PPPJ 2015

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

Max Grossman, Shams Imam, Vivek Sarkar

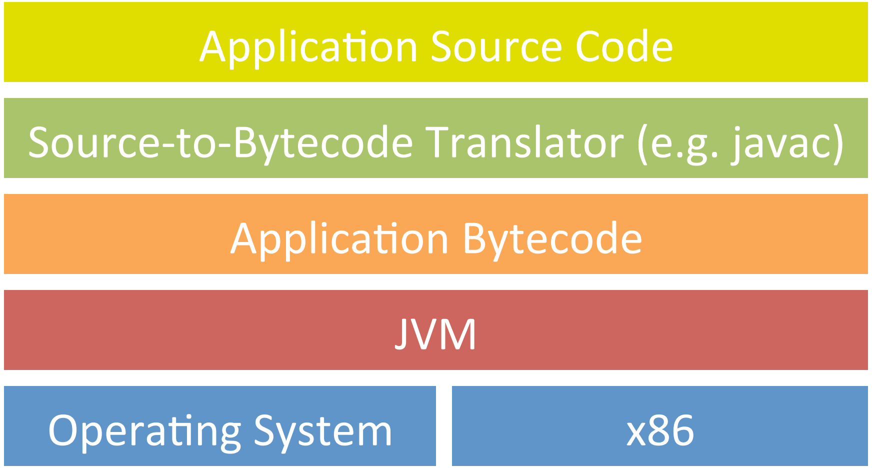
Habanero Extreme Scale Software Research Group

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)

JVM: Performance Problems

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
    .map((i : Int) => i + 2)
    .filter((i : Int) -> i > 10)
```

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

```
forall((i) -> {
        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

```
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 man… erhead… e JVM user cannot

A… flo… d para… ffers a promising avenue

t… tting these overheads.

…imiting expressiveness

Dynamic method resolution

Dynamic memory allocation

Object references

Exception semantics

```
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

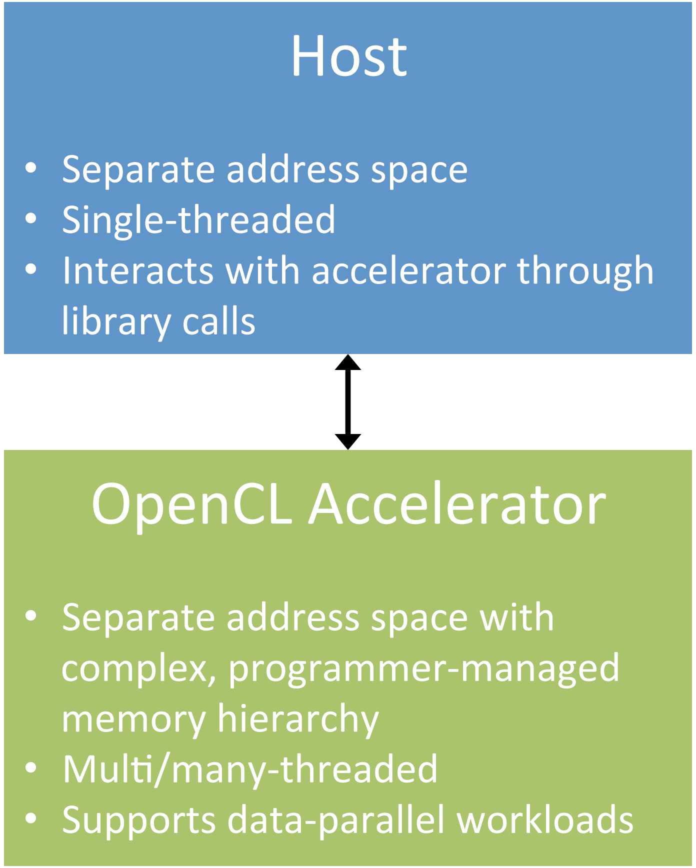
	J3	Oracle Java	JaBEE	CUDA
Program *P*	1	1.26	4.15	9.97
Program *O*	0.30	1.27	1.04	3.59

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

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



Object Serialization

Example JVM Object:

```
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);
  }
}
```

Object Code Generation

Object references in OpenCL:

```
__global edu_rice_hj_example_Point_s *point;
```

Data structure generation:

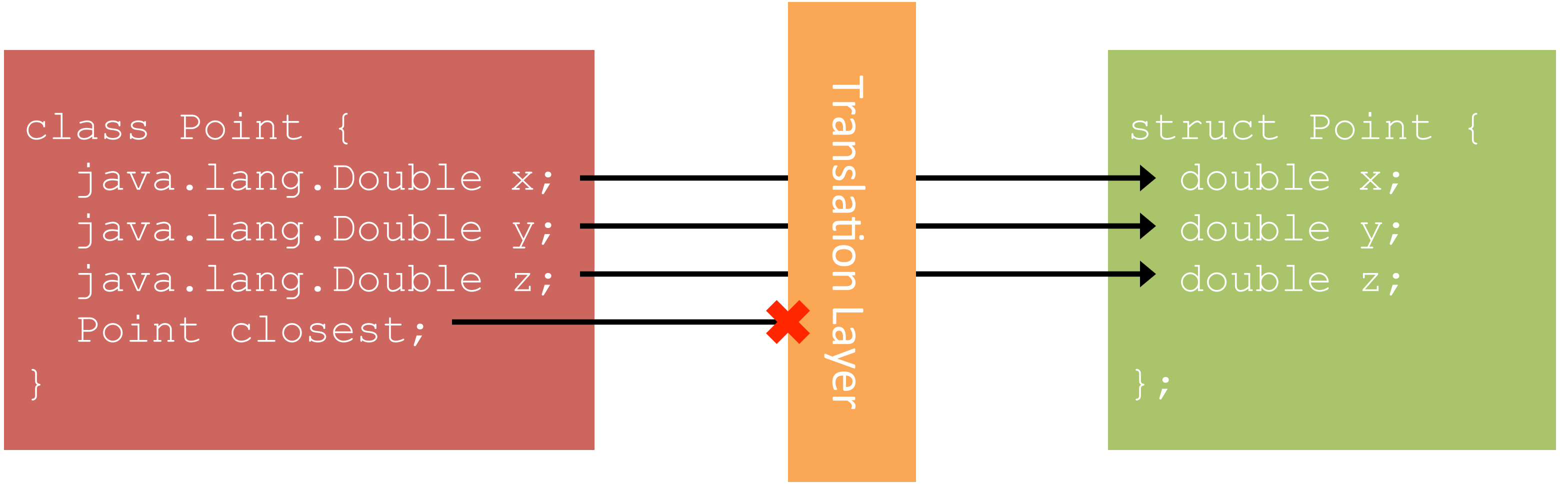
```
typedef struct edu_rice_hj_example_Point_s {
   float  x;
   float  y;
   float  z;
} edu_rice_hj_example_Point;
```

Method invocation:

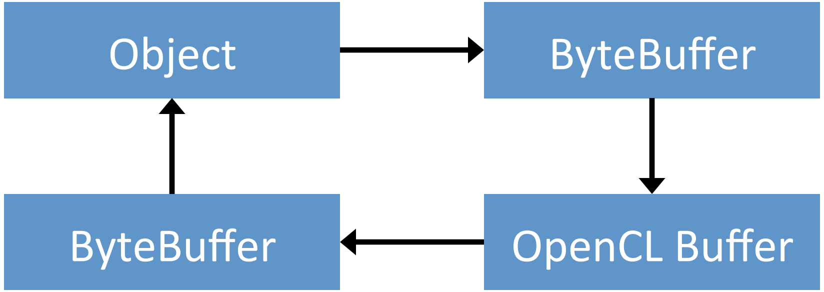
```
float edu_rice_hj_example_Point__distance(
     __global edu_rice_hj_example_Point *this,
     __global edu_rice_hj_example_Point *other) {
   ...
}

edu_rice_hj_example_Point__distance(point, other);
```

Object Serialization



Object lifecycle:



Dynamic Memory Allocation

Policy of iterative retries to support out-of-core memory allocations on the accelerator.

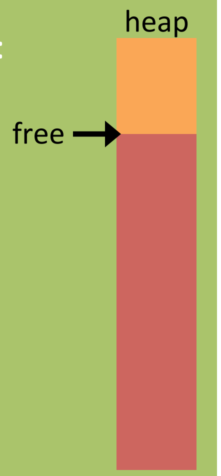
Host

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

OpenCL Accelerator

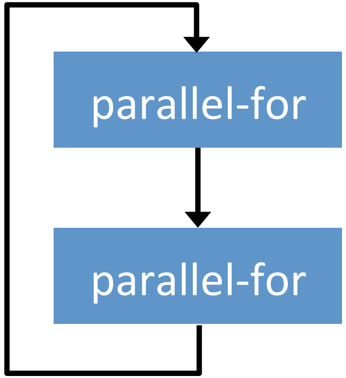
Thread *i* requests *B* bytes of memory:

free is atomically incremented by *B*
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*

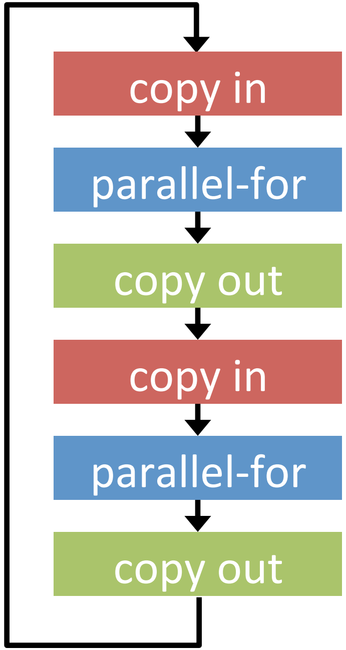


Data Movement Optimization

Common parallel pattern:

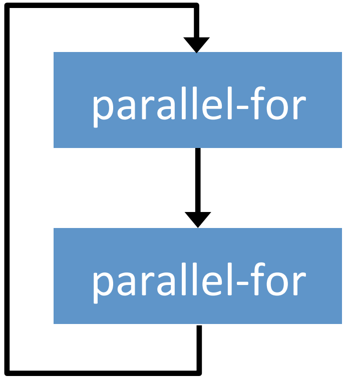


Adding accelerators naively translates this to:

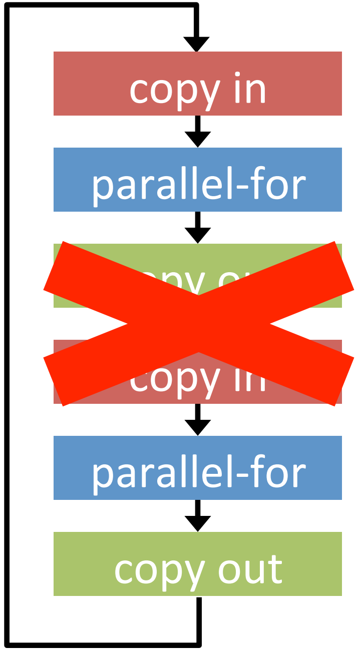


Data Movement Optimization

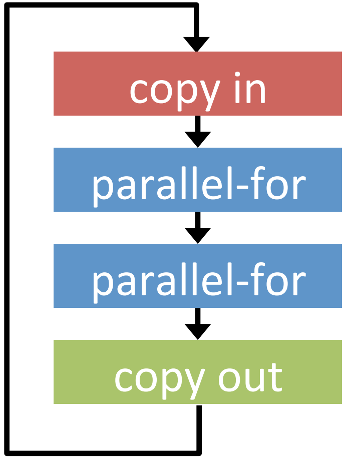
Common parallel pattern:



Adding accelerators naively translates this to:



Which can be optimized to:



Data Movement Optimization

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

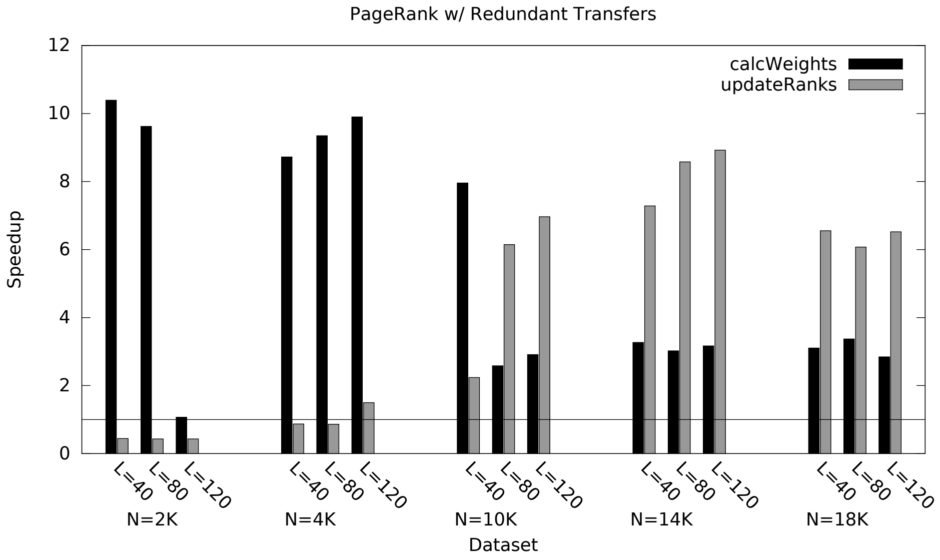
Speedup normalized to 800KB heap on same device.

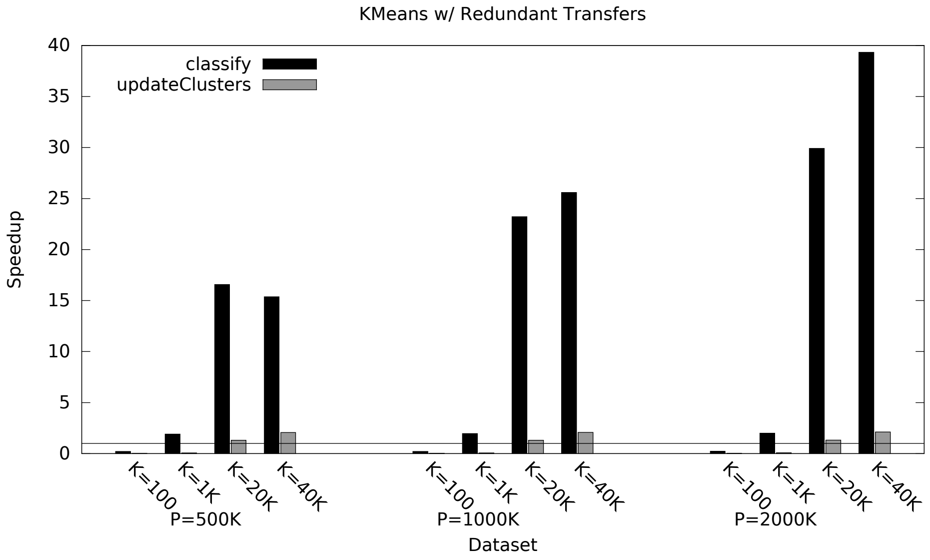
Heap Size	Retries	CPU	
		Time	Slowdown
800KB	1	12,323 ms	
400KB	2	13,210 ms	1.07×
200KB	3	16,492 ms	1.34×
100KB	5	23,317 ms	1.89×
50KB	10	37,557 ms	3.05×

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×

Kernel Perf w/ Redundant Transfers

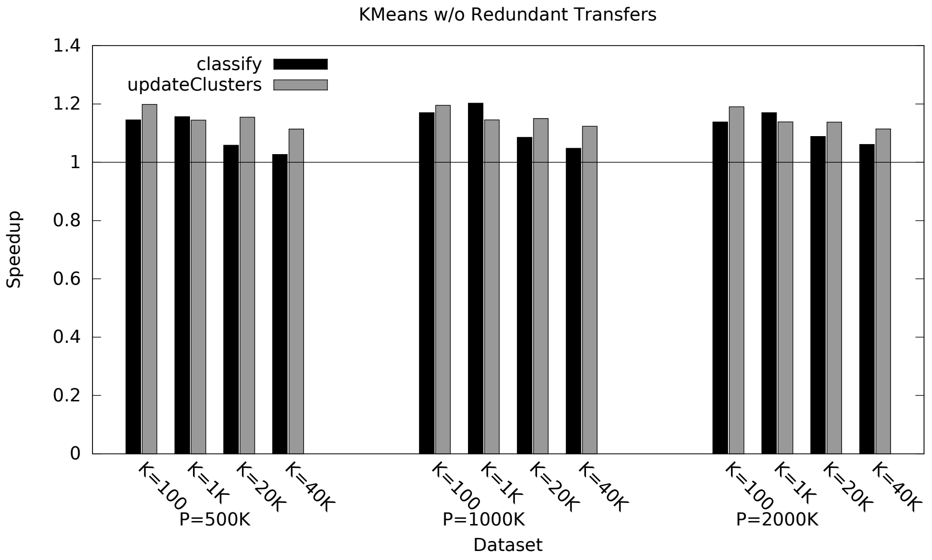
Speedup of HJ-OpenCL GPU, relative to HJ-lib, for various kernels.

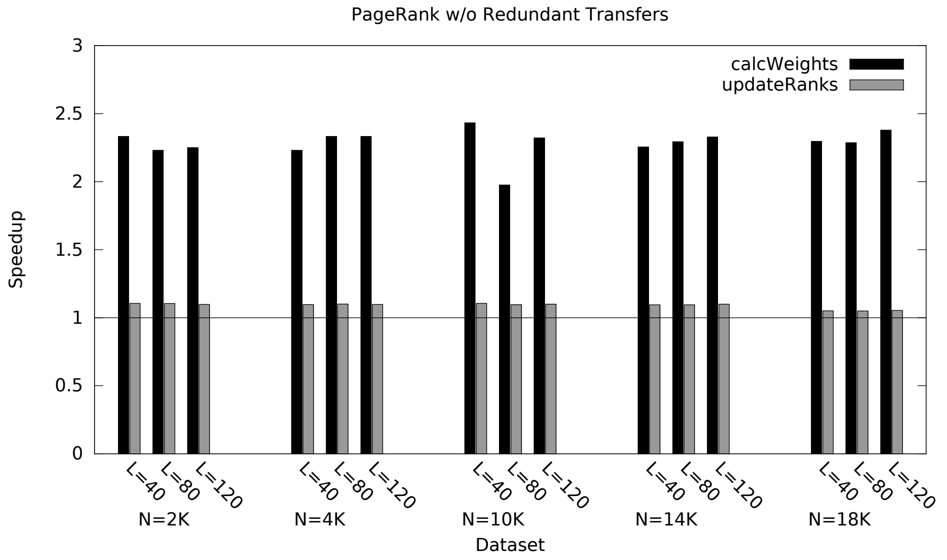




Kernel Perf w/o Redundant Transfers

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





Overall Speedup

Speedup normalized to Java Parallel Streams, best speedup highlighted

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

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

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