does not require a special-purpose compiler to perform GPU code generation.

Clyther[25] is similar to APARAPI, but is used from Python. It performs runtime GPU code generation for specially annotated Clyther functions which only use a subset of the Python language via a library-oriented approach.

6.2 Exception semantics in Java

For those programming models which enable hybrid JVM and GPU execution (i.e. Lime, RootBeer, JCUDA), preserving Java exception semantics is an important challenge which has largely been ignored up to this point. Some previous work focused on eliminating redundant checks for null pointers and out-of-bounds accesses, enabling optimized or native execution without breaking Java exception semantics.

RootBeer[23] supports CUDA code generation for user-thrown exceptions, and in the absence of a *try..catch* block bubbles execution up the stack until one is found or the kernel exits. However, no support is provided for catching the runtime exceptions which HJ-OpenCL focuses on.

Artigas et al.[2] and Moreira et al.[17] achieve MATMULT Java performance which is at least 80% of the peak Fortran performance using these optimizations. This work generates exception-safe and -unsafe regions of code. In exception-safe regions the compiler can perform aggressive loop optimization such as loop tiling without breaking exception semantics.

Würthinger et al. [28] proposes an algorithm on Static Single Assignment(SSA) form for the JIT compiler which eliminates unnecessary bounds checking.

ABCD[3] provides an array bounds checking elimination algorithm by adding edges to the SSA graph.

Jeffery et al.[24] proposed a static annotation framework to reduce the overhead of dynamic checking in the JIT compiler.

7. Conclusions

In this paper, we presented Habanero-Java with OpenCL generation (HJ-OpenCL). HJ-OpenCL facilitates OpenCL utilization from a parallel JVM program. The HJ-OpenCL language and compiler enable:

1. Automatic generation of OpenCL kernels and JNI glue code from a Java parallel-for construct, (`forall`).
2. Efficient access to regular, multi-dimensional arrays in Java kernels intended for OpenCL execution using the `array view` construct.
3. Support for all-to-all barriers in OpenCL kernels using the `next` language construct.

Unlike previous work, our approach also preserves Java exception semantics by generating code with *exception-safe* and *unsafe* regions using the `safe` language construct.

Performance evaluation on multiple heterogeneous platforms demonstrated the advantages of flexibly executing a single Java application across multiple execution modes. On an AMD APU accessed from OpenCL, our results show speedups of up to 36.7× relative to sequential Java on the host 4-core CPU, and of up to 55.0× on an integrated GPU. For a system with an Intel Xeon CPU and a discrete NVIDIA Fermi GPU, OpenCL execution demonstrates performance improvement up to 35.7× for the 12-core CPU and 324.0× for the GPU, relative to sequential Java. However, it is also important to note that some benchmarks tested performed best when executed on the HJ runtime due to OpenCL overheads and the benefits of load-balancing in the HJ runtime.

This work combines the best of managed and native execution. Using managed languages like Java, programmers gain programmability, portability, extensive libraries, and high-level language features. However, this work and others have demonstrated lackluster performance in the JVM on computationally heavy workloads. On the other hand, native execution in OpenCL not only uses native threads to eliminate overheads from managed execution but also facilitates execution on a variety of architectures. HJ-OpenCL combines the parallel HJ programming language, a multi-threaded JVM runtime, and OpenCL execution to enable complete portability, rapid prototyping, and transparent execution of Java applications across all OpenCL platforms.

Acknowledgements

We would like to thank members of the Habanero group at Rice University for valuable discussions related to this work. The Westemere results in this paper were obtained on the Rice DAVinCI system, which was supported in part by the Data Analysis and Visualization Cyberinfrastructure award funded by NSF under grant OCI-0959097.

References

[1] APARAPI. API for Data Parallel Java. http://code.google.com/p/aparapi/.

[2] Pedro V. Artigas, Manish Gupta, Samuel P. Midkiff, and José E. Moreira. Automatic loop transformations and parallelization for java. In *Proceedings of the 14th international conference on Supercomputing*, ICS '00, pages 1–10, New York, NY, USA, 2000. ACM.

[3] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. Abcd: eliminating array bounds checks on demand. *SIGPLAN Not.*, 35(5):321–333, May 2000.

[4] Vincent Cavé et al. Habanero-Java: the New Adventures of Old X10. In *PPPJ'11: Proceedings of 9th International Conference on the Principles and Practice of Programming in Java*, 2011.

[5] Satish Chandra et al. Type inference for locality analysis of distributed data structures. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 11–22, New York, NY, USA, 2008. ACM.

[6] Chapel. The Chapel language specification version 0.4, February 2005.

[7] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[8] Christophe Dubach, Perry Cheng, Rodric Rabbah, David F. Bacon, and Stephen J. Fink. Compiling a high-level language for gpus: (via language support for architectures and compilers). In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 1–12, New York, NY, USA, 2012. ACM.

[9] OpenACC Directives for accelerators. Openacc. http://www.openacc-standard.org/.

[10] Yi Guo et al. Work-First and Help-First Scheduling Policies for Async-Finish Task Parallelism. In *IPDPS '09: International Parallel and Distributed Processing Symposium*, 2009.

[11] Amir H. Hormati, Mehrzad Samadi, Mark Woh, Trevor Mudge, and Scott Mahlke. Sponge: portable stream programming on graphics engines. *SIGPLAN Not.*, 46(3):381–392, March 2011.

[12] JGF. The Java Grande Forum benchmark suite. http://www.epcc.ed.ac.uk/javagrande/javag.html.

[13] Mackale Joyner, Zoran Budimlić, and Vivek Sarkar. Subregion analysis and bounds check elimination for high level arrays. In *Proceedings of the 20th international conference on Compiler construction: part of the joint European conferences on theory and practice of software*, CC'11/ETAPS'11, pages 246–265, Berlin, Heidelberg, 2011. Springer-Verlag.

[14] Khronos OpenCL Working Group. The OpenCL Specification v1.2. 2012.

[15] khronos.org. Opencl. http://www.khronos.org/opencl/.

[16] Roberto Lublinerman et al. Delegated Isolation. In *OOPSLA '11: Proceeding of the 26th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, 2011.

[17] José E. Moreira, Samuel P. Midkiff, and Manish Gupta. From flop to megaflops: Java for technical computing. *ACM Trans. Program. Lang. Syst.*, 22(2):265–295, March 2000.

[18] Nvidia. NVidia CUDA Programming Guide version 1.0. http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf, 2007.

[19] Nathaniel Nystrom, Derek White, and Kishen Das. Firepile: run-time compilation for gpus in scala. *SIGPLAN Not.*, 47(3):107–116, October 2011.

[20] Parboil. Parboil benchmarks. http://impact.crhc.illinois.edu/parboil.aspx.

[21] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java concurrency in practice*. Addison-Wesley Professional, 2005.

[22] PolyBench. The polyhedral benchmark suite. http://www.cse.ohio-state.edu/ pouchet/software/polybench.

[23] P.C. Pratt-Szeliga, J.W. Fawcett, and R.D. Welch. Rootbeer: Seamlessly using gpus from java. In *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 375–380, June.

[24] Jeffery Von Ronne, Andreas Gampe, David Niedzielski, and Kleanthis Psarris. Safe bounds check annotations. In *Concurrency and Computations: Practice and Experience, Vol. 21, No. 1*, 2009.

[25] Sean Ross-Ross. Clyther: a python just-in-time specialialization engine for OpenCL.

[26] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 277–288, New York, NY, USA, 2008. ACM.

[27] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196, London, UK, UK, 2002. Springer-Verlag.

[28] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. Array bounds check elimination for the java hotspot client compiler. In *Proceedings of the 5th international symposium on Principles and practice of programming in Java*, PPPJ '07, pages 125–133, New York, NY, USA, 2007. ACM.