Accelerating 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

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

Jisheng Zhao  
Department of Computer Science  
Rice University  
Houston, TX, USA  
jisheng.zhao@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 OpenCL execution 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 [Concurrency and Parallelism]: Parallel Programming

General Terms Languages, Design

Keywords GPGPU, OpenCL, Java, Habanero-Java

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 HJ 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.

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. 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 of Array-Sum in Figure 2. In Rootbeer, any computation to be executed on a CUDA device must be encapsulated in a class which implements Rootbeer’s Kernel interface. This interface contains a method called gpuMethod(), 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.

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 clear that using Rootbeer dramatically increases code length (approximately 2x) and decreases readability by separating the computational kernels from the application code. The parallelism in this program is not readily apparent, hidden by a sequential loop that adds Kernel objects to the jobs list. It is also important to note that Rootbeer ignores Java exception semantics during GPU execution, and exceptions such

```
public class ArraySum {
    public static void main(String[] args) {
        int[][] arrays = new int[N][M];
        int[] result = new int[N];

        for (int i=0; i < N; i++) {
            result[i] = 0;
            for (int j=0; j < M; j++) {
                result[i] += arrays[i][j];
            }
        }
    }
}
```

Figure 1: Original Array-Sum Java Code. Note that while this code is short and simple, it is also \( O(N^2 M) \) where \( N \) is the number of arrays to sum and \( M \) is the length of each array.
as IndexOutOfBoundsException and NullPointerException 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 arrays is declared at line 6. This array view object provides two-dimensional indexing (e.g., arrays[1, j]) into a one-dimensional “base” array base. The use of a one-dimensional base array results in more efficient serialization when communicating with OpenCL devices. The second change, the addition of a safe block in line 8, enables the programmer to preserve Java exception semantics. A safe statement takes a boolean condition, and asserts that no exception will be 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 that can be selected for execution based on the evaluation of the condition. 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.

```java
public class ArraySum {
    public static void main(String[] args) {
        int[][] arrays = new int[N][M];
        int[] result = new int[N];
        ... initialization ...
        List<Kernel> jobs = new ArrayList<Kernel>();
        for (int i = 0; i < N; i++) {
            jobs.add(new ArraySumKernel(arrays[i], result, i);
        }
        Rootbeer rootbeer = new Rootbeer();
        rootbeer.runAll(jobs);
    }
}
```

Figure 2: Rootbeer Implementation.

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

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 permitted in HJ programs. HJ-OpenCL only supports OpenCL execution for a subset of the parallel language features of Habanero-Java, though the full HJ language can still be used on multiple CPU cores within the JVM. Work to date on HJ-OpenCL has focused on supporting forall parallel loops and next barrier statements for OpenCL execution:

1. forall: The statement “forall(point : 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:N-1,0:M-1] for a 2-D iteration space. Each iteration instance executes the loop body {stmt} given a different point in the iteration space, i.e., region. All forall loops end with an implicit barrier (finish scope).
2. next: The statement next represents an all-to-all barrier synchronization point with both signal and wait semantics in a parallel forall loop.

3.2 The Safe Construct

We propose the addition of a safe language construct to Habanero-Java. Programmers can use the safe language feature to assert conditions which ensure no exception will be thrown in a code region. The “safe (boolean cond) {stmt}” construct indicates that {stmt} is asserted to not 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 arrayViews, a programmer would put isSafe = (result != null & &

\[1\] HJ-OpenCL only supports the all-to-all barrier synchronization in phases due to OpenCL constraints. (See Section 4.2.)
arrays != null && arraysView != null). It would also be necessary to add \( N \cdot I \leq \text{result.length} \) and \( N^*M-I \leq \text{arrays.length} \) for array bounds checking. The schema for this approach is shown in Figure 3. 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. Note that this approach can be used to accelerate performance both on CPUs and on GPUs due to the use of OpenCL instead of Java, while still preserving exception semantics.

### 3.3 Support for Rectangular Multi-Dimensional Arrays

The OpenCL specification [14] does not support the use of Java-style and C-style multi-dimensional arrays in OpenCL kernels. However, computationally intensive Java programs often use rectangular multi-dimensional arrays as a basic data structure. Therefore, it is important to provide software developers with language support in HJ-OpenCL for rectangular 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 array at the Java level. 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 a Fortran-like comma-separate multi-dimensional array syntax, with bounds defined by `region`. The details of supporting array views at compile- and runtime in HJ-OpenCL are discussed in Section 4.3.

### 4. OpenCL Generation

While HJ-OpenCL provides language features which 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 to take JVM bytecode as input and produce OpenCL kernels and the required glue code (i.e. the JNI stub).

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 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 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 code generation steps. 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 an HJ program (see Section 3) as input, and produces the following outputs:

1. Java CLASS files intended for execution in the 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 the bytecode 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 \texttt{forall} loop, and replaces the original \texttt{forall} loop with a native method named \texttt{openCLKernel0()} (line 9). This native method invokes the OpenCL API to execute computations on an OpenCL device. Its definition is contained in a library, \texttt{libCalc} (line 2). These transformations translate the original HJ program with \texttt{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 \texttt{index[i] = 0;} in Figure 3 might throw IndexErrorOutOfBoundsException 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 \texttt{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 from is specified by the safe 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 safe block. For example, in Figure 3 the programmer has set the value of \texttt{isSafe} to \texttt{true} at the very beginning of the input HJ program (line 29-32). If these conditions hold at runtime, HJ-OpenCL assumes no exception will be thrown inside the safe block. For example, in Figure 3 the programmer has set the value of \texttt{isSafe} to \texttt{true} at the very beginning of the input HJ program (line 29-32).

The code generation process involves the following steps to run the ArraySum example in OpenCL:

1. The JNI call \texttt{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 to the OpenCL device (line 22).
3. The ArraySum kernel is compiled and executed on an OpenCL device (line 23).
4. The OpenCL API is invoked to retrieve results from the device if they are marked as output (line 29-32).

A powerful feature of Java is runtime exceptions, thrown when some safety constraints are violated. For example, the array access \texttt{index[i] = 0;} in Figure 3 might throw IndexErrorOutOfBoundsException 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 \texttt{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 from is specified by the safe 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 safe block. For example, in Figure 3 the programmer has set the value of \texttt{isSafe} to \texttt{true} at the very beginning of the input HJ program (line 29-32). If these conditions hold at runtime, HJ-OpenCL assumes no exception will be thrown inside the safe block. For example, in Figure 3 the programmer has set the value of \texttt{isSafe} to \texttt{true} at the very beginning of the input HJ program (line 29-32).

The code generation process involves the following steps to run the ArraySum example in OpenCL:

1. The JNI call \texttt{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 to the OpenCL device (line 22).
3. The ArraySum kernel is compiled and executed on an OpenCL device (line 23).
4. The OpenCL API is invoked to retrieve results from the device if they are marked as output (line 29-32).

A powerful feature of Java is runtime exceptions, thrown when some safety constraints are violated. For example, the array access \texttt{index[i] = 0;} in Figure 3 might throw IndexErrorOutOfBoundsException 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 \texttt{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 from is specified by the safe 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 safe block. For example, in Figure 3 the programmer has set the value of \texttt{isSafe} to \texttt{true} at the very beginning of the input HJ program (line 29-32). If these conditions hold at runtime, HJ-OpenCL assumes no exception will be thrown inside the safe block. For example, in Figure 3 the programmer has set the value of \texttt{isSafe} to \texttt{true} at the very beginning of the input HJ program (line 29-32).
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 new statement: `forall`.

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 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.

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).

The paragraphs above have covered the generation of all bytecode and JNI stubs 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 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 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 new statement: `forall`.

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 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.

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 `arraysView` which maps a two-dimensional region of size `[0:N-1,0:M-1]` to the one-dimensional array `arrays`. The array `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]`. 

---

Figure 7: Example code of `OpenCLKernel.class`

```java
public class OpenCLKernel
extends com.amd.aparapi.Kernel {
    int[] result, array;
    int OpenCLKernelSize;
    public OpenCLKernel();
    public void run() {
        int gid = this.getGlobalId(0);
        if (gid != OpEnCl.size) { return; }
        for (int j = 0; j < M; j++) {
            result[gid] += array[gid*M + j];
        }
    }
}
```

Figure 8: HJ-OpenCL Barrier Example.

```java
public class OpenCLKernel
extends com.amd.aparapi.Kernel {
    int passId;
    public void run() {
        // For synchronization point 1
        if (passId == 0) {
            method1(gid, j);
        }
        // For synchronization point 2
        if (passId == 1) {
            method2(gid, j);
        }
    }
}
```

Figure 9: Barrier Example in `OpenCLKernel.class`

```java
cl_int passId;
// For method1
passId = 0;
ret = clSetKernelArg(. . . , (void *) &passId);
return = clEnqueueNDRangeKernel(. . . );
clEnqueueBarrier(command_queue);
// For method2
passId = 1;
ret = clSetKernelArg(. . . , (void *) &passId);
return = clEnqueueNDRangeKernel(. . . );
clEnqueueBarrier(command_queue);
```

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

### 5.1 Performance on A10-5800K

Figure 12 shows the speedup numbers on the AMD A10-5800K APUs 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. Doitgen, Syrck, Spectral-norm, andSOR 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 Doit-

---

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

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

---

```
boolean isSafe = true;
if (M >= row.length) isSafe = false;
for (int id = 0; id < M; id++) {
    int row_begin = row[id];
    int row_end = row[id+1];
    for (int i = 0; i < (row_end - row_begin); i++) {
        if(row_begin + i < 0 || row_begin + i >= val.length) {
            row_begin + i >= val.length} {
                col[row_begin+i] < 0 || col[row_begin+i] >= x.length) {
                    isSafe = false;
            break;
        }
    }
}
```

Figure 11: Array Bounds Checking Code for SparseMatmult

APARAPI is then able to translate this statement to an OpenCL kernel. APARAPI does not support multi-dimensional arrays.

### 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 GPU 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 ram 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).

---

2 Some result shows super linear speedups because running natively in OpenCL incurs lower overhead than executing inside the JVM.
gen, both of which contain multi-dimensional array accesses. Table 2 shows the performance of each benchmark with exception checking using `safe` relative to without checking (not using `safe`) on the A10-5800K. Some benchmarks show 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, Figure 11 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-

<table>
<thead>
<tr>
<th>Platform</th>
<th>Blacksholes</th>
<th>Crypt</th>
<th>MatMult</th>
<th>Doitgen</th>
<th>MRIQ</th>
<th>Syrk</th>
<th>Jacobi</th>
<th>SparseMatMult</th>
<th>Spectral-Norm</th>
<th>SOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>HJ OpenCL CPU</td>
<td>0.99</td>
<td>0.99</td>
<td>1.00</td>
<td>1.04</td>
<td>1.03</td>
<td>0.99</td>
<td>1.00</td>
<td>0.94</td>
<td>0.98</td>
<td>0.98</td>
</tr>
<tr>
<td>HJ OpenCL GPU</td>
<td>1.02</td>
<td>0.99</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.97</td>
<td>0.91</td>
<td>0.98</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Platform</th>
<th>Spectral-Norm</th>
<th>SOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>HJ OpenCL CPU</td>
<td>0.10</td>
<td>0.08</td>
</tr>
<tr>
<td>HJ OpenCL GPU</td>
<td>0.19</td>
<td>0.29</td>
</tr>
</tbody>
</table>

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

```java
for (int j = 0; j < iter; j++) {
    forAll (point [j]:[0:n-1]) {
        method1(i, j);
    }
    // synchronization point 1
}
forAll (point [j]:[0:n-1]) {
    method2(i, j);
    // synchronization point 2
}
```

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

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
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 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.

![Figure 14: Performance improvements over sequential Java on Westmere](image)

<table>
<thead>
<tr>
<th>Platform</th>
<th>Blacksholes</th>
<th>Crypt-C</th>
<th>MatMult</th>
<th>Doitgen</th>
<th>MRIQ</th>
<th>Syrk</th>
<th>Jacobi</th>
<th>SparseMatMult</th>
<th>Spectral-norm</th>
<th>SOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>HJ OpenCL CPU</td>
<td>0.98</td>
<td>0.98</td>
<td>0.98</td>
<td>0.99</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.97</td>
<td>1.00</td>
<td>1.02</td>
</tr>
<tr>
<td>HJ OpenCL GPU</td>
<td>0.95</td>
<td>0.94</td>
<td>0.99</td>
<td>1.00</td>
<td>0.98</td>
<td>1.00</td>
<td>0.99</td>
<td>0.68</td>
<td>0.99</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Table 4: Slowdown for exception checking on the Westmere

<table>
<thead>
<tr>
<th>Platform</th>
<th>Sectral-Norm</th>
<th>SOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>HJ OpenCL CPU</td>
<td>0.14</td>
<td>0.06</td>
</tr>
<tr>
<td>HJ OpenCL GPU</td>
<td>0.35</td>
<td>0.19</td>
</tr>
</tbody>
</table>

Table 5: Slowdown for without next construct on the Westmere
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 gpamethod declared inside a class which implements Kernel. The RootBeer compiler translates the gpamethod method to a CUDA kernel (details of this translation have 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 explicitly specifies a task is to be scheduled on a GPU node.

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.

In ABCD [3], Bodik et. al provides an array bounds checking elimination algorithm, which is based on graph traversal on an extended SSA form.

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 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 for 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


