HJ-OpenCL: Reducing the Gap Between the JVM and Accelerators

Max Grossman
Rice University
6100 Main St
Houston, TX, USA
jmg3@rice.edu

Shams Imam
Rice University
6100 Main St
Houston, TX, USA
shams@rice.edu

Vivek Sarkar
Rice University
6100 Main St
Houston, TX, USA
shams@rice.edu

ABSTRACT

Recently there has been increasing interest in supporting execution of Java Virtual Machine (JVM) applications on accelerator architectures, such as GPUs. Unfortunately, there is a large gap between the features of the JVM and those commonly supported by accelerators. Examples of important JVM features include exceptions, dynamic memory allocation, use of arbitrary composite objects, file I/O, and more. Recent work from our research group tackled the first feature in that list, JVM exception semantics[14]. This paper continues along that path by enabling the acceleration of JVM parallel regions that include object references and dynamic memory allocation.

The contributions of this work include 1) serialization and deserialization of JVM objects using a format that is compatible with OpenCL accelerators, 2) advanced code generation techniques for converting JVM bytecode to OpenCL kernels when object references and dynamic memory allocation are used, 3) runtime techniques for supporting dynamic memory allocation on OpenCL accelerators, and 4) a novel redundant data movement elimination technique based on inter-parallel-region dataflow analysis using runtime bytecode inspection.

Experimental results presented in this paper show performance improvements of up to 18.33× relative to parallel Java Streams for GPU-accelerated parallel regions, even when those regions include object references and dynamic memory allocation. In our evaluation, we fully characterize where accelerators or the JVM see performance wins and point out opportunities for future work.

Keywords

JVM, OpenCL, serialization, GPU, offload

1. MOTIVATION

The Java Virtual Machine (JVM) is arguably the most common execution platform for managed runtimes. The abstractions, efficiency, security, portability, and other features offered by the JVM make it an attractive target for many high-level languages such as Java, Scala, and Clojure. These high-level frontends allow domain experts to quickly write application code that can target a wide range of platforms with reasonable computational efficiency.

Despite two decades of work investigating and reducing the performance overheads of running in the JVM, there still remain limitations on overall JVM performance and performance repeatability [22, 2]. Garbage collection (GC) can introduce overheads and causes jitter in application latency when GC pauses interfere with application threads. Dynamic class loading and method invocation add overheads to method dispatch. JIT compilation, while beneficial in the long term, also steals cycles from the user application. In addition, the abstraction layer of the JVM makes it challenging for programmers to control data layout.

Despite these limitations, the JVM is increasingly being used to execute performance-sensitive applications. JVM-based languages are being used in finance [8], in the natural sciences [23], in web servers [1], and in many other contexts. For many organizations, the portability, programmability, availability, large community, and security of the JVM make it an attractive target for new application development. While the JVM may not utilize hardware resources as efficiently as an optimized native application, JVM performance and latency is usually considered adequate in light of the productivity benefits that JVM languages provide.

Because JVM abstractions can make single-threaded optimizations challenging, one of the most common optimizations for JVM-based applications is to move from a single-threaded to a multi-threaded implementation. Multi-core architectures are now the standard in all hardware platforms, and many-core architectures are becoming more varied and widely available. Due to this trend, the latest generation of JVM-based programming models and frameworks have generally started to include intrinsically parallel operations as first-class citizens (e.g. Apache Spark, Clojure Futures, Scala Parallel Collections). Even existing JVM languages have expanded their support for parallelism in recent revisions, most notably Java 8 with Java Streams.

One academic example of this expanded support for parallelism is the HJlib parallel programming library[16]. HJlib uses Java 8 lambdas to support a library-based approach to parallel programming, rather than a language-based one. For example, an asynchronous parallel task is created using the following library call:

```
async(() -> { ... });
```


DOI: 10.1145/1235

rather than using a syntactic language construct as follows[6]:

```
async { ... }
```

HJlib executes parallel work on the Habanero Java work-stealing runtime[5] or the cooperative parallel runtime[15] while supporting a wide range of parallel constructs, including fork-join parallelism, parallel-for loops, actors, and eurekas[17]. HJlib is used as both a research tool, and an educational tool to introduce students to parallel programming at the sophomore level[24].

While JVM languages and frameworks now support increased levels of parallelism, and multi-core architectures are more widely available, one significant hardware domain that has remained largely beyond the purview of the JVM is accelerators, such as GPUs. Accelerators usually have limited or no support for disk I/O, network I/O, virtual memory, complex/dynamic execution, and other basic facilities that are standard in most computing systems today. As such, the functionality and efficiency of a full JVM implementation on accelerators would be limited. However, using the computational power of accelerators to offload selected parallel code regions from a JVM program can help accelerate JVM applications while offering a simpler and more familiar programming interface for accelerators.

Today, the most portable programming framework for accelerators is OpenCL. OpenCL is an open standard that defines a universal, data-parallel programming model for working with a broad range of accelerators. Currently, OpenCL implementations exist for Graphics Processing Units (GPUs), Central Processing Units (CPUs), Field Programmable Gate Arrays (FPGAs), and the Xeon Phi Coprocessor. OpenCL's portability among different accelerators mirrors the JVM's portability among different, conventional computing platforms. Hence, it is natural to pair the JVM with OpenCL over other accelerator programming models.

However, while an expert OpenCL developer can produce an order of magnitude or more performance improvement with the right hardware and the right kernels, OpenCL is generally considered to be challenging to work with for most application developers. Using OpenCL requires explicit memory management on both the host processor (generally a CPU) as well as one or more attached accelerators. OpenCL also places the burden of platform management, runtime kernel compilation, and task dependency declarations on the programmer. The OpenCL kernel language is syntactically C but with limitations, and OpenCL's more complex memory model can be challenging to learn and use.

Due to the relative performance limitations of JVM execution, the widespread adoption of the JVM as a universal execution platform, and the lack of programmability in OpenCL and other accelerator programming models, we believe that bridging the gap between high-level JVM-based programming languages and accelerators is an important and open research problem. This problem is complicated by the need to support legacy JVM code that was written to target the JVM (i.e., it was not written to use a restricted subset of the JVM bytecode specification with hardware acceleration in mind).

In this work, we continue past work on maintaining JVM semantics when executing parallel regions on an OpenCL accelerator. Our approach focuses on using code generation techniques from JVM bytecode to the OpenCL kernel language to enable the acceleration of user-written computational kernels in any JVM-based programming language. Parallel code regions are defined as lambdas passed to the parallel-for API of the Habanero Java Parallel Library (HJlib) [16], though this approach can also be applied to the Java Streams API. In particular, this work makes the following novel contributions:

1. Efficient serialization and deserialization of JVM objects at runtime to a format amenable to storage on an OpenCL accelerator.
2. Support for dynamic memory allocation on accelerators using a concurrent heap and kernel retry to handle out-of-memory errors.
3. Removal of redundant data movement between CPU cores and accelerators by inspecting the dataflow between parallel regions in the calling context at runtime.

The rest of the paper is organized as follows: Section 2 discusses past work on this topic, and how this work extends the existing literature in this area. Section 3 describes details of the techniques and algorithms used in our approach. Section 4 presents an experimental evaluation of the performance improvement from our approach, relative to the HJlib parallel programming library and parallel Java Streams. Section 5 discusses limitations of our current approach, which in turn offer opportunities for future work. Section 6 wraps up with a summary of contributions, conclusions, and future work.

2. PAST WORK

To the best of our knowledge, the earliest known tools to enable JVM programs to execute on accelerators are Rootbeer [21] and APARAPI [10].

Rootbeer offloads JVM computations to NVIDIA GPUs by performing static code generation from JVM bytecode to NVIDIA CUDA kernels, thereby limiting its scope to only include JVM applications that do not rely on dynamic class loading. The body of a parallel region is written as a method inside a user-written class that extends the Rootbeer `Kernel` interface. A parallel region is created by invoking Rootbeer-specific library methods on the user-written `Kernel` class. Rootbeer supports limited JVM object references in parallel regions executed on the GPU. The Rootbeer paper [21] goes into detail on the object storage format used for GPUs. However, it does not describe the process of serializing and deserializing JVM objects, a large source of overhead that we address in detail in this paper. Rootbeer does support dynamic memory allocation in CUDA but does not support parallel regions whose memory allocations exceed the GPU's physical memory capacity. Because GPUs lack virtual memory and usually have a smaller memory capacity than their host systems, this represents a limitation on the applications that can successfully complete without an `OutOfMemoryException` under Rootbeer. Our approach to dynamic memory allocation addresses this limitation. Rootbeer also does not describe any optimization of the transfers performed to and from the accelerator. The lack of optimization limits the kernels that can benefit from Rootbeer to only those whose computational speedup from GPU execution can offset the serialization and transfer overheads.

APARAPI differs from Rootbeer in many respects. First, rather than using static code generation at compile-time,

APARAPI dynamically translates JVM bytecode to accelerator kernels at runtime. This approach supports dynamic loading of external classes and ensures it never executes stale code on the accelerator. APARAPI also targets OpenCL rather than CUDA, thereby supporting a wider range of accelerators. However, Rootbeer supports a greater subset of the JVM bytecode specification (e.g., object references, dynamic memory allocation). Both Rootbeer and APARAPI share a similar API, using a special `Kernel` interface to store the body of parallel regions that will be executed on an accelerator.

In general, APARAPI and Rootbeer share some limitations. Both rely on using external classes to define the bodies of parallel regions, thus obfuscating the parallelism within an application. Both do little in terms of optimizing the execution of accelerator kernels in the context of the host application. Both have little or no support for dynamic memory allocation, which severely limits the JVM applications and kernels they can be used for.

Past work by our research group looked at addressing some inconsistencies between JVM and accelerator execution. The work in [13] started the development of HJ-OpenCL by offloading parallel `for` loops written in the Habanero Java (HJ) parallel programming language to a GPU by extending the code generation module in APARAPI. This work included a number of extensions such as more efficient support for multi-dimensional arrays, global barrier synchronization in parallel regions executing on the accelerator, and avoiding accelerated execution of parallel regions that may throw `Exception`s. [14] extended the work on HJ-OpenCL by improving support for JVM exception semantics on accelerators. [12] included work on extending support for more complex data structures on accelerators, most notably sparse vectors.

Ongoing work in Project Sumatra [20] is building support for GPU code generation in to the JVM itself. Working from inside the JVM may enable optimizations that are not possible externally and allow for more seamless integration with existing parallel programming interfaces like Java Streams. In 2013, AMD demonstrated a prototype based on the Heterogeneous System Architecture (HSA) which did not support dynamic memory allocation or exceptions [9].

In [11], a functional array-based API for GPU programming from the JVM is presented. This API is similar to Scala Parallel Collections and Java Streams. While our work focuses on the acceleration of parallel for loops, it would be a natural and straightforward extension to use a more functional API like the one in [11].

JaBEE [25] accelerates JVM computational kernels in the J3 JVM using NVIDIA GPUs. JaBEE manipulates JVM-internal data structures to support object references on the GPU, including nested object references stored in fields of other objects. JaBEE also supports dynamic memory allocation on the GPU. However, the performance evaluation of JaBEE's object support shows that GPU execution is slower than the JVM when object references are present. The support for dynamic memory allocation is limited to programs whose allocations are smaller than GPU memory (like Rootbeer), and relies on CUDA's dynamic memory allocator. JaBEE's construction on CUDA limits its scope to NVIDIA GPU accelerators, and the use of JVM internals means it can only execute applications run on the J3 JVM.

The work described in this paper continues the work on HJ-OpenCL started by [13] and [14] by addressing the challenges of supporting accelerated execution of parallel regions which contain JVM object references. In particular, object reference support introduces several new challenges:

1. The dynamic allocation of object types on accelerators that do not natively support dynamic memory allocation and include only a limited amount of DRAM.

2. The representation of JVM object types in a format that the accelerator can access and modify.

3. The serialization and deserialization of JVM objects and arrays of objects.

4. The mitigation of overheads due to JVM object serialization and deserialization.

3. OVERVIEW OF OUR APPROACH

In this section, we detail the contributions of this work. We start by describing the existing frameworks which this work builds on. Then, we discuss how JVM objects containing primitive fields are represented as C-style `struct`s on OpenCL accelerators. Following that, we describe in detail the process of serializing and deserializing JVM objects using the `sun.misc.Unsafe` package, as well as optimizations made to reduce unnecessary data transfers and serialization between the JVM and accelerator. Finally, we describe the techniques introduced in this work to support dynamic memory allocation on OpenCL accelerators when the amount of physical memory is insufficient to satisfy all threads' memory allocation requests.

In this work, we extend APARAPI's existing support for OpenCL kernel generation from JVM bytecode. While APARAPI's code generation module supports a reasonable subset of the JVM bytecode specification, many of the JVM's most commonly used features (e.g., object references, the `NEW` opcode) are not supported in APARAPI kernels. This work removes some of these limitations, while reusing much of the core code generation framework.

One of the less usable aspects of APARAPI and Rootbeer is the way in which the body of a parallel region is defined. Both require the programmer to extend a `Kernel` interface similar to Java's `Runnable` class. Code generation techniques are then applied to the `run` method defined inside. While this is a common pattern in many legacy Java programs, it can obfuscate the parallelism in complex parallel JVM applications. With the introduction of lambda functions in Java 8, we can more clearly link the sites at which parallelism is created with the body of the parallel computation. In this work, we extend the blocking parallel-for construct, `forall`, from the Java 8 parallel library HJlib[16] to automatically offload its computational body to OpenCL accelerators. The computation of HJlib's `forall` API is defined with a Java lambda as follows:

```
1 forall(0, niters - 1, (iter) -> {
2     ... // loop body as a function of iter
3 });
```

This work adds the construct `forall_acc` to HJlib, which has the same semantics as `forall` (blocking, all iterations run in parallel) but executes the body of the lambda on an OpenCL accelerator. Using Java lambdas maintains the

clear link between the creation of parallelism and its computation. Future work could include a unified `forall` construct which auto-selects the accelerator based on lambda characteristics and datasize using techniques from past work on auto-scheduling in [12] or [4].

Additionally, we use The RetroLambda[3] tool at compile time to convert Java 8 JVM bytecode that uses the new `INVOKEDYNAMIC` opcode to instead use an inner class to store the closure for a lambda. This is not visible to the programmer, but is necessary for compatibility with APARAPI.

3.1 Storing JVM objects in OpenCL

This work focuses on supporting JVM object references in accelerated parallel regions, but limits the current scope of that work to only support references to primitive fields in those objects, or instance methods which only reference primitive fields. For example, it would be valid to reference objects of the type `Point` defined below so long as those references only used the methods `getX`, `getY`, `getZ`, or `distance`. Note that methods with object references as parameters and local variables are supported – the main constraint is that the method is not permitted to access a non-primitive field during GPU execution. Hence, use of the methods `getClosest` and `distanceToClosest` is not supported in accelerated parallel regions.

```
1  package edu.rice.hj.example.Point;
2
3  public class Point {
4    private float x, y, z;
5    Point closest;
6
7    public Point(float x, float y, float z, Point
         closest) {
8      this.x = x; this.y = y; this.z = z;
9      this.closest = closest;
10   }
11
12   public float getX() { return x; }
13   public float getY() { return y; }
14   public float getZ() { return z; }
15   public float distance(Point other) {
16     return (float)Math.sqrt(Math.pow(other.x - x, 2) +
           Math.pow(other.y - y, 2) + Math.pow(other.z -
           z, 2));
17   }
18
19   public Point getClosest() { return closest; }
20   public float distanceToClosest() {
21     return distance(closest);
22   }
23 }
```

The primitive fields of a JVM class referenced from an accelerated parallel region are represented by a corresponding `struct` in the automatically generated OpenCL kernel definition. For each primitive field in the JVM object, a similarly typed field is created in a `struct` definition auto-generated by the modified APARAPI code generator. An example of the `struct` generated for the `Point` class above is shown below:

```
1  typedef struct __attribute__ ((packed))
       edu_rice_hj_example_Point_s {
2    float x;
3    float y;
4    float z;
5  } edu_rice_hj_example_Point;
```

Note that this conversion is only carried out for object types which are used in an accelerated parallel region, not for all classes loaded by an HJlib application. If a referenced object has a superclass whose primitive fields are also referenced, those fields will be included inline in the generated `struct`. One side benefit of this approach is that only primitive fields are transferred to the accelerator, reducing data movement relative to approaches that copy the whole JVM object. Future work could further improve the data movement efficiency of this approach by only transferring primitive fields that are referenced from accelerated parallel regions.

Once the primitive fields of a class and its superclasses have been identified using Java reflection, it is straightforward to build a one-to-one mapping from Java primitive types to OpenCL primitive types. For example, reflection would tell us that the `Point` class above has a field named `x` whose type descriptor is "F". The type descriptor "F" can be converted to the OpenCL primitive type `float` using a lookup table.

Using the `packed` attribute for generated `struct`s simplifies object serialization by guaranteeing that all `struct` fields are stored consecutively in memory. Future work could extend these techniques to support representing nested objects as inlined structs, when legal to do so.

3.2 Serializing and Deserializing JVM objects

Runtime serialization and deserialization of dynamically loaded JVM objects is an expensive operation that generally takes O(`N`) time, where `N` is the size of the data to be serialized. Here we describe our strategy for serializing and deserializing JVM objects to the format defined by Section 3.1, which ensures that the resulting data is consumable by an OpenCL accelerator. This work does not focus on optimizing this process. We rely on the techniques presented in Section 3.3 to minimize the number of times expensive serialization and deserialization operations take place.

Data referenced from an accelerated parallel region can be classified into two categories: 1) local variables in the method scope enclosing the lambda creation site, and 2) fields in the class scope enclosing the lambda creation site. Local variables in the enclosing method are captured as fields in an anonymous inner class auto-generated by the RetroLambda pass, including `this`. Fields of the enclosing class are accessible through the `this` reference saved in the RetroLambda anonymous class. Both types of data are accessible from accelerated parallel regions, including fields of the enclosing `this` instance.

Prior to launching a parallel loop on an accelerator, we must iterate over each of these fields that is referenced from inside that parallel loop and decide how to store them on the accelerator. To do so, we start by classifying each into one of four categories: primitive, non-array object, array of primitives, or array of objects.

Supporting primitives on the accelerator is straightforward. Primitives are passed by value to the accelerator kernel using OpenCL's `clSetKernelArg` API. Their values are fetched from the JVM using the JNI APIs.

Singleton objects which are shared across all iterations of a parallel loop may arise in a number of situations, such as the `center` object in the following code example:

```
Point center = new Point(...);
forall(0, P - 1, (iter) -> {
  Point mine = points[iter];
  float distToCenter = mine.distance(center);
  ...
});
```

For this code snippet, the `center` object can be represented by a single `Point struct` allocated on the accelerator. The first step in creating that allocation is to serialize the JVM object to a byte buffer that matches the layout of the OpenCL `struct`. This can be done using the code template below, which takes as input the object to be transferred to the accelerator, a byte buffer to write the serialized object to, and a list of type descriptors and offsets for each field in the object being transferred. This field metadata is sorted in the same order as the fields appear in the OpenCL `struct` definition. The object to be transferred can be loaded by name using Java reflection from either the enclosing class or the RetroLambda-generated closure.

```
void writeObjectToStream(Object ele,
     List<FieldDescriptor> structMemberInfo, ByteBuffer
     bb) {
  for (FieldDescriptor fieldDesc : structMemberInfo) {
    TypeSpec typeDesc = fieldDesc.typ();
    long offsetInClass = fieldDesc.offset();

    switch (typeDesc) {
      case (TypeSpec.I):
        bb.putInt(Unsafe.getInt(ele, offsetInClass));
        break;
      case (TypeSpec.F):
        bb.putFloat(Unsafe.getFloat(ele, offsetInClass));
        break;
      ...
    }
  }
}
```

The above code snippet relies on the `sun.misc.Unsafe` package to fetch both the offset of a field inside an object as well as the value of that field. The Unsafe package is used rather than Java reflection to enable future work on transferring whole JVM objects to accelerators and accessing fields by offsets rather than packing them into a `struct`.

The contents of the generated `ByteBuffer` can then be transferred to the accelerator and used to represent this JVM object. Given that the order of the fields in the `struct` and the order of the fields passed to `writeObjectToStream` are the same, the `packed` attribute used in Section 3.1 ensures that the `ByteBuffer` and OpenCL representations are compatible. The `byte[]` backing the `ByteBuffer` can then be passed through JNI to OpenCL and transferred to the accelerator. Once on the accelerator, this object can be referenced using an appropriately typed pointer, such as:

```
edu_rice_hj_example_Point *ptr;
```

Public fields in the `Point` object can be loaded using the `->` operator:

```
float tmp = ptr->x;
```

Methods of the `Point` class are mangled and include `this` as an explicit parameter. Thus, a call to `mine.distance(center)` would be transformed to:

```
static float
     edu_rice_hj_example_Point__distance(__global Point
     *this, __global Point *other) {
  ...
}

float d = edu_rice_hj_example_Point__distance(mine,
     center);
```

For consistency, all object references on the accelerator are represented as `__global` pointers, pointing to memory in the accelerator's global address space. This design choice will be an important constraint when we discuss serializing arrays of objects.

Transferring arrays of primitives to an accelerator starts with passing the array directly through JNI to a native function. The native code then uses the appropriate JNI API to extract the buffer backing the Java array. For example, for an `int[]` in the JVM the function `GetIntArrayElements` would be used to extract an `int*` pointer to the contents of the JVM array. The values pointed to by the retrieved pointer can then be directly transferred to the OpenCL accelerator.

Transferring arrays of objects to an accelerator re-uses `writeObjectToStream` by iterating over the elements of the array and applying `writeObjectToStream` to each object with the same output `ByteBuffer`. This produces a single byte buffer that stores the contents of each object in the object array, consecutively. However, to keep the representation of object references as `__global` pointers consistent across object singletons and object arrays, it is necessary to allocate an additional array of `__global` pointers on the accelerator which represent the array of object references being serialized on the JVM. The `ByteBuffer` emitted by the successive calls to `writeObjectToStream` stores the data backing those object references. This additional array of pointers has the same length as the object array, and element `i` in the pointer array is pre-populated with the address of element `i` in the data buffer on the accelerator. This preprocessing is done in parallel on the accelerator itself after both buffers have been allocated and the contents of the data buffer have been transferred to the accelerator, but before the main computational kernel is launched.

Null pointers in arrays of objects require special treatment for consistency between the JVM and accelerator. When serializing an array of objects to the accelerator, an extra array of `byte`s is allocated whose length is the same as the array of objects. If element `i` in the array of objects is `null`, element `i` in this `byte` array is set to 1. Otherwise, it is set to 0. This `isNull` array is transferred to the accelerator along with the array of objects and used in a pre-processing kernel to initialize the `__global` pointer array described in the preceding paragraph. If an object reference in the JVM was `null`, the corresponding entry in the pointers array is set to `NULL`. The extra `isNull` array is necessary because the object references themselves are never transferred to or from the accelerator; only the contents of the pointed-to objects. The `isNull` array is necessary to determine whether each element of the corresponding array-of-pointers on the accelerator should be set to `NULL` or to point to an element in the array-of-structs that stores the object contents.

Transferring singleton objects, primitive arrays, and ar-

rays of objects back from the OpenCL accelerator requires performing the same operations as described above, in reverse.

For objects, the serialized object is transferred back from the accelerator and the method `readObjectFromStream` is used to populate the associated JVM object with any changes. Note that HJ-OpenCL does not currently support atomic operations, reduction operations, or `synchronized` regions on shared objects inside accelerated parallel regions. If a field of a singleton object is written from multiple threads, no guarantees are made as to its state when it is restored in the JVM.

```
1  void readObjectFromStream(Object ele,
       List<FieldDescriptor> structMemberInfo, ByteBuffer
       bb) {
2    for (FieldDescriptor fieldDesc : structMemberInfo) {
3      TypeSpec typeDesc = fieldDesc.typ();
4      long offsetInClass = fieldDesc.offset();
5
6      switch (typeDesc) {
7        case (TypeSpec.I):
8          Unsafe.putInt(ele, offsetInClass, bb.getInt)
9          break;
10       case (TypeSpec.F):
11         Unsafe.putFloat(ele, offsetInClass, bb.getFloat)
12         break;
13     }
14   }
15 }
```

Arrays of primitives are restored in the JVM by transferring the accelerator buffer directly into the JVM through a primitive pointer retrieved from JNI.

For arrays of objects, we first launch a post-processing accelerator kernel that iterates across the array of pointers storing the current state of object references on the accelerator. If pointer `i` no longer points to object `i` in the associated data buffer, it must either have been set to `null` or to point at a different object on the accelerator. If the former is true, we mark that object reference as having been nullified in the `isNull` array. Otherwise, the contents of object `i` are updated in the backing array to be the contents of the object instance pointed to by pointer `i` on the accelerator. The object referenced by pointer `i` may change if it now points to an object that was constructed on the accelerator, or to another object of the same type that was transferred from the JVM. More details on how dynamic object allocation is supported are available in Section 3.4.

Once the contents of the array of objects have been updated based on changes in the array of pointers, the contents of the array of objects are transferred out of the accelerator and used to update the contents of JVM objects by iterating over the JVM array and updating each object using `readObjectFromStream`. If an object reference is marked as null in `isNull`, the corresponding object reference in the JVM is also set to `null`.

This work assumes that no two elements in an array of object references transferred to the accelerator point to the same object. We discuss how this restriction could be lifted in Section 5.

Clearly, the serialization and deserialization of JVM objects and particularly arrays of JVM objects is an expensive operation. If `M` is the number of fields in a given type, serializing or deserializing an object of that type is `O(M)`. For an object array of length `N` that cost increases to `O(NM)`. Past work[7] has looked at using GPU parallelism and memory bandwidth to accelerate re-formatting the data layout of data structures. Our work could be extended to use similar techniques to accelerate data serialization.

The next section describes the bytecode analysis techniques used to eliminate redundant data movement between successive accelerated parallel regions. This optimization reduces the serialization and deserialization overheads incurred from supporting object references in accelerated parallel regions.

3.3 Removing Redundant Data Movement Through Context Inspection

APARAPI supports basic load/store optimizations by analyzing the bodies of parallel regions for read-only and write-only buffers. If a buffer is detected as read-only, it is only transferred from the host JVM to the accelerator. If it is write-only, it is only transferred from the accelerator to the JVM.

In this work, we extend this capability by loading and inspecting the bytecode of the method in which this parallel region is launched and finding local variables or class fields that are passed to multiple, successive, accelerated parallel regions without being referenced from the JVM between those parallel regions. At a high level, a buffer (object, primitive array, or object array) will only be transferred back to the JVM after an accelerated parallel region if it may be read before the JVM launches another accelerated parallel region that is passed the same buffer. A buffer is only transferred to the accelerator prior to an accelerated parallel region if we find that this buffer may have been written from the JVM since the last accelerated parallel region that referenced it.

This analysis is currently intra-procedural and context-insensitive. As a result, when looking for reads following a parallel region the analysis will transfer a buffer back if there exists any control flow path from that parallel region to a method return that does not pass through a parallel for region that uses this buffer. Likewise, when looking for writes preceding a parallel region the analysis will transfer a buffer to the accelerator if there exists any control flow path from the start of the enclosing method to this parallel region that does not pass through any other parallel regions that use this buffer.

This analysis is particularly useful for applications that exhibit a pipeline of parallel regions. For example, the KMeans machine learning algorithm consists of a pipeline of two kernels: one kernel that classifies each data point into a cluster, and a second kernel that recalculates each cluster's centroid based on its member points. In this pipeline, we can safely skip both retrieving the point classifications after the first kernel and copying them back to the accelerator for the second kernel. Our analysis captures this information as long as 1) both parallel regions are launched in the same function, and 2) the analysis described below indicates the point classifications are not read or written from the JVM between the two parallel regions.

For each buffer used by any accelerated parallel region, the location in the source code of the last accelerated parallel region that referenced that buffer is stored by the HJ-OpenCL runtime. This last referenced location is updated following the completion of each parallel region for all buffers referenced. To determine if a given buffer must be transferred

to the accelerator prior to launching a parallel region, we first check that the last referenced location is in the same method as the current accelerated parallel region. If it is not, then the transfer must be performed. Otherwise, we start from the last referenced location and traverse forward over bytecode instructions to verify that the preceding parallel region is post-dominated by the current parallel region and that the state of the buffer could not have been modified in the JVM. If any of the following conditions are met, our analysis indicates that the current buffer may have been modified and, therefore, must be updated on the accelerator before the parallel region can be launched:

1. If the current buffer is stored in a local variable slot and we find a local variable store opcode (e.g. `ASTORE_3`) to that slot.
2. If we encounter a return statement. Note that even though the last referenced location for a buffer is within the same method it may have been during a different call to this method. Checking for a return is necessary to check for this case.
3. If we encounter an `AASTORE` opcode.
4. If we encounter a `PUTFIELD` opcode.
5. If we encounter any type of method invocation.
6. If we encounter any opcode that may throw an exception (e.g. a divide-by-zero exception thrown by an `IDIV` instruction)

When deciding whether a buffer must be transferred back from the accelerator following an accelerated parallel region, we perform a similar traversal starting at the current parallel region and verify that the current parallel region is post-dominated by parallel regions that use the same buffer and that no reads of this buffer may be performed from the JVM between parallel regions. If any of the following conditions is met by any of the bytecode instructions following this parallel region and before encountering another parallel region that uses this buffer, the current buffer is transferred back to the JVM:

1. If the current buffer is stored in a local variable slot and we find a local variable load opcode (e.g. `ALOAD_3`) for that slot.
2. If we encounter a return statement.
3. If we encounter an `AALOAD` opcode.
4. If we encounter a `GETFIELD` opcode.
5. If we encounter any type of method invocation.
6. If we encounter any opcode that may throw an exception (e.g. a divide-by-zero exception thrown by an `IDIV` instruction)

These transfer decisions are cached for each `forall_acc` region. This allows us to only perform this analysis once for each buffer in each parallel region regardless of how many times that region is entered, reducing overhead.

One special case is that of an accelerated parallel region that does not have any buffers that need to be transferred back to the JVM. That is, all buffers used by this kernel are also referenced by other accelerated parallel regions that post-dominate the current parallel region, and none of those buffers is referenced from the JVM. If this is the case, we can safely allow the non-producing accelerated parallel region to execute asynchronously on the accelerator while the host system continues JVM execution. All accelerated parallel regions wait for any preceding kernels before starting their computation.

Buffers are allocated the first time they need to be transferred to the accelerator and are freed after they are copied back to the host following the last accelerated parallel region to reference them. Hence, a buffer may be allocated on the accelerator even though a currently active parallel region is not working on it if the active parallel region is between two regions that do. This may lead to out-of-memory errors even when the working set for a single parallel region is within the limits of accelerator memory. Supporting out-of-core data is beyond the scope of this work.

Future work could extend this analysis to be inter-procedural and control flow sensitive, which would likely improve the accuracy of the redundant data movement elimination.

3.4 Dynamic Memory Allocation in Auto-Generated OpenCL Kernels

While supporting JVM object references on accelerators helps to expand the domain of applications that can be transparently offloaded, object references have limited usefulness without the ability to dynamically allocate new objects. However, the OpenCL standard does not include dynamic memory allocation as a supported operation and most accelerators have little or no native support for it. In this section, we describe a technique for supporting dynamic memory allocation on accelerators by transparently replacing the `NEW` opcode and object constructors during code generation.

On the accelerator, there are four global data structures used to support dynamic memory allocation in our approach:

- `heap`: A large, shared, byte-array pre-allocated in the OpenCL `__global` address space, where it is accessible to all threads.
- `top`: A single, shared atomic integer initialized to zero. When a thread performs a dynamic memory allocation, it atomically increments this integer by the number of bytes it is allocating. If the new value of the integer is less than or equal to the number of bytes in the `__global` heap, the allocation has succeeded. If the old value of the atomic integer were stored in `oldValue`, the current thread's allocation is now available at `(char *)heap + oldValue` and is followed by a contiguous chunk of the requested number of bytes.
- `complete`: An array of integers whose length is equal to the number of iterations in the current parallel loop. Each element in this array is initialized to zero at the start of processing a parallel region. Element `i` of this array is set to one when iteration `i` of the parallel for loop completes successfully on the accelerator. An iteration can only fail if a dynamic memory allocation cannot be satisfied by the heap.
- `anyFailed`: A single integer on the accelerator that is

initialized to zero and set to one by any thread which fails a dynamic memory allocation.

Using these data structures (`heap`, `top`, `complete`, `anyFailed`), the dynamic memory allocation technique works as follows from the host application:

1. Buffers for the accelerator data structures described above are allocated. `complete` is zeroed.
2. The host application zeroes `top` and `anyFailed` and launches the generated kernel.
3. When the kernel completes, `anyFailed` is transferred back to the host. If it is non-zero, we return to step 2 and re-execute with a reset heap. However, iterations of the parallel loop that have marked themselves as completed in the `complete` buffer skip re-execution. Otherwise, if `anyFailed` is zero we continue.

This technique supports execution of parallel regions on accelerators whose dynamic memory allocation requirements exceed the size of the allocated heap by repeatedly retrying subsets of iterations in the same accelerated parallel region until all iterations succeed. This requires an extra allocation for any output object-typed arrays on the accelerator to persist the final object stored by a completed thread before resetting the heap. To support dynamic memory allocation, some extensions to APARAPI's code generation are also required.

First, all methods that perform dynamic memory allocation (i.e., use the `NEW` opcode) are identified, along with all callers of these methods. A field named `allocFailed` is added to the `This struct` that each thread has a thread-private copy of. This field is initialized to zero before performing the work for each iteration of the original parallel for loop. If an allocation fails, `allocFailed` is set to one and the memory allocation function returns. All calls to the memory allocation function as well as any calls that may lead to the memory allocation function being called are followed by a check that `allocFailed` is still zero. If `allocFailed` is found to be non-zero, the current function immediately returns as will all others on the current call stack. For functions with a non-void return value, a default value is returned based on its return type (e.g. `0` for `int`). Below is a code snippet demonstrating the structure of the generated functions:

```
__global void *alloc(..., int *allocFailed) {
  __global void *allocation = ...;
  if (allocation == NULL) {
    *allocFailed = 1;
  } else {
    *allocFailed = 0;
  }
  ...
}

void foo(This *this, ...) {
  __global void *ptr = alloc(..., &this->allocFailed);
  if (this->allocFailed) return;
  ...
}

void bar(This *this, ...) {
  foo(this, ...);
  if (this->allocFailed) return;
  ...
}

__kernel void run(...) {
  This this;
  ...
  for (int i = tid; i < nthreads; i++) {
    ...
    bar(&this, ...);
    if (this.allocFailed) {
      complete[i] = 0;
      *anyFailed = 1;
    } else {
      complete[i] = 1;
    }
  }
  ...
}
```

This generated code simulates an exception-like mechanism on the accelerator, albeit only for out-of-memory situations. At the top-level, if `allocFailed` is found to be non-zero, the element in `complete` corresponding to this iteration is not set and `anyFailed` is set to one. When the host discovers a non-zero value in `anyFailed`, this kernel will be rerun but any iterations with a non-zero value in `complete` will be skipped.

This technique does not make any guarantees on completion. With a small heap or large allocations, it is theoretically possible for this approach never to converge to completion. Future work could address this by reducing the level of parallelism at each re-execution if no progress was made. Halving the number of threads executing the parallel region would increase the chance of individual parallel iterations succeeding as contention for the heap decreases. In the worst case, this would devolve to a single-threaded parallel region and guarantee completion (but not efficiency) as long as the allocated heap was sufficiently large to support the dynamic allocations of each individual thread. If that were not the case, this condition would be easily detectable and handled by reverting to JVM execution.

This technique also assumes that the body of a single iteration of the enclosing `forall_acc` is idempotent: it does not modify its own input state. If a thread modified some input state, failed an allocation, and then was retried in a later kernel invocation, it would read partially updated state on the accelerator. This requirement is not currently enforced during code generation, but could be in future work. While this is a limitation, this assumption is implicitly true in the functional style programming that is common in parallel programming frameworks (e.g., Scala parallel collections) and in current JVM acceleration research[11].

4. EXPERIMENTAL EVALUATION

In this section, we will start by summarizing the experimental setup and applications used to benchmark the HJ-OpenCL system. We will then begin the performance evaluation by comparing HJlib and HJ-OpenCL performance at the granularity of parallel regions, without any redundant transfer elimintation. HJ-OpenCL will use a GPU accelerator. After enabling redundant transfer elimination we rerun all parallel regions and then note and explain any change in performance. Using this information, we statically select whether to run each parallel region as a `forall` (for JVM execution) or a `forall_acc` (for native CPU or GPU execution) and measure overall speedup of HJ-OpenCL using

native threads on the CPU or GPU, comparing against parallel Java Streams. Finally, we measure how performance of one benchmark degrades as the size of the accelerator heap is artificially reduced.

4.1 Experimental Setup

All benchmarks in this section are run on the same hardware platform, containing a 12-core 2.80GHz Intel X5660 CPU, 48GB of system RAM, and 2 discrete NVIDIA M2050 GPUs each with 2.5GB of global memory. All tests are run using all 12 CPU cores but only 1 GPU, and with the maximum heap size of the JVM set to 48GB. All tests are repeated 30 times inside the same JVM to minimize the impact of random variations and to allow JIT compilation to improve JVM performance. The median execution time is reported. All tests are also run across a range of inputs to quantify where performance is lost or gained using accelerated parallel regions. These experiments use the Hotspot JVM v1.8.0_45 and the NVIDIA OpenCL 1.1 implementation included with CUDA 6.0.1.

Three benchmarks were used to evaluate HJ-OpenCL: KMeans, PageRank, and NBody.

KMeans iteratively finds `K` clusters in an input dataset of `P` points. Each iteration of KMeans is a two-stage pipeline: the first stage classifies each point into a cluster based on a Euclidean distance measure from each cluster's centroid to that point's location. The second stage computes new cluster centroids based on the point memberships calculated in the first stage. This pipeline is executed for `I` iterations or until the cluster centroids converge to a steady state. In our experiments we keep `I` as a constant, 10, but vary the number of clusters and data points. The first stage is parallelized across data points and the second stage is parallelized across clusters. The second stage allocates a new `Point` object for each cluster to store the re-calculated cluster centroid.

NBody simulates particle-particle force interactions and the resulting changes in particle positions. This implementation of NBody is precise, and does not use grid approximations to reduce the ratio of computation to communication. Each time step of an NBody simulation includes a two-stage pipeline: the first stage updates each particle's acceleration and velocity based on the position and mass of all other particles. The second stage updates each particle's position based on its re-computed velocity. In this evaluation we execute 20 timesteps and vary the number of points. Both stages of NBody are parallelized across particles.

PageRank is also an iterative application with a two-stage pipeline which assigns a rank to each node in a directed graph based on the ranks of its neighbors with inbound edges. The first stage in PageRank's pipeline assigns a weight to each edge in the graph based on the source node's rank and its number of outbound edges. The second stage uses edge weights to re-compute each nodes' rank based on the weights assigned to its inbound edges. We keep the number of iterations as a constant, 10, but vary the number of nodes and edges. The first stage of PageRank is parallelized across edges in the graph and the second stage is parallelized across nodes.

Table 1 details the characteristics of each benchmark as they relate to evaluating the contributions of this work. The first column indicates if any of the parallel regions in the benchmark contain object references. The second column indicates if this benchmark contains transfers to or from the accelerator which our transfer elimination algorithm identifies as redundant. The third column indicates if any dynamic memory allocations are performed in parallel regions of this benchmark.

Benchmark	Obj Refs	Copy Elim	Dyn Alloc
KMeans	Y	Y	Y
NBody	Y	Y	N
PageRank	Y	Y	N

Table 1: Characteristics of each benchmark as they relate to the contributions of this work.

4.2 Kernel Performance With Redundant Transfers

Initially, we compare performance of every parallel region running as a `forall` loop and as an accelerated `forall_acc` loop on a GPU with no redundant transfer elimination. All input and output singleton objects, primitive arrays, and object arrays are transferred to and from the accelerator in these experiments. The results for each benchmark are listed in Figures 1, 2, and 3. For each dataset and kernel, either the `forall` or `forall_acc` results are shaded gray to visually indicate the higher performing execution mode. In many cases, smaller datasets perform better when running on the JVM with the HJlib runtime, and larger datasets perform better when an accelerator is used (e.g. `updateClusters` in KMeans and `updateRanks` in PageRank).

These results show that even with redundant transfers, sufficiently large datasets which produce enough parallelism benefit from the accelerated parallel regions implemented in this work. Note that for improved performance to be achieved, the computational acceleration must be sufficient to offset both increased costs from transfers over the PCIe bus as well as costs from data serialization and deserialization.

Figure 1 shows that KMeans acceleration is primarily a function of the number of clusters (`K`) being calculated. `K` serves as a multiplier of the amount of work performed for each data point. Because communication scales by `O(P + K)`, computation scales by `O(PK)`, and `P` is generally much larger than `K`, increasing `K` increases the chances that the accelerator will have a measurably positive impact on overall parallel region execution time.

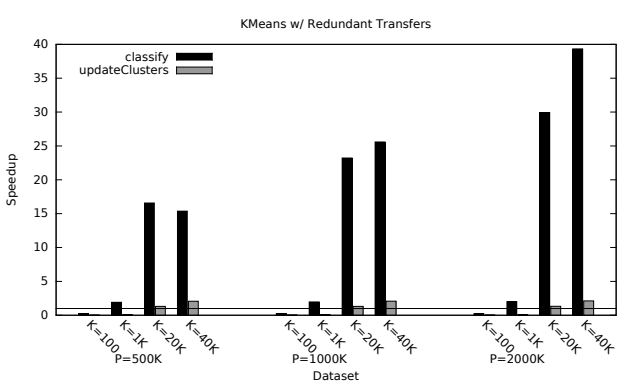

Figure 1: Kernel speedup with redundant copies in the KMeans benchmark.

Figure 2 shows that while accelerated parallel regions in NBody have an impact at larger datasets for `updateVel`, `updatePos` always runs faster on the JVM with the HJlib parallel runtime. The `updatePos` kernel includes a trivial amount of work but requires transferring and serializing JVM objects to and from the accelerator. Hence, overheads dominate accelerated execution of `updatePos` on the accelerator.

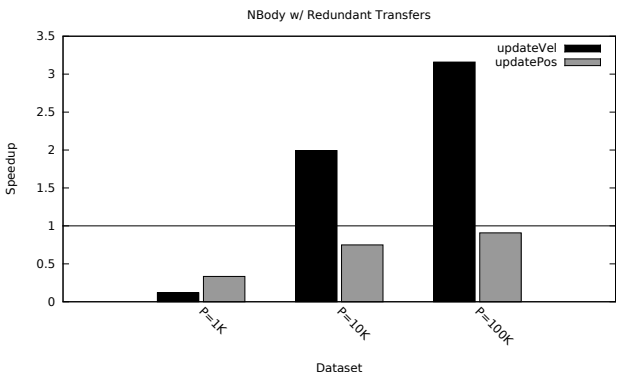

Figure 2: Kernel speedup with redundant copies in the NBody benchmark.

As we would expect, Figure 3 shows that for small node counts the `updateRanks` kernel in PageRank (which is parallelized across nodes) does not perform well on the accelerator. Like the kernels of KMeans and `updatePos` in PageRank, `updateRanks` is dominated by transfer and serialization overheads at smaller node counts with little computational work to accelerate. However, the `calcWeights` kernel always performs better on the accelerator than the JVM (for the data sizes that we studied).

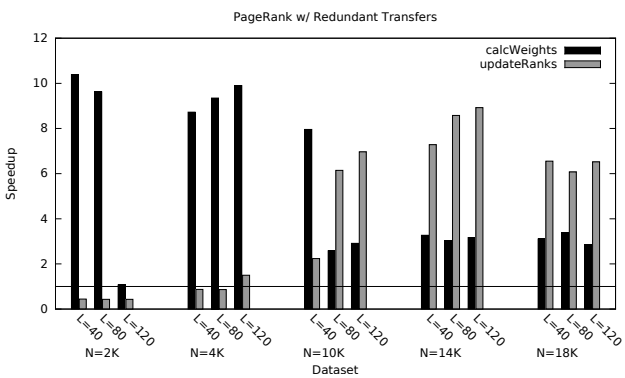

Figure 3: Kernel speedup with redundant copies in the PageRank benchmark.

4.3 Kernel Performance Without Redundant Transfers

Building on the results in Section 4.2, we enable the redundant transfer elimination described in Section 3.3 and rerun all experiments. For these experiments, we explicitly force all accelerated parallel regions to block on accelerator computation before returning to the host program. This simplifies the performance comparison in this section for kernels which have no buffers that must be transferred back to the JVM on completion and would therefore execute asynchronously.

Eliminating redundant transfers in KMeans does add some performance benefit. A summary of the results before enabling this optimization (copy-all) and after (elim) is provided in Figure 4. Redundant transfer elimination saves transferring 50% of data by eliminating all transfers from the accelerator following `classify` and to the accelerator before `updateClusters`. We can see that for every dataset this improves execution time. However, for kernels and datasets where JVM execution was faster than accelerated execution with redundant transfers (Figure 1), the performance improvement from redundant transfer elimination is insufficient to make those execution configurations now faster on the accelerator.

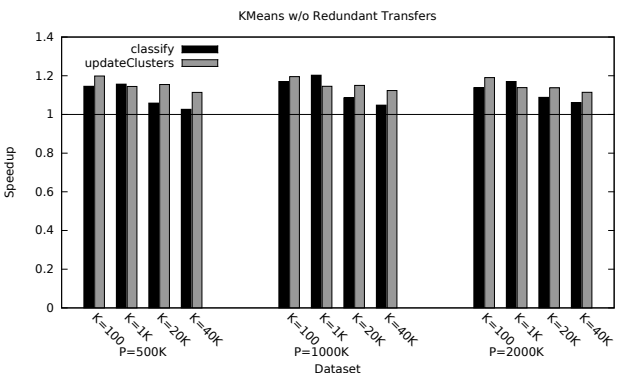

Figure 4: Kernel speedup with redundant copy elimination in the KMeans benchmark.

We find that for the NBody `updateVel` and `updatePos` kernels there is no improvement in performance. While our redundant transfer elimination algorithm keeps the velocity and position buffers on the accelerator between these two kernels, our NBody implementation is heavily computation-bound and so eliminating these transfers does not significantly improve the performance of the overall parallel region. For simulations of 100,000 particles this elimination reduces bytes transferred to and from the device by 58.8%.

The PageRank results with redundant transfers eliminated are summarized in Figure 5. Here, eliminating redundant transfers provides a clear performance benefit across all datasets. In every case, the optimized version performs better than the naive, copy-everything version. Redundant transfer elimination reduces the number of bytes transferred to and from the accelerator by 50%.

While the results for PageRank are similar to KMeans in that no kernel and dataset execution configuration which performed better on the JVM in Figure 3 now performs better on the accelerator, it is clear the inflection point at which the accelerator is the better choice has moved down as a function of `L`. For example, consider the kernel `updateRanks` for tests with `N=4K`. Let us fit a quadratic function `f(L) = E` to this data where `E` is the expected execution time for `updateRanks`. Without redundant transfer eliminiation, the fitted equations for `forall` and `forall_acc` predict an inflection point at `L=91`. With redundant transfer elimination, that inflection point reduces to `L=84`, increasing the number of datasets that execute faster on the accelerator.

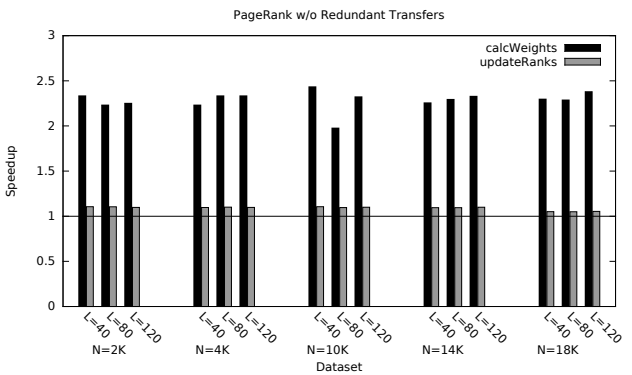

Figure 5: Kernel speedup with redundant copy elimination in the PageRank benchmark.

4.4 Overall Speedup

Using the insights gained in Sections 4.2 and 4.3, this section measures the overall speedup of whole benchmarks relative to a parallel implementation using Java Streams. We choose to compare performance against Java Streams because it is an industry standard with similar parallel functionality to HJlib's `forall` construct. Table 2 lists the kernels we choose to offload to the accelerator; `updatePos` in NBody is the only kernel that was not offloaded. Table 3 lists the overall speedup achieved on each benchmark and each dataset. All speedups are normalized to the execution time of an implementation that uses parallel Java Streams. We compare performance of HJlib's `forall` running in the JVM, `forall_acc` using OpenCL to run on the GPU, and `forall_acc` using OpenCL to run on the CPU.

As expected from the results in Sections 4.2 and 4.3, the GPU-accelerated version of the application generally performs better than Java Streams, `forall`, and the CPU-accelerated version as the datasize and parallelism of the input dataset increases.

The `forall` and Java Streams versions of each benchmark generally outperforms the GPU-accelerated version on smaller datasets where there is insufficient work for the GPU's parallelism to offset the overheads of data serialization and transfer.

The CPU-accelerated version (i.e., the version in which OpenCL code is executed on the CPU) of each benchmark offers an interesting tradeoff compared to the other execution platforms. Like `forall` and Java Streams, it executes on the CPU and therefore handles irregular computation and memory accesses better than the GPU. The CPU also handles non-coalesced memory accesses better than the GPU, a common access pattern when referencing arrays-of-structs rather than structs-of-arrays. Due to the serialization techniques described in Sections 3.1 and 3.2, all of these benchmarks operate on arrays-of-structs. However, when using the CPU as an accelerator we incur the same data serialization overhead that we do on the GPU, but the data transfer overhead is lower because it does not go over the PCIe bus. These results show that for some benchmarks there is a middle ground where the size of the dataset is sufficiently large for native CPU execution to demonstrate a performance benefit over JVM execution (`forall` or Java Streams) despite serialization overheads, but still small enough that the acceleration from GPU execution is insufficient to offset the transfer overheads. In particular, note the results for KMeans when `K=1K` or the PageRank dataset `N=4K, L=120`.

Benchmark	Kernel	Accelerator?
KMeans	classify	Y
	updateClusters	Y
NBody	updateVel	Y
	updatePos	N
PageRank	calcWeights	Y
	updateRanks	Y

Table 2: Kernels selected for acceleration.

Automatic runtime identification of the best performing configuration (JVM, native OpenCL code on CPU, native OpenCL code on GPU) for a given kernel is a subject for future work.

4.5 Performance Degradation as Heap Contention Increases

One of the contributions of this paper is support for dynamic memory allocation on OpenCL accelerators. The techniques described in Section 3.4 enable acceleration of JVM applications where the dynamic memory allocations exceed the size of the heap that is allocatable on the accelerator. Because these techniques are based on retrying threads that fail to complete successfully, launching multiple kernels per parallel region becomes necessary, but naturally introduces overhead. In this section, we study how overhead and overall execution time of the KMeans benchmark increases as we artificially constrain the heap size to force allocation failures.

In KMeans, a new object is allocated on each iteration for each cluster that stores the new coordinates of that cluster. In these experiments, we test against the dataset with the largest `K`. For `K=40,000`, KMeans dynamically allocates 12 bytes per cluster (480KB in total).

Tables 4 and 5 show the number of kernel retries and total execution time for KMeans running on the GPU and CPU, as a function of the heap size. As we halve the heap size, the number of retries necessary approximately doubles. Overall execution time increases sub-linearly because each successive kernel retry in the same parallel region contains less work than the preceding one as more parallel iterations complete successfully. The execution time of Java Streams on the same KMeans dataset was 70,091 ms. Even with an artificially small heap size, HJ-OpenCL running on the GPU and CPU is able to maintain a performance advantage over Java Streams. The performance of CPU-accelerated HJ-OpenCL degrades at a slightly slower rate because the latency to transfer `anyFailed` from the accelerator to the JVM to check for failed allocations after every kernel launch is lower.

5. LIMITATIONS

5.1 JVM Object Support on Accelerators

In this paper, we provide a detailed description of our approach to supporting JVM object references in accelerated parallel regions. While allowing some object references increases the scope of JVM acceleration, limitations remain on the type of objects that can be serialized to OpenCL ac-

	Dataset	HJlib	GPU	CPU
KMeans	P=500K, K=100	0.97×	0.09×	0.19×
	P=500K, K=1K	1.01×	0.61×	1.31×
	P=500K, K=20K	1.23×	10.26×	6.92×
	P=500K, K=40K	1.21×	11.94×	5.81×
	P=1000K, K=100	1.05×	0.10×	0.22×
	P=1000K, K=1K	1.12×	0.71×	1.63×
	P=1000K, K=20K	1.06×	10.36×	6.93×
	P=1000K, K=40K	1.23×	15.78×	7.51×
	P=2000K, K=100	1.05×	0.01×	0.21×
	P=2000K, K=1K	1.22×	0.71×	1.63×
	P=2000K, K=20K	1.10×	11.14×	7.42×
	P=2000K, K=40K	1.23×	18.33×	8.74×
NBody	P=1K	0.50×	0.08×	0.07×
	P=10K	1.02×	0.89×	0.61×
	P=100K	0.89×	1.23×	0.72×
PageRank	N=2K, L=40	1.00×	0.45×	0.74×
	N=2K, L=80	1.04×	0.45×	0.81×
	N=2K, L=120	1.03×	0.45×	0.80×
	N=4K, L=40	1.03×	0.89×	1.05×
	N=4K, L=80	1.05×	0.90×	1.05×
	N=4K, L=120	0.93×	1.53×	1.83×
	N=10K, L=40	1.03×	2.43×	1.41×
	N=10K, L=80	0.98×	5.95×	2.61×
	N=10K, L=120	1.02×	6.45×	1.85×
	N=14K, L=40	0.96×	7.05×	3.11×
	N=14K, L=80	0.98×	8.60×	1.69×
	N=14K, L=120	0.98×	9.03×	1.78×
	N=18K, L=40	0.97×	5.88×	3.06×
	N=18K, L=80	0.95×	6.57×	1.76×
	N=18K, L=120	1.04×	6.68×	1.66×

Table 3: Speedup of overall execution time for all benchmarks, relative to parallel Java Streams. This table compares HJlib's `forall` to HJ-OpenCL's `forall_acc` using the GPU or CPU as accelerators. The fastest performing platform for each test is grayed in.

Heap Size	Retries	GPU	
		Time	Slowdown
800KB	1	6,202 ms	
400KB	2	6,965 ms	1.12×
200KB	3	8,434 ms	1.36×
100KB	5	13,079 ms	2.11×
50KB	10	22,682 ms	3.66×

Table 4: Performance degradation of the KMeans benchmark as the HJ-OpenCL heap size is reduced on the GPU, tested with 500,000 data points and 40,000 clusters.

Heap Size	Retries	CPU	
		Time	Slowdown
800KB	1	12,323 ms	
400KB	2	13,210 ms	1.07×
200KB	3	16,492 ms	1.34×
100KB	5	23,317 ms	1.89×
50KB	10	37,557 ms	3.05×

Table 5: Performance degradation of the KMeans benchmark as the HJ-OpenCL heap size is reduced on the CPU, tested with 500,000 data points and 40,000 clusters.

celerators. Most notably, the only object fields that can be accessed on the accelerator are those with primitive types. This is not an absolute limitation: JaBEE[25] is an example of existing work that supports nested object references. However, JaBEE's results demonstrate that adding this support cancelled out the performance advantage of accelerator execution. Not only does indirection complicate the serialization logic, but multiple layers of indirection is not generally a pattern that fits well with simpler accelerator architectures. We chose to focus on kernels that were most likely to be amenable to acceleration, kernels that operate on primitive or simple object types. Future work could investigate alternatives to JaBEE's approach to indirection that perform better on accelerators.

This brings up a tension between how much of the JVM bytecode specification can be supported on accelerators and how much should be supported. The choice to move to accelerators is usually made primarily for performance, and secondly for energy efficiency. We believe the goal of research like ours should be to investigate how certain portions of the JVM specification are supportable on the accelerator, what compromises must be made on performance and/or energy efficiency, and how those compromises change with application or dataset characteristics. JaBEE [25] did this for indirect object references. Previous work in HJ-OpenCL did this for exceptions [14] and global synchronization [13]. This work does the same for dynamic memory allocation. Through experimentation the community can converge on a reasonable subset of the JVM bytecode specification that can be supported on accelerators without sacrificing performance/energy efficiency and retaining enough JVM semantics to remain useful and intuitive for JVM programmers.

For example, a possible future investigation would remove the requirement that no two object references in an array of object references transferred to the accelerator be aliased. This investigation would weigh the benefits of bringing the semantics of a `forall_acc` loop in line with regular JVM semantics against the added performance cost of checking all object references in an array against all other object references at runtime.

5.2 Redundant Transfer Elimination Limitations

The main limitation to the redundant transfer elimination techniques described in this paper is the simplicity of the change detection between accelerated parallel regions. Our work focuses on pipelines of accelerated parallel regions where the code between one region and the next is primarily setting up loop bounds, loading local variables, etc. For that

type of parallel application, our implementation of redundant transfer elimination works well. Integrating a modern alias analysis algorithm for JVM bytecode such as the one in [19] or integrating this work with the JVM itself would remove these limitations but is beyond the scope of this work.

6. CONCLUSIONS

In this paper, we presented techniques towards bridging the gap between the JVM and modern accelerators. While the JVM offers portability and high-level programming abstractions, it is not generally considered useful for performance-critical applications that leverage accelerators.

However, OpenCL offers a portable programming platform for many accelerator architectures which are beyond the reach of the JVM. In this work, we transparently accelerate the existing `forall` parallel-for loop construct from HJlib using OpenCL accelerators, demonstrating significant performance improvements with minimal code change for existing HJlib or Java Streams applications. More notably, we increase the JVM features supported on OpenCL accelerators by allowing object references and dynamic memory allocation in accelerated parallel regions. We also present an algorithm for redundant transfer elimination based on JVM bytecode inspection for dataflow dependencies between accelerated parallel regions. Our approach can easily be extended to support the Java Streams API as input instead of HJlib `forall` loops.

There are many possible directions for future work to enable more general JVM computation on accelerators. The object support in this paper could be extended to support a wider range of object definitions, including nested object references. The dynamic memory allocation techniques would have to be similarly extended, and could be made more efficient by using OpenCL local memory to satisfy allocations, thereby reducing heap contention and access latency. There are also opportunities to adapt garbage collectors and memory management systems to be better optimized for accelerator-based execution. The redundant transfer elimination algorithm could be improved by using techniques in JVM alias analysis and control flow analysis to make the analysis more accurate. For example, because the current approch is not context sensitive and because the tested applications generally consist of an iterative pipeline of parallel regions, no transfers to the accelerator can be eliminated for the first kernel in the pipeline and no transfers from the accelerator can be eliminated for the last kernel in the pipeline. The algorithm has no way of determining if we are entering the first kernel as part of the first iteration of the enclosing loop, or if we will exit the enclosing loop after leaving the last kernel. This leads to unnecessary transfer and serialization overheads in the current approach.

Additionally, simple automated data layout transformations for accelerator kernels would likely be effective in improving the efficiency of individual kernels[18]. Past work[7] has used the parallelism and memory bandwidth of GPUs to accelerate data layout transformations. Our work could be extended to include the techniques from both [18] and [7] to perform efficient data layout transformations at runtime. Along the same vein of automated optimization, identifying data that follows certain access patterns would us to make more use of special-purpose OpenCL memory regions such as `__constant`, texture, or `__local` memory.

However, despite these limitations in our current approach, the work reported in this paper significantly broadens the class of JVM applications that can be offloaded to accelerators.

Implementing the techniques described in this paper within the JVM itself would allow for an increase in efficiency. Representing JVM objects on accelerators would require matching the padding of an auto-generated native struct with that of the JVM object's fields, skipping the need for a serialization or deserialization step but consuming more space on the accelerator. Access to the full state of the JVM would simplify optimizing redundant transfer elimination. The dynamic allocation techniques presented here could be re-used.

Additionally, ongoing work uses the techniques described in this paper to accelerate distributed Apache Spark applications. Spark's extensive use of lambdas and parallel collections as an abstraction for distributed execution means that many of the ideas in this paper apply directly to accelerating Spark applications.

As a wider range of applications with varied performance characteristics come under the umbrella of JVM execution, it is important to give application developers more flexibility in their hardware platform to guarantee applications can meet new performance requirements. This work makes progress towards that goal by building on past work and extending the subset of JVM applications that can be transparently and efficiently executed on OpenCL accelerators.

7. REFERENCES

[1] Apache Tomcat. http://tomcat.apache.org/. Accessed: 2015-06-05.

[2] Everything I Ever Learned About JVM Performance Tuning. http://bit.ly/QOYhg6. Accessed: 2015-06-04.

[3] RetroLambda. https://github.com/orfjackal/retrolambda. Accessed: 2015-06-05.

[4] G. K. V. S. Akihiro Hayashi, Kazuaki Ishizaki. Machine-Learning-based Performance Heuristics for Runtime CPU/GPU Selection. In *12th International Conference on the Principles and Practice of Programming on the Java Platform, PPPJ*, 2015.

[5] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-java: the new adventures of old x10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, pages 51–61. ACM, 2011.

[6] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *Acm Sigplan Notices*, 40(10):519–538, 2005.

[7] S. Che, J. W. Sheaffer, and K. Skadron. Dymaxion: optimizing memory access patterns for heterogeneous systems. In *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, page 13. ACM, 2011.

[8] R. Coleman, U. Ghattamaneni, M. Logan, and A. Labouseur. Computational Finance with Map-Reduce in Scala. In *Conference on Parallel and Distributed Processing (PDPTAâĂŹ12), CSREA*, 2012.

[9] Eric Caspole. AMD's Prototype HSAIL-enabled JDK8 for the OpenJDK Sumatra Project.

http://www.oracle.com/technetwork/java/jvmls2013caspole-2013527.pdf, 2013.

[10] G. Frost. APARAPI in AMD Developer Website.

[11] J. J. Fumero, M. Steuwer, and C. Dubach. A composable array function interface for heterogeneous computing in java. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, page 44. ACM, 2014.

[12] M. Grossman, M. Breternitz, and V. Sarkar. Hadoopcl2: Motivating the design of a distributed, heterogeneous programming system with machine-learning applications. In *IEEE Transactions on Parallel and Distributed Systems*.

[13] A. Hayashi, M. Grossman, J. Zhao, J. Shirako, and V. Sarkar. Accelerating Habanero-Java programs with OpenCL generation. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 124–134. ACM, 2013.

[14] A. Hayashi, M. Grossman, J. Zhao, J. Shirako, and V. Sarkar. Speculative execution of parallel programs with precise exception semantics on gpus. In *Languages and Compilers for Parallel Computing*, pages 342–356. Springer, 2014.

[15] S. Imam and V. Sarkar. Cooperative Scheduling of Parallel Tasks with General Synchronization Patterns. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 618–643. Springer, 2014.

[16] S. Imam and V. Sarkar. Habanero-Java Library: A Java 8 Framework for Multicore Programming. In *11th International Conference on the Principles and Practice of Programming on the Java Platform, PPPJ*, volume 14, 2014.

[17] S. Imam and V. Sarkar. The Eureka Programming Model for Speculative Task Parallelism. In *European Conference on Object-Oriented Programming (ECOOP)*, 2015.

[18] D. Majeti, R. Barik, J. Zhao, M. Grossman, and V. Sarkar. Compiler-driven data layout transformation for heterogeneous platforms. In *Euro-Par 2013: Parallel Processing Workshops*, pages 188–197. Springer, 2014.

[19] D. Nikolic and F. Spoto. Definite expression aliasing analysis for java bytecode. In *Theoretical Aspects of Computing–ICTAC 2012*, pages 74–89. Springer, 2012.

[20] OpenJDK. Project Sumatra. http://openjdk.java.net/projects/sumatra/.

[21] 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. IEEE, 2012.

[22] G. L. Taboada, S. Ramos, R. R. Expósito, J. Touriño, and R. Doallo. Java in the High Performance Computing arena: Research, practice and experience. *Science of Computer Programming*, 78(5):425–444, 2013.

[23] W. VanderHeyden, E. D. Dendy, and N. Padial-Collins. CartaBlancaâĂŤa pure-Java, component-based systems simulation tool for coupled nonlinear physics on unstructured gridsâĂŤan update. *Concurrency and Computation: Practice and Experience*, 15(3-5):431–458, 2003.

[24] Vivek Sarkar. COMP 322: Introduction to Parallel Programming. https://wiki.rice.edu/confluence/display/PARPROG/COMP322.

[25] W. Zaremba, Y. Lin, and V. Grover. Jabee: framework for object-oriented java bytecode compilation and execution on graphics processor units. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, pages 74–83. ACM, 2012.