Compiling and Optimizing Java 8 Programs for GPU Execution

PACT 2015

Kazuaki Ishizaki [+], Akihiro Hayashi [*], Gita Koblents [-],
Vivek Sarkar [*]

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

Why Java for GPU Programming?

- High productivity
  - Safety and flexibility
  - Good program portability among different machines
    - “write once, run anywhere”
  - One of the most popular programming languages
    - Hard to use CUDA and OpenCL for non-expert programmers

- Many computation-intensive applications in non-HPC area
  - Data analytics and data science (Hadoop, Spark, etc.)
  - Security analysis
  - Natural language processing

Fewer Code Makes GPU Programming Easy

- 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, Host2Device);
  GPU<<<N, 1>>>(d_A, d_B, N);
  cudaMemcpy(B, d_B, sizeN, Device2Host);
  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;
  });
}
```

Goal

- Build a Java just-in-time (JIT) compiler to generate high performance GPU code from a parallel loop construct
  - Support standard Java 8 language

Contributions

- Implementing four performance optimizations
- Offering four performance evaluations on POWER8 with a GPU
  - including comparisons with hand-coded CUDA or the state-of-the-art compilation approach "Aparapi"
- Supporting precise exception semantics and virtual method calls (see the paper)

Outline

- Motivation
- Goal and Contributions
- Overview of Our Approach
- Optimizations
- Performance Evaluation
- Related Work
- Conclusion

Parallel Programming in Java 8

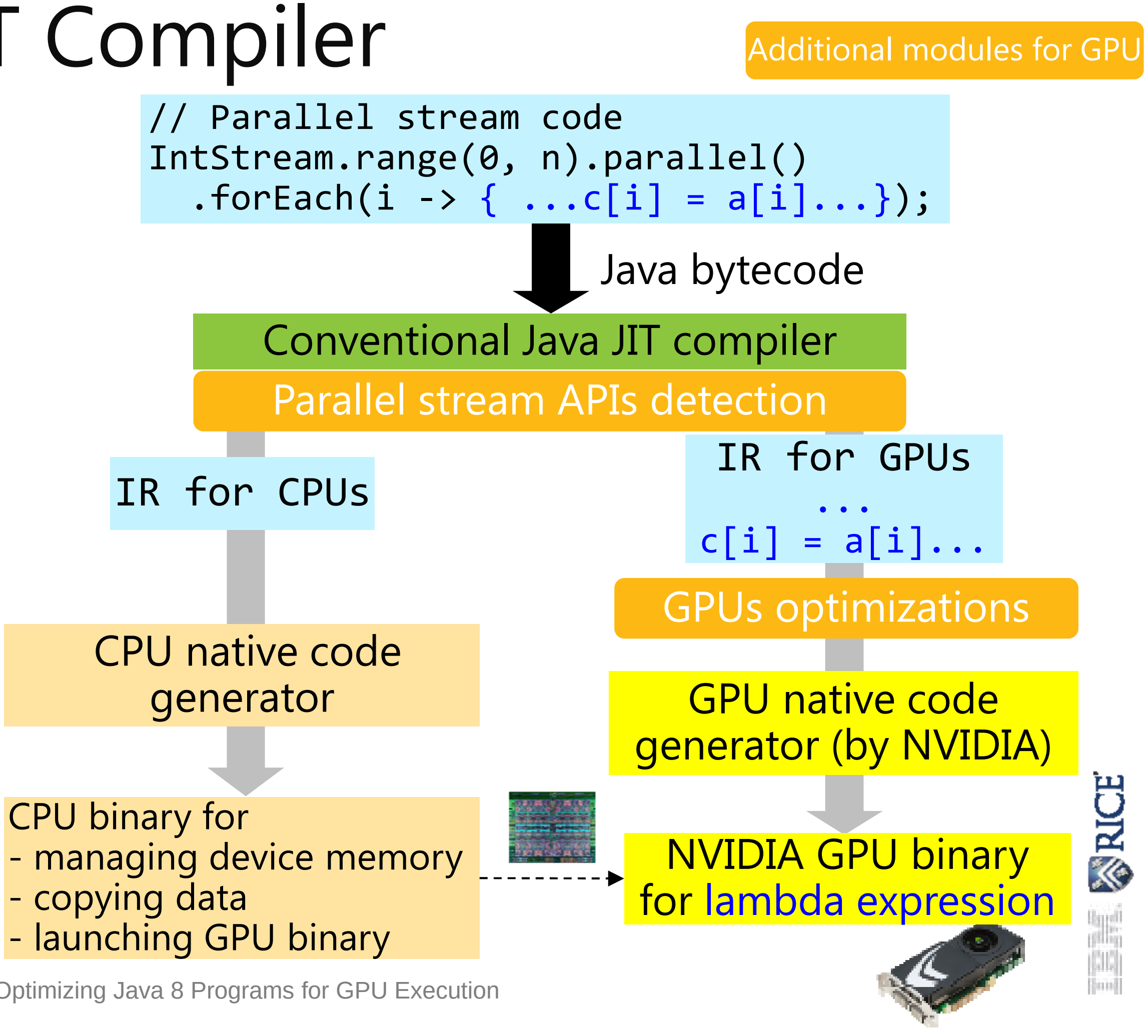
- Express parallelism by using Parallel Stream API among iterations of a lambda expression (index variable: *i*)

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

Note: Current version supports one-dimensional arrays with primitive types in a lambda expression

Overview of Our JIT Compiler

- Java bytecode sequence is divided into two intermediate presentation (IR) parts
  - Lambda expression: generate GPU code using NVIDIA tool chain (right hand side)
  - Others: generate CPU code using conventional JIT compiler (left hand side)



Additional modules for GPU

```
// Parallel stream code
IntStream.range(0, n).parallel()
  .forEach(i -> { ...c[i] = a[i]...});
```

Java bytecode

Conventional Java JIT compiler

Parallel stream APIs detection

`IR for CPUs`

```
IR for GPUs
    ...
c[i] = a[i]...
```

GPUs optimizations

CPU native code generator

GPU native code generator (by NVIDIA)

CPU binary for
- managing device memory
- copying data
- launching GPU binary

NVIDIA GPU binary for lambda expression

Outline

- Motivation
- Goal and Contributions
- Overview of Our Approach
- Optimizations
- Performance Evaluation
- Related Works
- Conclusion

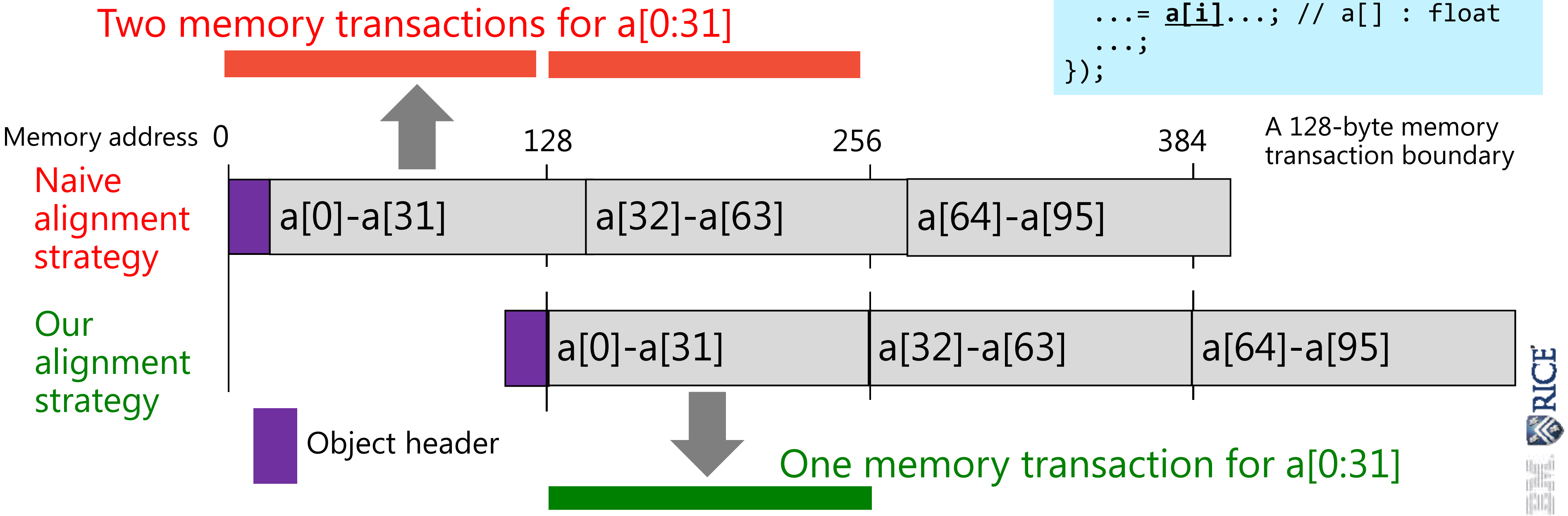
Optimizations for GPU in Our JIT Compiler

- Optimizing alignment of Java arrays on GPUs
  - Reduce # of memory transactions to a GPU global memory
- 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
  - Reduce # of instructions in GPU binary

Optimizing Alignment of Java Arrays on GPUs

- Aligning the starting address of an array body in GPU global memory with memory transaction boundary

```
IntStream.range(0,n).parallel().
forEach(i->{
  ...= a[i]...; // a[] : float
  ...;
});
```



Using Read-Only Cache

- Must keep only a read-only array in a read-only cache
  - Lexically different variables (e.g. a[] and b[]) may point to the same array that may be updated

- Perform alias analysis to identify a read-only array by
  - Static analysis in JIT compiler
    - identifies lexically read-only arrays and lexically written arrays
  - Dynamic alias analysis in generated code
    - checks a lexically read-only array that may alias with any lexically written arrays
    - executes code with a read-only cache if not aliased

```
if (!(a[] aliases with b[]) && !(a[] aliases with c[])) {
  IntStream.range(0, n).parallel().forEach( i -> {
    b[i] = ROa[i] * 2.0; // use RO cache for a[]
    c[i] = ROa[i] * 3.0; // use RO cache for a[]
  });
} else {
  // execute code w/o a read-only cache
}
```

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

Optimizing Data Copy between CPU and GPU

- Eliminate data copy from GPU if an array (e.g. a[]) is not updated in GPU binary [Jablin11][Pai12]
- Copy only a read or write set if an array index form is ‘i + constant’ (the set is contiguous)

```
sz = (n – 0) * sizeof(float)
cudaMemCopy(&a[0], d_a, sz, H2D); // copy only a read set
~~cudaMemCopy(&b[0], d_b, sz, H2D);~~
~~cudaMemCopy(&c[0], d_c, sz, H2D);~~
IntStream.range(0, n).parallel().forEach( i -> {
  b[i] = a[i]...;
  c[i] = a[i]...;
});
~~cudaMemcpy(a, d_a, sz, D2H);~~
cudaMemcpy(&b[0], d_b, sz, D2H); // copy only a write set
cudaMemcpy(&c[0], c_b, sz, D2H); // copy only a write set
```

Eliminating Redundant Exception Checks

- Generate GPU code without exception checks by using
  - loop versioning [Artigas00] that guarantees safe region by using pre-condition checks on CPU

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

```
if (
  // check cond. for NullPointerException
  a != null && b != null && c != null &&
  // check cond. For ArrayIndexOutOfBoundsException
  0 <= a.length && a.length < n &&
  0 <= b.length && b.length < n &&
  0 <= c.length && c.length < n) {
  ...
  <<<1024, n/1024>>> GPUbinary(...)
  ...
} else {
  // execute this construct on CPU
  //   to produce an exception
  //   under the original exception semantics
}
```

```
GPU binary for {
  // safe region:
  //   no exception
  //   check is required
 i = ...;
  b[i] = a[i] * 2.0;
  c[i] = a[i] * 3.0;
}
```

Automatically Optimized for CPU and GPU

- CPU code
  - handles GPU device memory management and data copying
  - checks whether optimized CPU and GPU code can be executed
- GPU code is optimized
  - Using read-only cache
  - Eliminating exception checks

CPU

```
if (a != null && b != null && c != null &&
    0 <= a.length && a.length < n &&
    0 <= b.length && b.length < n &&
    0 <= c.length && c.length < n &&
    !(a[] aliases with b[]) && !(a[] aliases with c[])) {
  cudaMalloc(d_a, a.length*sizeof(float)+128);
  if (b!=a) cudaMalloc(d_b, b.length*sizeof(float)+128);
  if (c!=a && c!=b) cudaMalloc(d_c, c.length*sizeof(float)+128);

  int sz = (n – 0) * sizeof(float), szh = sz + Jhdrsz;
  cudaMemCopy(a, d_a + align - Jhdrsz, szh, H2D);

  <<1024, n/1024>> GPU(d_a, d_b, d_c, n) // launch GPU

  cudaMemcpy(b + Jhdrsz, d_b + align, sz, D2H);
  cudaMemcpy(c + Jhdrsz, d_c + align, sz, D2H);
  cudaFree(d_a);
  if (b!=a) cudaFree(d_b);
  if (c=!a && c!=b) cudaFree(d_c);
} else {
  // execute CPU binary
}
```

GPU

```
void GPU (float *a,
  float *b, float *c, int n) {
  // no exception checks
  i = ...
  b[i] = ROa[i] * 2.0;
  c[i] = ROa[i] * 3.0;
}
```

Outline

- Motivation
- Goal and Contributions
- Overview of Our Approach
- Optimizations
- Performance Evaluation
- Related Work
- Conclusion

Performance Evaluation Methodology

- Measured performance using eight programs (on next slide)
  - Performance improvements over executions on CPU
  - Performance impact of each optimization
  - Performance comparison with the-state-of-the-art approach “Aparapi”
  - Performance comparison with hand-coded CUDA implemented by others

- Experimental environment used
  - Two 10-core 8-SMT IBM POWER8 CPUs at 3.69 GHz with 256GB memory
    - With one NVIDIA Kepler K40m GPU at 876 MHz with 12-GB global memory (ECC off)
  - Ubuntu 14.10, CUDA 5.5
  - Modified IBM Java 8 runtime for PowerPC

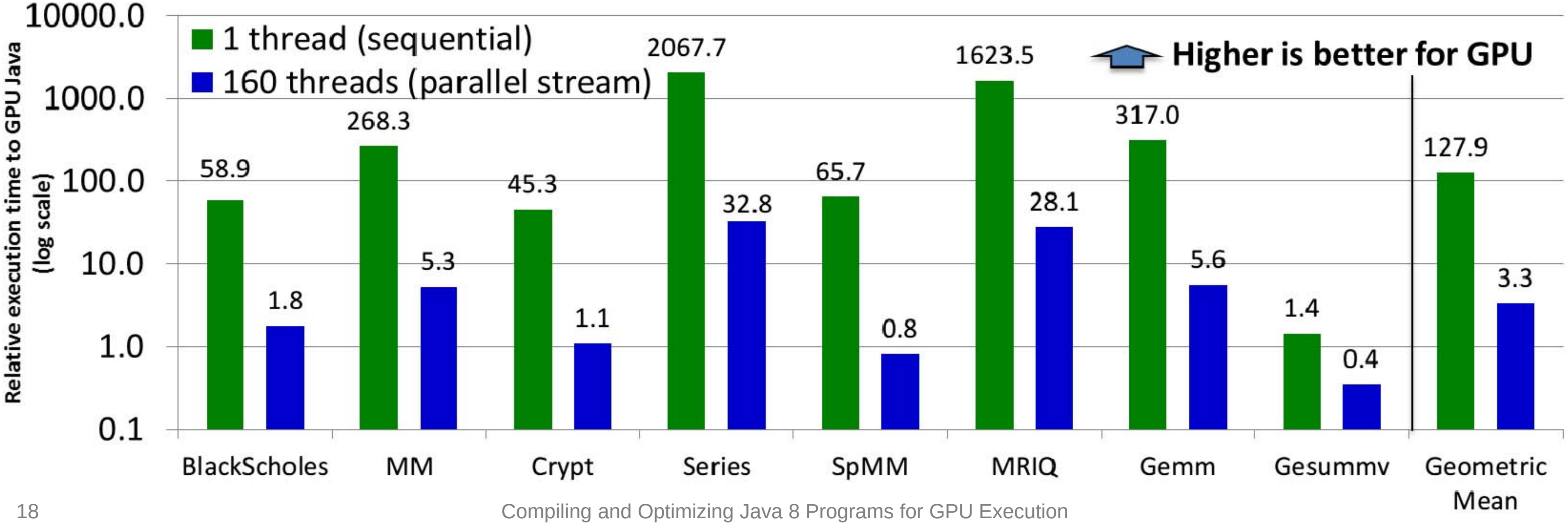
Benchmark Programs

- Prepare sequential and parallel stream API versions in Java

Name	Summary	Data size	Type
Blackscholes	Financial application that calculates the price of put and call options	4,194,304 virtual options	double
MM	A standard dense matrix multiplication: C = A.B	1,024 x 1,024	double
Crypt	Cryptographic application [Java Grande Benchmarks]	N = 50,000,000	byte
Series	the first N fourier coefficients of the function [Java Grande Benchamark]	N = 1,000,000	double
SpMM	Sparse matrix multiplication [Java Grande Benchmarks]	N = 500,000	double
MRIQ	3D image benchmark for MRI [Parboil benchmarks]	64x64x64	float
Gemm	Matrix multiplication: $C = \alpha.A.B + \beta.C$ [PolyBench]	1,024 x 1,024	int
Gesummv	Scalar, vector, and Matrix multiplication [PolyBench]	4,096 x 4,096	int

Performance Improvements of GPU Version Over Sequential and Parallel CPU Versions

- ☺ Achieve 127.9x on geomean and 2067.7x for Series over 1 CPU thread
- ☺ Achieve 3.3x on geomean and 32.8x for Series over 160 CPU threads
- ☹ Degrade performance for SpMM and Gesummv against 160 CPU threads



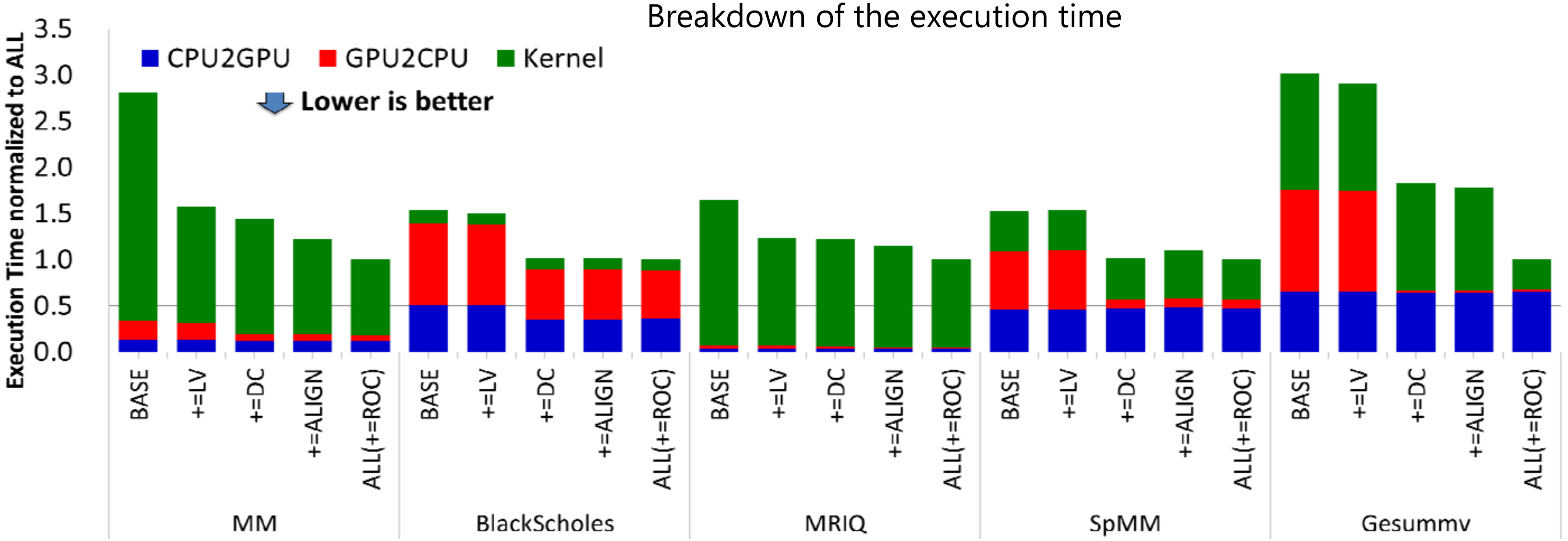
Performance Impact of Each Optimization

- ☺ MM: LV/DC/ALIGN/ROC are very effective
- ☺ BlackScholes: DC is effective
- ☺ MRIQ: LV/ALIGN/ROC is effective
- ☹ SpMM and Gesummv: data transfer time for large arrays is dominant

Apply optimizations cumulatively

- ● BASE: Disabled our four optimizations
- ● LV: loop versioning
- ● DC: data copy
- ● ALIGN: alignment optimization
- ● ROC: read-only cache

Breakdown of the execution time

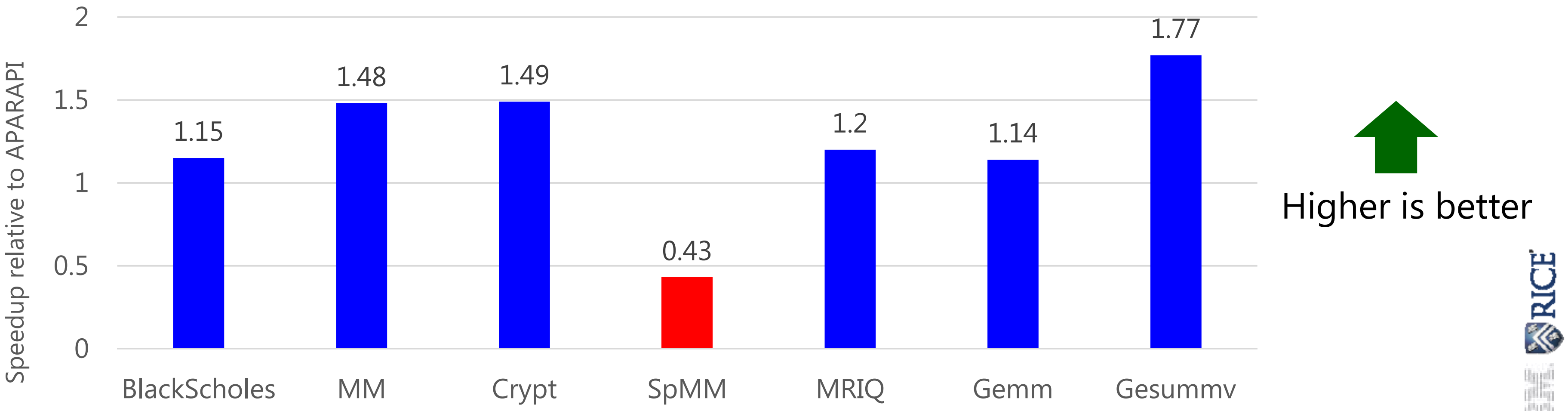


Performance Comparison with Aparapi

- Compare only GPU kernel execution time
  - Aparapi [https://github.com/aparapi/] generates OpenCL from Java bytecode
    - Aparapi does not support Java exception
    - Aparapi failed to run Series

☺ Achieve better performance (1.15x on geomean), except SpMM

- Unrolled a body of the kernel loop until PTX generation



Performance Comparison with Hand-Coded CUDA

- Achieve 0.83x on geomean over CUDA

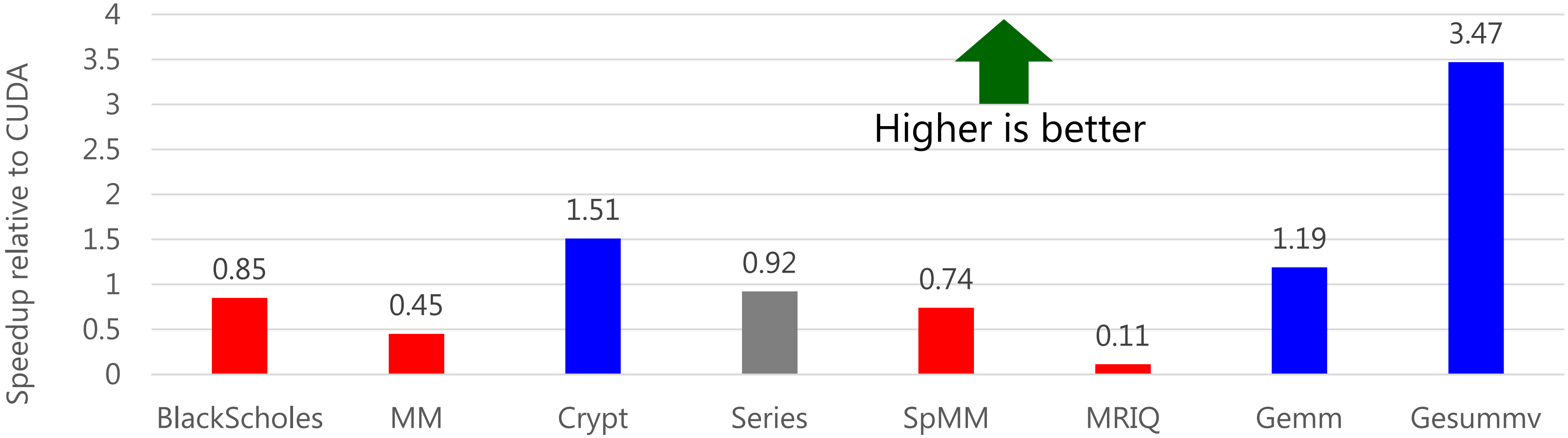
☺ Crypt, Gemm, and Gesummv: usage of a read-only cache

☹ BlackScholes: usage of larger CUDA threads per block (1024 vs. 128)

☹ SpMM: overhead of exception checks

☹ MRIQ: miss of '-use-fast-math' compile option

☹ MM: lack of usage of shared memory with loop tiling



Outline

- Motivation
- Goal and Contributions
- Overview of Our Approach
- Optimizations
- Performance Evaluation
- Related Work
- Conclusion

Related Work

- Our JIT compiler enables memory and communication optimizations
  - with standard Java parallel stream APIs and precise exception semantics

Work	Language	Exception support	JIT compiler	How to write GPU kernel	Data copy optimization	GPU memory optimization
JCUDA	Java	×	×	CUDA	Manual	Manual
JaBEE	Java	×	√	Override run method	×	×
Aparapi	Java	×	√	Override run method/Lambda	×	×
Hadoop-CL	Java	×	√	Override map/reduce method	×	×
Rootbeer	Java	×	√	Override run method	Not described	×
[PPPJ09]	Java	√	√	Java for-loop	Not described	×
HJ-OpenCL	Habanero-Java	√	√	forall constructs	√	×
Our work	**Java**	**√**	**√**	**Standard parallel stream API**	**√**	**ROCache / alignment**

Conclusion

- Built a Java JIT compiler to generate high performance GPU code from a lambda expression with parallel stream APIs
- Implemented performance optimizations
  - Optimizing alignment of Java arrays on GPUs
  - Using read-only cache
  - Optimizing data copy between CPU and GPU
  - Eliminating redundant exception checks
- Offered performance improvements by
  - 127.9x over sequential execution
  - 3.3x over 160-CPU-thread parallel execution
  - 1.15x and 0.83x over Aparapi and hand-coded CUDA