Easy and High Performance GPU Programming for Java Programmers

GTC 2016

Kazuaki Ishizaki (kiszk@acm.org) [+], Gita Koblents [-], Alon Shalev Housfater [-], Jimmy Kwa [-], Marcel Mitran [-], Akihiro Hayashi [*], Vivek Sarkar [*]

[+] IBM Research – Tokyo
[-] IBM Canada
[*] Rice University

Java Program Runs on GPU with IBM Java 8

IBM SDK, Java Technology Edition, Version 8

News

Abstract

IBM® SDK, Java™ Technology Edition, Version 8 is now available on z/OS® as well as AIX® and Linux platforms.

Content

IBM SDK, Java Technology Edition, Version 8 is a new release that provides compatibility with Oracle Java SE version 8 class libraries, while exploiting the unique capabilities of IBM platforms to achieve performance and usability improvements. Follow these links to read about the highlights:

- IBM z Systems exploitation
- General improvements to throughput and startup relative to IBM SDK, Java Technology Edition, Version 7 Release 1
- Improved graphics processor unit (GPU) support

Improved graphics processor unit (GPU) support

IBM SDK, Java Technology Edition, Version 7 Release 1 provides capabilities that allow certain functions to be processed on a GPU instead of a CPU, which can improve application performance. GPU support is enhanced with this release, adding new function that allows the Just-In-Time (JIT) compiler to offload specific parallel processing tasks to the GPU. This new capability exploits certain IBM POWER8™ systems with NVIDIA Compute Unified Device Architecture (CUDA) devices.

http://www-01.ibm.com/support/docview.wss?uid=swg21696670

PARALLEL FORALL Features CUDACasts CUDA Pro

← Previous Next →

The Next Wave of Enterprise Performance with Java, POWER Systems and NVIDIA GPUs

Share:

Posted on October 8, 2014 by Tim Ellison | 14 Comments
Tagged Compilation, CUDA, IBM, Java, JIT, OpenPower,

Programming GPUs with Pure Java

Once we had demonstrated the benefit to core Java APIs, we started to look at enabling user code directly. New parallel programming concepts introduced in Java 8 help here, especially lambda expressions.

Lambda expressions are blocks of code that can be assigned to variables and passed as arguments to methods. Their use can greatly simplify many coding tasks in Java that previously required creating a lot of boilerplate code (for example, coordinating and synchronizing multiple threads). Moreover, Java's

https://devblogs.nvidia.com/parallelforall/next-wave-enterprise-performance-java-power-systems-nvidia-gpus/

RICE

Java Meets GPUs

IBM SDK, Java Technology Edition, Version 8

News

Abstract

IBM® SDK, Java™ Technology Edition, Version 8 is now available on z/OS® as well as AIX® and Linux platforms.

Content

IBM SDK, Java Technology Edition, Version 8 is a new release th… bility with Oracle Java SE version 8 class libraries, while exploiting the … ns to achieve performance and usability improvements. Follow th…

- IBM z Systems exploitation
- General improvements to throughput and start-up relati… logy Edition, Version 7 Release 1
- Improved graphics processor unit (GPU) support

Improved graphics processor unit (GPU) support

IBM SDK, Java Technology Edition, Version 7 Release 1 provides capabilities that a… functions to be processed on a GPU instead of a CPU, which can improve applicatio… GPU support is enhanced with this release, adding new function that allows the Just-In… compiler to offload specific parallel processing tasks to the GPU. This new capability exploits certain IBM POWER8™ systems with NVIDIA Compute Unified Device Architecture (CUDA) devices.

PARALLEL FORALL Features CUDACasts CUDA Pro

← Previous Next →

Java

The Next Wave of Enterprise Performance with Java, POWER Systems and NVIDIA

…ison | 14 Comments

…, JIT, OpenPower,

…we started to look at …ncepts introduced in …

…ambda expressions are blocks of code that can be assigned to variables and passed as arguments to methods. Their use can greatly simplify many coding tasks in Java that previously required creating a lot of boilerplate code (for example, coordinating and synchronizing multiple threads). Moreover, Java's

What You Will Learn from this Talk

- How to program GPUs in pure Java
  - using standard parallel stream APIs
- How IBM Java 8 runtime executes the parallel program on GPUs
  - with optimizations without annotations
    - GPU read-only cache exploitation
    - data copy reductions between CPU and GPU
    - exception check eliminations for Java
- Achieve good performance results using one K40 card with
  - 58.9x over 1-CPU-thread sequential execution on POWER8
  - 3.7x over 160-CPU-thread parallel execution on POWER8

Outline

- Goal
- Motivation
- How to Write a Parallel Program in Java
- Overview of IBM Java 8 Runtime
- Performance Evaluation
- Conclusion

Why We Want to Use Java for GPU Programming

- High productivity
  - Safety and flexibility
  - Good program portability among different machines
    - "write once, run anywhere"
  - Ease of writing a program
    - Hard to use CUDA and OpenCL for non-expert programmers

<title>code ninja</title>

- Many computation-intensive applications in non-HPC area
  - Data analytics and data science (Hadoop, Spark, etc.)
  - Security analysis (events in log files)
  - Natural language processing (messages in social network system)

From https://www.flickr.com/photos/dlato/5530553658

Programmability of CUDA vs. Java for GPUs

- CUDA requires programmers to explicitly write operations for
  - managing device memories
  - copying data between CPU and GPU
  - expressing parallelism

```
void fooCUDA(N, float *A, float *B, int N) {
  int sizeN = N * sizeof(float);
  cudaMalloc(&d_A, sizeN); cudaMalloc(&d_B, sizeN);
  cudaMemcpy(d_A, A, sizeN, HostToDevice);
  GPU<<<N, 1>>>(d_A, d_B, N);
  cudaMemcpy(B, d_B, sizeN, DeviceToHost);
  cudaFree(d_B); cudaFree(d_A);
}
// code for GPU
__global__ void GPU(float* d_a, float* d_b, int n) {
  int i = threadIdx.x;
  if (n <= i) return;
  d_b[i] = d_a[i] * 2.0;
}
```

- Java 8 enables programmers to just focus on
  - expressing parallelism

```
void fooJava(float A[], float B[], int n) {
  // similar to for (idx = 0; i < n; i++)
  IntStream.range(0, N).parallel().forEach(i -> {
    b[i] = a[i] * 2.0;
  });
}
```

Safety and Flexibility in Java

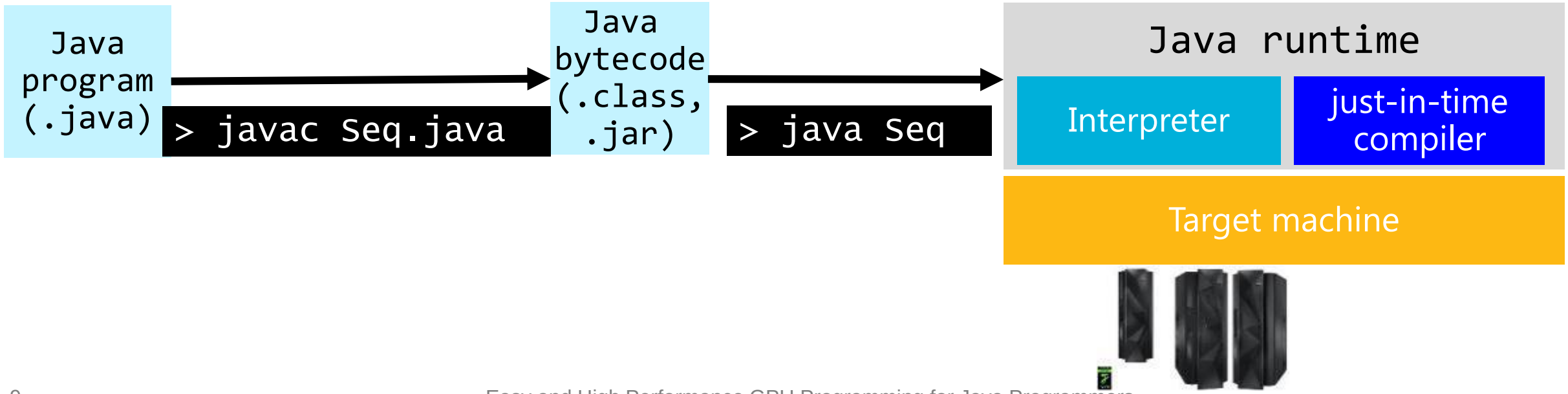
- Automatic memory management
  - No memory leak
- Object-oriented
- Exception checks
  - No unsafe memory accesses

```
float[] a = new float[N], b = new float[N]
new Par().foo(a, b, N)
// unnecessary to explicitly free a[] and b[]

class Par {
  void foo(float[] a, float[] b, int n) {
    // similar to for (idx = 0; i < n; i++)
    IntStream.range(0, N).parallel().forEach(i -> {
      // throw an exception if
      //   a[] == null, b[] = null
      //   i < 0, a.length <= i, b.length <= i
      b[i] = a[i] * 2.0;
    });
  }
}
```

Portability among Different Hardware

- How a Java program works
  - 'javac' command creates machine-independent Java bytecode
  - 'java' command launches Java runtime with Java bytecode
    - An interpreter executes a program by processing each Java bytecode
    - A just-in-time compiler generates native instructions for a target machine from Java bytecode of a hotspot method



Outline

- Goal
- Motivation
- How to Write a Parallel Program in Java
- Overview of IBM Java 8 Runtime
- Performance Evaluation
- Conclusion

How to Write a Parallel Loop in Java 8

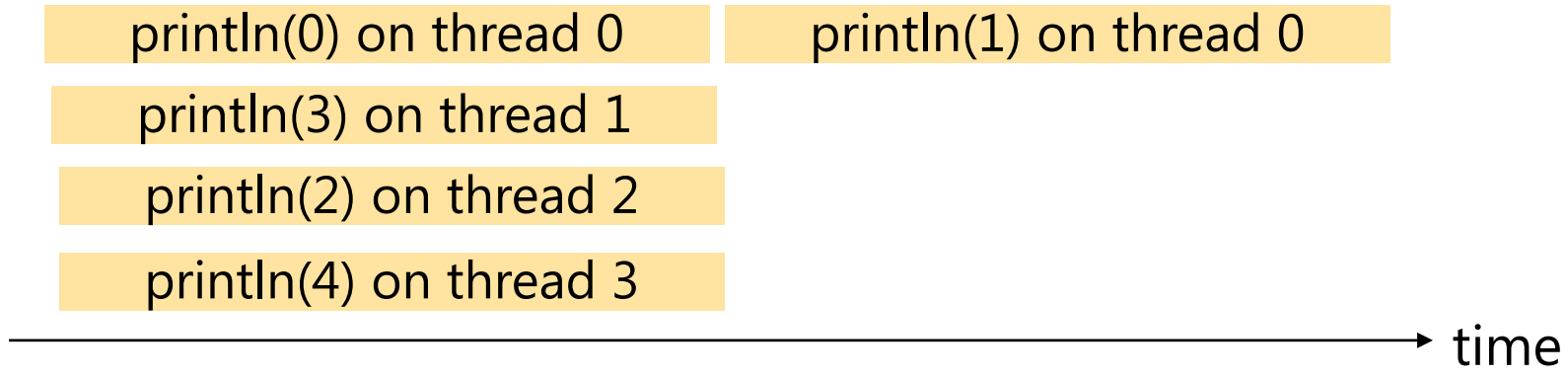
- Express parallelism by using parallel stream APIs among iterations of a lambda expression (index variable: *i*)

```
IntStream.range(0, 5).parallel().
  forEach(i -> { System.out.println(i);});
```

Example

```
0
3
2
4
1
```

Reference implementation of Java 8 can execute this on multiple CPU threads

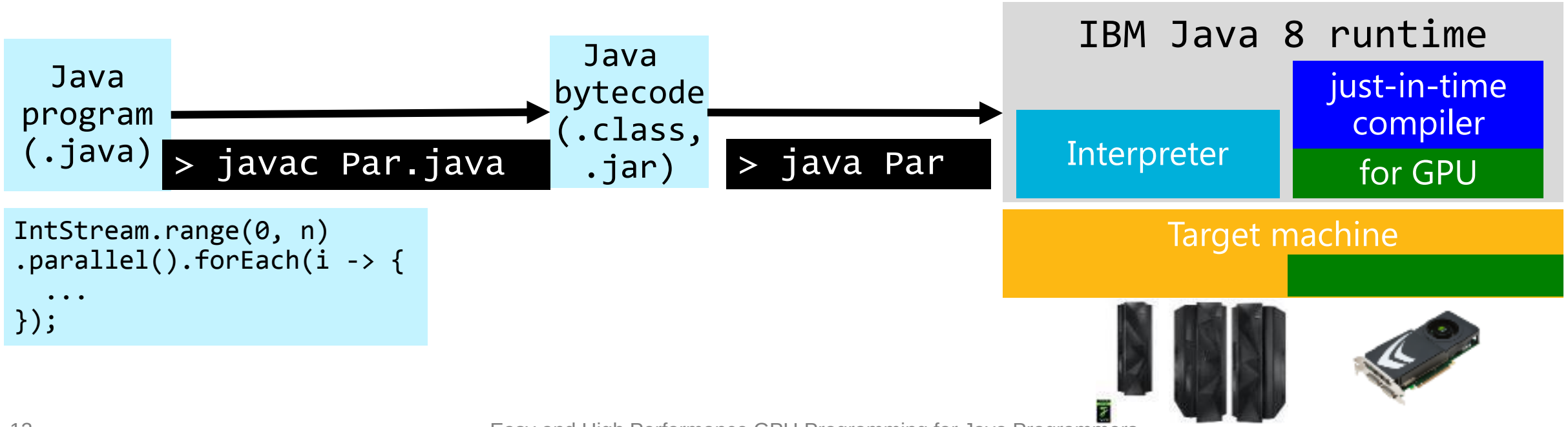


Outline

- Goal
- Motivation
- How to Write and Execute a Parallel Program in Java
- Overview of IBM Java 8 Runtime
- Performance Evaluation
- Conclusion

Portability among Different Hardware (including GPUs)

- A just-in-time compiler in IBM Java 8 runtime generates native instructions
  - for a target machine including GPUs from Java bytecode
  - for GPU which exploit device-specific capabilities more easily than OpenCL



IBM Java 8 Can Execute the Code on CPU or GPU

- Generate code for GPU execution from a parallel loop
  - GPU instructions for code in blue
  - CPU instructions for GPU memory manage and data copy
- Execute this loop on CPU or GPU base on cost model
  - e.g., execute this on CPU if 'n' is very small

```
class Par {
  void foo(float[] a, float[] b, float[] c, int n) {
    IntStream.range(0, n).parallel().forEach(i -> {
      b[i] = a[i] * 2.0;
      c[i] = a[i] * 3.0;
    });
  }
 }
```

Note: GPU support in current version is limited to lambdas with one-dimensional arrays and primitive types

Optimizations for GPUs in IBM Just-In-Time Compiler

- Using read-only cache
  - reduce # of memory transactions to a GPU global memory
- Optimizing data copy between CPU and GPU
  - reduce amount of data copy
- Eliminating redundant exception checks for Java on GPU
  - reduce # of instructions in GPU binary

Using Read-Only Cache

- Automatically detect a read-only array and access it thru read-only cache
  - read-only cache is faster than other memories in GPU

```
float[] A = new float[N], B = new float[N], C = new float[N];
foo(A, B, C, N);

void foo(float[] a, float[] b, float[] c, int n) {
  IntStream.range(0, n).parallel().forEach(i -> {
    b[i] = a[i] * 2.0;
    c[i] = a[i] * 3.0;
  });
}
```

Equivalent to CUDA code

```
__device__ foo(*a, *b, *c, N)
  b[i] = __ldg(&a[i]) * 2.0;
  c[i] = __ldg(&a[i]) * 3.0;
}
```

Optimizing Data Copy between CPU and GPU

- Eliminate data copy from GPU to CPU
  - if an array (e.g., a[]) is not written on GPU
- Eliminate data copy from CPU to GPU
  - if an array (e.g., b[] and c[]) is not read on GPU

```
void foo(float[] a, float[] b, float[] c, int n) {
  // Data copy for a[] from CPU to GPU
  // No data copy for b[] and c[]
  IntStream.range(0, n).parallel().forEach(i -> {
    b[i] = a[i] * 2.0;
    c[i] = a[i] * 3.0;
  });
  // Data copy for b[] and c[] from GPU to CPU
  // No data copy for a[]
}
```

Optimizing Data Copy between CPU and GPU

- Eliminate data copy between CPU and GPU
  - if an array (e.g., a[] and b[]), which was accessed on GPU, is not accessed on CPU

```
// Data copy for a[] from CPU to GPU
for (int t = 0; t < T; t++) {
  IntStream.range(0, N*N).parallel().forEach(idx -> {
    b[idx] = a[...];
  });
  // No data copy for b[] between GPU and CPU
  IntStream.range(0, N*N).parallel().forEach(idx -> {
    a[idx] = b[...];
  }
  // No data copy for a[] between GPU and CPU
}
// Data copy for a[] and b[] from GPU to CPU
```

How to Support Exception Checks on GPUs

- IBM just-in-time compiler inserts exception checks in GPU kernel

```
// Java program
IntStream.range(0,n).parallel().
forEach(i -> {
  b[i] = a[i] * 2.0;
  c[i] = a[i] * 3.0;
});
```

```
// code for CPU
{
  ...
  launch GPUkernel(...)
  if (exception) {
    goto handle_exception;
  }
  ...
}
```



```
__device__ GPUkernel(…) {
  int i = ...;
  if ((a == NULL) || i < 0 || a.length <= i) {
    exception = true; return; }
  if ((b == NULL) || b.length <= i) {
    exception = true; return; }
  b[i] = a[i] * 2.0;
  if ((c == NULL) || c.length <= i) {
    exception = true; return; }
  c[i] = a[i] * 3.0;
}
```

Eliminating Redundant Exception Checks

- Speculatively perform exception checks on CPU if the form of an array index is simple (x*i* + y)

```
// code for CPU
if (
  // check conditions for null pointer
  a != null && b != null && c != null &&
  // check conditions for out of bounds of array index
  0 <= a.length && a.length < n &&
  0 <= b.length && b.length < n &&
  0 <= c.length && c.length < n) {
  ...
  launch GPUkernel(...)
  ...
} else {
  // execute this loop on CPU to produce an exception
}
```

```
IntStream.range(0,n).parallel().
forEach(i -> {
  b[i] = a[i] * 2.0;
  c[i] = a[i] * 3.0;
});
```

```
__device__ GPUkernel(…) {
  // no exception check is
  // required
  i = ...;
  b[i] = a[i] * 2.0;
  c[i] = a[i] * 3.0;
}
```

Outline

- Goal
- Motivation
- How to Write and Execute a Parallel Program in Java
- Overview of IBM Java 8 Runtime
- Performance Evaluation
- Conclusion

Performance Evaluation Methodology

- Measured performance improvement by GPU using four programs (on next slide) over
  - 1-CPU-thread sequential execution
  - 160-CPU-thread parallel execution
- Experimental environment used
  - IBM Java 8 Service Release 2 for PowerPC Little Endian
    - Download for free at http://www.ibm.com/java/jdk/
  - Two 10-core 8-SMT IBM POWER8 CPUs at 3.69 GHz with 256GB memory (160 hardware threads in total)
    - With one NVIDIA Kepler K40m GPU (2880 CUDA cores in total) at 876 MHz with 12GB global memory (ECC off)
  - Ubuntu 14.10, CUDA 5.5

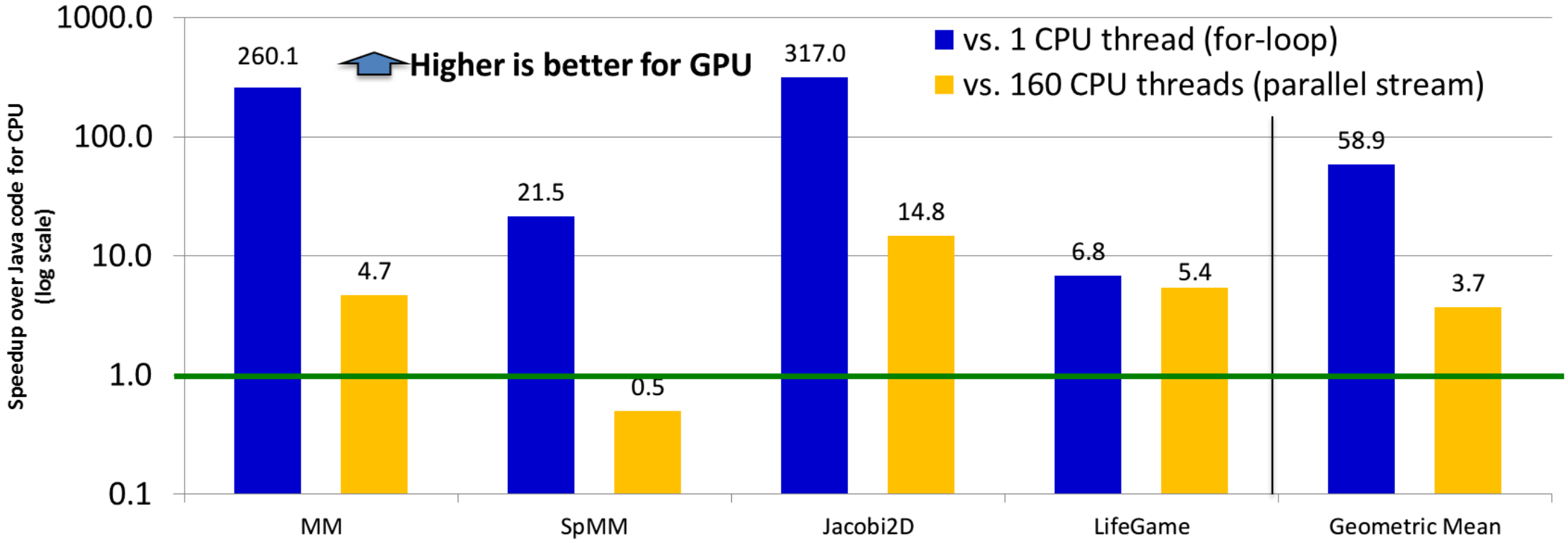
Benchmark Programs

- Prepare sequential and parallel stream API versions in Java

Name	Summary	Data size	Type
MM	A dense matrix multiplication: C = A.B	1,024 × 1,024	double
SpMM	A sparse matrix multiplication: C = A.B	500,000 × 500,000	double
Jacobi2D	Solve an equation using the Jacobi method	8,192 × 8,192	double
LifeGame	Conway's game of life. Iterate 10,000 times	512 × 512	byte

Performance Improvements of GPU Version over Sequential and Parallel CPU Versions

- ☺ Achieve 58.9x on geomean and 317.0x for Jacobi2D over 1 CPU thread
- ☺ Achieve 3.7x on geomean and 14.8x for Jacobi2D over 160 CPU threads
- ☹ Degrade performance for SpMM against 160 CPU threads



Conclusion

- Program GPUs using pure Java with standard parallel stream APIs
- Compile a Java program without annotations for GPUs by IBM Java 8 runtime with optimizations
  - read-only cache exploitation
  - data copy optimizations between CPU and GPU
  - exception check eliminations
- Offer performance improvements using GPUs by
  - 58.9x over sequential execution
  - 3.7x over 160-CPU-thread parallel execution

Details are in our paper “Compiling and Optimizing Java 8 Programs for GPU Execution” (PACT2015)