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

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

```java
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);
}
```

```java
__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
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)

```java
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)

// 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

CPU native code generator

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

IR for GPUs

c[i] = a[i]...

GPUs optimizations

GPU native code generator (by NVIDIA)

NVIDIA GPU binary for lambda expression

Additional modules for GPU
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
    ...;
});
```

Two memory transactions for a[0:31]

<table>
<thead>
<tr>
<th>Memory address</th>
<th>0</th>
<th>128</th>
<th>256</th>
<th>384</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive alignment strategy</td>
<td>a[0]-a[31]</td>
<td>a[32]-a[63]</td>
<td>a[64]-a[95]</td>
<td></td>
</tr>
<tr>
<td>Our alignment strategy</td>
<td>a[0]-a[31]</td>
<td>a[32]-a[63]</td>
<td>a[64]-a[95]</td>
<td></td>
</tr>
<tr>
<td>Object header</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A 128-byte memory transaction boundary</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

```java
if (!(a[0] aliases with b[0]) && !(a[0] aliases with c[0])) {
    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
}
```

```java
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)

```
int 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

```java
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
}
```

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

```java
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

```java
void GPU (float *a, float *b, float *c, int n) {
    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;
        cudaMemcpy(a, d_a + align - Jhdrsz, szh, H2D);
        cudaMemcpy(b + Jhdrsz, d_b + align, sz, D2H);
        cudaMemcpy(c + Jhdrsz, d_c + align, sz, D2H);
        cudaFree(d_a);
        if (b!=a) cudaMemcpy(d_b);
        if (c!=a && c!=b) cudaMemcpy(d_c);
    } else {
        // execute CPU binary
    }
}
```
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

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

<table>
<thead>
<tr>
<th></th>
<th>BlackScholes</th>
<th>MM</th>
<th>Crypt</th>
<th>SpMM</th>
<th>MRIQ</th>
<th>Gemm</th>
<th>Gesummv</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup relative to APARAPI</td>
<td>1.15</td>
<td>1.48</td>
<td>1.49</td>
<td>0.43</td>
<td>1.2</td>
<td>1.14</td>
<td>1.77</td>
</tr>
</tbody>
</table>

Higher is better
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

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