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

Max Grossman, Shams Imam, Vivek Sarkar
Habanero Extreme Scale Software Research Group
Rice University
JVM: A Portable Abstraction

JVM: platform-agnostic execution model

- Primary customers: language, tool developers
- Allows JVM users to focus on loftier ideas
- Includes parallelism, great toolset, many language choices
- Arguably the most common execution platform besides raw hardware
JVM: A Behavioral Specification

Specification of behavior, not performance
JVM: Performance Matters

JVM must support high data/compute bandwidth requirements while remaining usable for domain experts

1. Enterprise apps with high transaction bandwidth (Oracle Apps)
2. Web servers (Apache Tomcat)
3. Machine learning (Apache Spark, Mahout, GraphX)
4. Prototype novel algorithms/models on JVM, run natively in production (Financial forecasting)
Running HPC applications on the JVM is still a frustrating experience:

- Various sources of jitter and overhead
- Tweaking the right knobs in the JVM for performance requires specialized JVM expertise and patience
- Abstractions get in the way of manual optimizations
Can Parallelism Help?

New parallel/functional APIs:
  • Scala parallel collections, Java parallel streams, HJ-lib

JVM inefficiencies can be somewhat offset using parallelism
  • Speedup over a slow baseline...

```
newCollection = collection
  .parallelStream()
  .map((i -&gt; i + 2)
  .filter((i -&gt; i &gt; 10));

forall((i) -&gt; {
    C[i] = A[i] + B[i];
});
```
Our Proposal

JVM use is ubiquitous
JVM overheads exist, constrained by spec
Experts can manage these overheads, the average JVM user cannot

Accelerator offload of selected parallel regions offers a promising avenue towards offsetting these overheads.

- Avoid limiting expressiveness

```java
forall_acc(0, N, (i) -> {
    p[i] = new Point(x[i], y[i], z[i]);
    if (p[i].dist(origin) > THRESHOLD) {
        throw new OutOfBoundsException();
    }
});
```
Our Proposal

JVM use is ubiquitous. JVM overheads exist, constrained by spec.
Experts can manage these overheads, but the average JVM user cannot.
Accelerator offload of selected parallel regions offers a promising avenue
towards offsetting these overheads.

- Avoid limiting expressiveness

```scala
forall_acc(0, N, (i) -> {
    p[i] = new Point(x[i], y[i], z[i]);
    if (p[i].dist(origin) > THRESHOLD) {
        throw new OutOfBoundsException();
    }
});
```
Providing Context

1. Seminal works: **APARAPI** (AMD) and **Rootbeer** (Syracuse University)
2. **Accelerating Habanero-Java Programs with OpenCL Generation.**
   Hayashi, Grossman, Zhao, Shirako, Sarkar
3. **Speculative Execution of Parallel Programs with Precise Exception Semantics on GPUs.**
   Hayashi, Grossman, Zhao, Shirako, Sarkar
4. **JaBEE: Framework for Object-Oriented Java Bytecode Compilation and Execution on GPUs**
   Zaremba, Lin, Grover

<table>
<thead>
<tr>
<th></th>
<th>J3</th>
<th>Oracle Java</th>
<th>JaBEE</th>
<th>CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program P</td>
<td>1</td>
<td>1.26</td>
<td>4.15</td>
<td>9.97</td>
</tr>
<tr>
<td>Program O</td>
<td>0.30</td>
<td>1.27</td>
<td>1.04</td>
<td>3.59</td>
</tr>
</tbody>
</table>
Contributions of this Work

Address issues that arise when supporting object references in accelerated JVM programs:

- Object serialization, code generation for object references
- Dynamic allocation of objects
- Communication optimization

```java
forall_acc(0, N, (i) -> {
    p[i] = new Point(x[i], y[i], z[i]);
    if (p[i].dist(origin) > THRESHOLD) {
        throw new OutOfBoundsException();
    }
});
```
package edu.rice.hj.example;

public class Point {
    private float x, y, z;
    private Point closest;

    public Point(float x, float y, float z, Point closest) {
        this.x = x;
        this.y = y;
        this.z = z;
        this.closest = closest;
    }

    public float getX() { return x; }
    public float getY() { return y; }
    public float getZ() { return z; }
    public float distance(Point other) {
        return (float)Math.sqrt(Math.pow(other.x - x, 2) +
                               Math.pow(other.y - y, 2) + Math.pow(other.z - z, 2));
    }

    public Point getClosest() { return closest; }
    public float distanceToClosest() { return distance(closest); }
}

typedef struct edu_rice_hj_example_Point_s {
  float x;
  float y;
  float z;
} edu_rice_hj_example_Point;

Method invocation:

float edu_rice_hj_example_Point__distance(
  __global edu_rice_hj_example_Point *this,
  __global edu_rice_hj_example_Point *other) {
  ...
}

edu_rice_hj_example_Point__distance(point, other);
class Point {
    java.lang.Double x;
    java.lang.Double y;
    java.lang.Double z;
    Point closest;
}

struct Point {
    double x;
    double y;
    double z;
};
Policy of iterative retries to support out-of-core memory allocations on the accelerator.

**Host**

do:

- launch parallel-for region on accelerator
- any_failed = check for failed allocations
- reset free
- while any_failed

**OpenCL Accelerator**

Thread $i$ requests $B$ bytes of memory:

- if $free$ is above the heap limit:
  - mark current thread as failed
  - abort and return up the stack
- else:
  - return the memory previously pointed to by $free$
Data Movement Optimization

Common parallel pattern:

```
parallel-for
```

Adding accelerators naively translates this to:

```
copy in
parallel-for
copy out
copy in
parallel-for
copy out
```
Data Movement Optimization

Common parallel pattern:
parallel-for
parallel-for

Adding accelerators naively translates this to:
copy in
parallel-for
copy in
parallel-for
copy in

Which can be optimized to:
copy in
parallel-for
copy in
parallel-for
copy out
Bytecode analysis to verify objects are not referenced between parallel regions:

```
82: invokedynamic #44, 0
87: invokestatic #45 // Method edu/rice/hj/Module1 forall_acc:(IILedu/rice/hj/api/HjProcedure;)V
90: iconst_0
91: aload 5
93: arraylength
94: iconst_1
95: isub
96: aload 5
98: aload 6
100: aload_0
101: iload_2
102: invokedynamic #46, 0
107: invokestatic #45 // Method edu/rice/hj/Module1 forall_acc:(IILedu/rice/hj/api/HjProcedure;)V
110: iinc 7, 1
113: goto 65
116: aload 5
118: areturn
```
Evaluation

Evaluation metrics:
• Performance degradation from heap contention
• Compare Java Parallel Streams, HJ-lib, HJ-OpenCL on CPU and GPU accelerators
• Kernel performance with redundant data movement
• Kernel performance without redundant data movement
• Overall application performance

Evaluation platform:
• 12-core 2.8GHz Intel X5660, 48GB RAM
• NVIDIA Tesla M2050, 2.5GB global memory
• Hotspot JVM v1.8.0_45 with -Xmx48g
• CUDA 6.0.1, OpenCL 1.1
## Heap Contention

Speedup normalized to 800KB heap on same device.

<table>
<thead>
<tr>
<th>Heap Size</th>
<th>Retries</th>
<th>CPU Time</th>
<th>Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>800KB</td>
<td>1</td>
<td>12,323 ms</td>
<td>1.07×</td>
</tr>
<tr>
<td>400KB</td>
<td>2</td>
<td>13,210 ms</td>
<td>1.07×</td>
</tr>
<tr>
<td>200KB</td>
<td>3</td>
<td>16,492 ms</td>
<td>1.34×</td>
</tr>
<tr>
<td>100KB</td>
<td>5</td>
<td>23,317 ms</td>
<td>1.89×</td>
</tr>
<tr>
<td>50KB</td>
<td>10</td>
<td>37,557 ms</td>
<td>3.05×</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Heap Size</th>
<th>Retries</th>
<th>GPU Time</th>
<th>Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>800KB</td>
<td>1</td>
<td>6,202 ms</td>
<td>1.07×</td>
</tr>
<tr>
<td>400KB</td>
<td>2</td>
<td>6,965 ms</td>
<td>1.12×</td>
</tr>
<tr>
<td>200KB</td>
<td>3</td>
<td>8,434 ms</td>
<td>1.36×</td>
</tr>
<tr>
<td>100KB</td>
<td>5</td>
<td>13,079 ms</td>
<td>2.11×</td>
</tr>
<tr>
<td>50KB</td>
<td>10</td>
<td>22,682 ms</td>
<td>3.66×</td>
</tr>
</tbody>
</table>
Kernel Perf w/ Redundant Transfers

Speedup of HJ-OpenCL GPU, relative to HJ-lib, for various kernels.

**PageRank w/ Redundant Transfers**

- calcWeights
- updateRanks

**KMeans w/ Redundant Transfers**

- classify
- updateClusters

Dataset: PageRank w/ Redundant Transfers
- N=2K
- N=4K
- N=10K
- N=14K
- N=18K

Dataset: KMeans w/ Redundant Transfers
- K=1K
- K=20K
- K=40K
- P=500K
- P=1000K
- P=2000K
Kernel Perf w/o Redundant Transfers

Speedup relative to w/ redundant transfers, for various kernels.
## Overall Speedup

Speedup normalized to Java Parallel Streams, best speedup highlighted

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Kernel</th>
<th>Accelerator?</th>
<th>HJlib</th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>KMeans</td>
<td>classifying</td>
<td>Y</td>
<td>0.97×</td>
<td>0.19×</td>
<td>0.09×</td>
</tr>
<tr>
<td></td>
<td>updateClusters</td>
<td>Y</td>
<td>1.01×</td>
<td>1.31×</td>
<td>0.61×</td>
</tr>
<tr>
<td>P=500K, K=100</td>
<td></td>
<td></td>
<td>1.23×</td>
<td>6.92×</td>
<td>10.26×</td>
</tr>
<tr>
<td>P=500K, K=40K</td>
<td></td>
<td></td>
<td>1.21×</td>
<td>5.81×</td>
<td>11.94×</td>
</tr>
<tr>
<td>NBody</td>
<td>updateVelocity</td>
<td>Y</td>
<td>1.12×</td>
<td>1.63×</td>
<td>0.71×</td>
</tr>
<tr>
<td></td>
<td>updatePosition</td>
<td>N</td>
<td>1.06×</td>
<td>6.93×</td>
<td>10.36×</td>
</tr>
<tr>
<td>P=2000K, K=100</td>
<td></td>
<td></td>
<td>1.23×</td>
<td>7.51×</td>
<td>15.78×</td>
</tr>
<tr>
<td>PageRank</td>
<td>calculateWeights</td>
<td>Y</td>
<td>1.22×</td>
<td>1.63×</td>
<td>0.71×</td>
</tr>
<tr>
<td></td>
<td>updateRanks</td>
<td>Y</td>
<td>1.10×</td>
<td>7.42×</td>
<td>11.14×</td>
</tr>
<tr>
<td>P=2000K, K=20K</td>
<td></td>
<td></td>
<td>1.23×</td>
<td>8.74×</td>
<td>18.33×</td>
</tr>
<tr>
<td>P=10K</td>
<td></td>
<td></td>
<td>0.50×</td>
<td>0.07×</td>
<td>0.08×</td>
</tr>
<tr>
<td>P=100K</td>
<td></td>
<td></td>
<td>1.02×</td>
<td>0.61×</td>
<td>0.89×</td>
</tr>
<tr>
<td>P=2K, L=40</td>
<td></td>
<td></td>
<td>1.00×</td>
<td>0.74×</td>
<td>0.45×</td>
</tr>
<tr>
<td>NBody</td>
<td></td>
<td></td>
<td>1.04×</td>
<td>0.81×</td>
<td>0.45×</td>
</tr>
<tr>
<td>P=2K, L=80</td>
<td></td>
<td></td>
<td>1.03×</td>
<td>0.80×</td>
<td>0.45×</td>
</tr>
<tr>
<td>N=4K, L=40</td>
<td></td>
<td></td>
<td>1.03×</td>
<td>1.05×</td>
<td>0.89×</td>
</tr>
<tr>
<td>PageRank</td>
<td></td>
<td></td>
<td>1.05×</td>
<td>1.05×</td>
<td>0.90×</td>
</tr>
<tr>
<td>N=4K, L=80</td>
<td></td>
<td></td>
<td>0.93×</td>
<td>1.83×</td>
<td>1.53×</td>
</tr>
<tr>
<td>P=10K, L=40</td>
<td></td>
<td></td>
<td>1.03×</td>
<td>1.41×</td>
<td>2.43×</td>
</tr>
<tr>
<td>P=10K, L=80</td>
<td></td>
<td></td>
<td>0.98×</td>
<td>2.61×</td>
<td>5.95×</td>
</tr>
<tr>
<td>N=14K, L=40</td>
<td></td>
<td></td>
<td>1.02×</td>
<td>1.85×</td>
<td>6.45×</td>
</tr>
<tr>
<td>N=14K, L=80</td>
<td></td>
<td></td>
<td>0.96×</td>
<td>3.11×</td>
<td>7.05×</td>
</tr>
<tr>
<td>N=14K, L=120</td>
<td></td>
<td></td>
<td>0.98×</td>
<td>1.69×</td>
<td>8.60×</td>
</tr>
<tr>
<td>PageRank</td>
<td></td>
<td></td>
<td>0.98×</td>
<td>1.78×</td>
<td>9.03×</td>
</tr>
<tr>
<td>N=18K, L=40</td>
<td></td>
<td></td>
<td>0.97×</td>
<td>3.06×</td>
<td>5.88×</td>
</tr>
<tr>
<td>N=18K, L=80</td>
<td></td>
<td></td>
<td>0.95×</td>
<td>1.76×</td>
<td>6.57×</td>
</tr>
<tr>
<td>N=18K, L=120</td>
<td></td>
<td></td>
<td>1.04×</td>
<td>1.66×</td>
<td>6.68×</td>
</tr>
</tbody>
</table>
Limitations

JVM object references in accelerated regions:
  • Primitive fields only
  • JaBEE showed that more complex data types may be infeasible (serialization overheads + inefficient on many accelerator architectures)
  • Ongoing work with Apache Spark loosens these limitations

Redundant transfer elimination
  • No alias analysis leads to conservative transfer decisions
Conclusions & Future Work

Improved object support in accelerated parallel regions of JVM programs
  • OpenCL code generation for object references
  • Efficient object serialization
  • Dynamic object allocation
  • Inter-region data transfer optimization.

Similarities between HJ-lib and Java Parallel Streams make this work directly related to the parallel APIs introduced in Java 8.

Ongoing work extends these techniques and applies them to Apache Spark applications