2015 International Conference on Parallel Architecture and Compilation

Compiling and Optimizing Java 8 Programs for GPU Execution

Kazuaki Ishizaki
IBM Research - Tokyo
kiszk@acm.org

Akihiro Hayashi
Rice University
ahayashi@rice.edu

Gita Koblents
IBM Canada
koblents@ca.ibm.com

Vivek Sarkar
Rice University
vsarkar@rice.edu

***Abstract*—GPUs can enable significant performance improvements for certain classes of data parallel applications and are widely used in recent computer systems. However, GPU execution currently requires explicit low-level operations such as 1) managing memory allocations and transfers between the host system and the GPU, 2) writing GPU kernels in a low-level programming model such as CUDA or OpenCL, and 3) optimizing the kernels by utilizing appropriate memory types on the GPU. Because of this complexity, in many cases, only expert programmers can exploit the computational capabilities of GPUs through the CUDA/OpenCL languages. This is unfortunate since a large number of programmers use high-level languages, such as Java, due to their advantages of productivity, safety, and platform portability, but would still like to exploit the performance benefits of GPUs. Thus, one challenging problem is how to utilize GPUs while allowing programmers to continue to benefit from the productivity advantages of languages like Java.**

This paper presents a just-in-time (JIT) compiler that can generate and optimize GPU code from a pure Java program written using lambda expressions with the new parallel streams APIs in Java 8. These APIs allow Java programmers to express data parallelism at a higher level than threads and tasks. Our approach translates lambda expressions with parallel streams APIs in Java 8 into GPU code and automatically generates runtime calls that handle the low-level operations mentioned above. Additionally, our optimization techniques 1) allocate and align the starting address of the Java array body in the GPUs with the memory transaction boundary to increase memory bandwidth, 2) utilize read-only cache for array accesses to increase memory efficiency in GPUs, and 3) eliminate redundant data transfer between the host and the GPU. The compiler also performs loop versioning for eliminating redundant exception checks and for supporting virtual method invocations within GPU kernels. These features and optimizations are supported and automatically performed by a JIT compiler that is built on top of a production version of the IBM Java 8 runtime environment.

Our experimental results on an NVIDIA Tesla GPU show significant performance improvements over sequential execution (127.9 × geometric mean) and parallel execution (3.3 × geometric mean) for eight Java 8 benchmark programs running on a 160-thread POWER8 machine. This paper also includes an in-depth analysis of GPU execution to show the impact of our optimization techniques by selectively disabling each optimization. Our experimental results show a geometric-mean speed-up of 1.15 × in the GPU kernel over state-of-the-art approaches. Overall, our JIT compiler can improve the performance of Java 8 programs by automatically leveraging the computational capability of GPUs.

***Keywords*-JIT compiler, GPU, Java 8, Parallel streams**

I. Introduction

The Java language has become increasingly popular over the last two decades because of the productivity advantages of managed runtime, which includes type safety, polymorphism, garbage collection, platform portability, and precise exception semantics. A just-in-time (JIT) compiler plays an important role in achieving significant performance improvements across multiple platforms while maintaining these productivity advantages.

Graphics processing units (GPUs) are an increasingly popular means of achieving high performance in recent computer systems. However, current programming models, such as CUDA [25] and OpenCL [19], are not easy for non-expert programmers to use since they require one to explicitly write and optimize low-level operations to manage memory allocations on the GPU, transfer data between the host system and the GPU, and use appropriate memory types in the GPU kernel such as read-only cache and shared memory. Another programming model, OpenACC [28], has recently been used for GPU application development. In this model, the programmer inserts directives or annotations that direct the compiler to generate these low-level operations automatically on the basis of its analysis of directives specified by programmers. However, they are only accessible from C/C++ and FORTRAN.

Our research is motivated by the fact that there have only been a few studies on using pure Java to generate GPU code. Since GPUs can perform the same instruction on different data in parallel, we believe the newly introduced Java 8 parallel streams APIs [29] are suitable for expressing such parallelism in a high level and machine independent manner. With our approach, a JIT compiler translates parallel loops written in Java 8 parallel streams APIs into GPU code, and automatically generates low-level operations for GPU execution. In addition, we transparently perform optimizations to exploit the computation capabilities of GPUs. In cases when a GPU is not available on some platform, the parallel streams code is executed on a fork/join thread pool as a conventional implementation does. Therefore, portability across different platforms is maintained.

This paper describes the first JIT compiler that compiles and optimizes parallel streams APIs for GPU execution while supporting Java language features such as precise

1089-795X/15 $31.00 © 2015 IEEE
DOI 10.1109/PACT.2015.46

IEEE computer society

Work	Target Language	Java Excp.	JIT Comp.	How to Write GPU Kernel	GPU Memory Optimization	Communication Optimization
JCUDA [36]	Java	×	×	CUDA	Manual	Manual
Lime [5]	Lime	×	✓	Override `map/reduce` operators	OpenCL Memory	Not described
JaBEE [37]	Java	×	✓	Override `run` method	×	×
Aparapi [1]	Java	×	✓	Override `run` method / Lambda	×	×
Hadoop-CL [9]	Java	×	✓	Override `map/reduce` method	×	×
RootBeer[33]	Java	✓	✓	Override `run` method	×	Not Described
[22]	Java	✓	✓	Java `for`-loop	×	Not Described
HJ-OpenCL [11], [12]	Habanero-Java	✓	×	`forall` construct	×	✓
Our work	Java	✓	✓	Parallel Stream API	ROCache / Align	✓

Table I
SUMMARY OF WORK ON USING JVM-COMPATIBLE LANGUAGES TO GENERATE GPU CODE

exception semantics and virtual method invocations. Our compiler is implemented on top of the IBM Testarossa JIT compiler [8] in IBM SDK Java Technology Edition, Version 8 [14]. This paper also introduces four new optimizations for high performance that do not require the user to insert directives or annotations. In particular, the JIT compiler automatically performs memory optimizations to reduce the number of memory transactions and to increase cache utilization in GPUs [23]. It also eliminates redundant data transfer between the host and GPU [16], [17]. To support Java language features, it performs loop versioning [2] for speculative exception checks [22] and for supporting virtual method calls within GPU kernels [10].

We conducted experiments that ran eight benchmark programs on a NVIDIA Tesla GPU, in which our JIT compiler showed significant performance improvements over sequential and parallel versions of a 160-thread POWER8 machine. Our compiler also outperformed Aparapi [1] and performed comparably to hand-written CUDA programs.

To the best of our knowledge, no previous study has compiled and optimized Java 8 parallel streams APIs. Some studies [1], [11], [12], [22], [33] have devised optimizations for loop versioning and communication elimination, and we have used them for compiling programs written purely in Java, along with devising new optimizations to achieve high performance while maintaining the safety and portability of these programs.

To summarize, this paper makes the following contributions.

- Supporting Java language features such as precise exception checks and virtual method calls in GPU execution (see Section IV).
- Implementing performance optimizations that are transparently enabled without the user's assistance (see Section V), by
  - Aligning the starting address of a Java array on a GPU with a memory transaction boundary (see Section V-A)
  - Utilizing read-only cache for array accesses (see Section V-B)
  - Optimizing data transfers for a partial region of an array (see Section V-C)
  - Eliminating exception checks by loop versioning, which is an extension of previous work (see Section V-D)
- Offering detailed performance evaluations (see Section VI),
  - Showing performance improvements of up to 2067.7 × (geometric mean of 127.9 ×) relative to sequential execution, and performance improvements up to 32.8 × (geometric mean of 3.3 ×) relative to parallel execution for eight Java benchmark programs on a 160-thread POWER8 machine (see Section VI-A).
  - Evaluating the effectiveness of improvements by selectively disabling the individual optimizations described in Section V (see Sections VI-B).
  - Showing a 1.15 × geometric mean performance improvements relative to Aparapi [1] (see Section VI-C). Aparapi is the current state-of-the-art approach for generating GPU code from Java, but it does not support language features like precise exception semantics and virtual method calls, as in our approach.
  - Showing that our approach comes within 83% (geometric mean) of performance of hand-written CUDA code while maintaining Java's precise exception semantics (see Section VI-D).

II. MOTIVATION

Java 8 offers more opportunities for parallel programming than previous editions have supported. Our work is motivated by the fact that the Java 8 parallel streams API provides a good starting point for enabling portable execution of parallel kernels across a wide range of platforms including multi-core CPUs and many-core GPUs.

Table I compares ours and previous studies on using Java-compatible languages to generate GPU code. All prior approaches provide an external compiler / class libraries to increase programmability so that programmers do not need to write low-level operations for GPU execution. In contrast, one of our goals is to compile pure Java programs for GPU

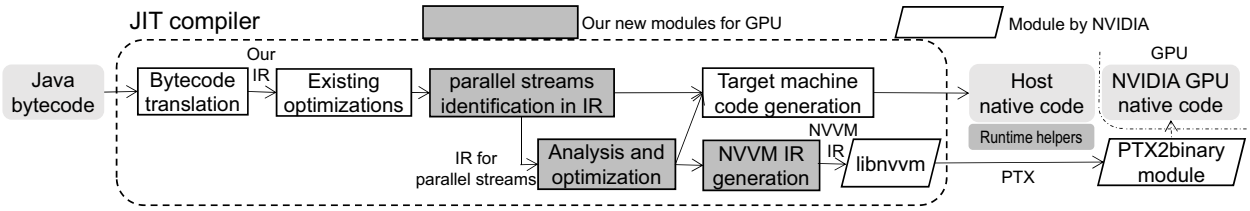

Figure 1. JIT compiler overview

execution without any extensions, and we believe that the new Java 8 parallel streams API is promising way to achieve it. In Java 8, data parallelism can be expressed as parallel streams API with lambda expressions as follow:

```
IntStream.range(low, up).parallel().forEach(i -> <lambda>)
```

This API iterates a lambda expression containing a lambda parameter (i) and lambda body ($\langle lambda \rangle$) in parallel within the range of $low \leq i < up$, where i is a lambda parameter, up is an upper exclusion limit and low is a lower inclusion limit. It is worth noting that the OpenJDK Sumatra project [34], which is not publicly available as of this writing, also tries to utilize this API. Section VI-C will discuss the difference between our approach and Aparapi with lambda expressions. Aparapi also uses lambda expressions to generate GPU code.

Kernel and communication optimizations are very important for GPU execution [16], [17], [23]. However, these optimizations are limited in many of the cases shown in Table I (see the GPU Memory Optimization and Communication Optimization columns). For example, many of the prior approaches copy arrays back from GPUs even if they were not updated on the GPUs. Our approach handles such case, and makes use of additional data transfer optimizations as far as possible. For kernel execution, we propose new optimization techniques that align and allocate the starting address of a Java array body in GPU memory with the memory transaction boundary to increase memory bandwidth in GPUs and that utilize read-only cache for array accesses to increase memory efficiency in GPUs (See Section V).

Preserving Java exceptions is also an important challenge because programmers want and would expect to be able to do standard Java programming even with GPUs. This issue has been ignored in some of the previous studies; by contrast, our approach handles Java's precise exception semantics with GPUs.

In summary, our work combines some of the previous contributions but also devises several new optimization techniques for GPUs. A detailed discussion on the differences between ours and the previous implementations can be found in Section VII.

III. Overview

A. Structure of the JIT compiler

Our JIT compiler for GPUs is built on top of the production version of the IBM Java 8 runtime environment [14] that consists of the J9 virtual machine and Testarossa JIT compiler [8]. The Java runtime environment determines the method to be compiled on the basis of runtime profiling information. Figure 1 shows an overview of our JIT compiler. First, the compiler transforms Java bytecode of the compilation target method into an intermediate representation (IR), and then applies numerous advanced optimizations. It does so by using the existing optimization modules that were originally designed for CPUs, such as dead code elimination, copy propagation, and partial redundancy elimination.

Next, the JIT compiler looks for a call to the `java.util.Stream.IntStream.forEach()` method with `parallel()`. If it identifies the method call, the IR for a lambda expression in `forEach()` is extracted together with a pair of lower and upper bounds. After this identification, it transforms this parallel `forEach` into a regular loop in the IR. We will refer to this parallel `forEach` as a parallel loop in the following. Then, the compiler analyzes the IR, and applies optimizations to it. The optimized IR consists of two parts. One is translated into an NVVM IR [26] to be executed on the GPU. The other part is translated into a host binary, which includes calls to CUDA driver APIs to allocate memory to GPUs, transfer data between the host and the GPU, call a GPU binary translator with PTX instructions [27]. This allows Java programmers to avoid explicitly writing low-level operations. At runtime, the host binary calls a CUDA Driver API to compile PTX instructions into a NVIDIA GPU's binary, after which the GPU binary will be executed.

We chose to generate PTX instructions through NVVM IR, which is tightly coupled with the NVIDIA GPU, to enable aggressive GPU-aware optimizations, although we could have easily generated GPU code through OpenCL SPIR [20] instead.

B. Supporting Java constructs

Our compiler can currently generate GPU code from the following two styles of inner most parallel streams code to express data parallelism. Supporting nested parallel streams will be a subject of our future work.

Listing 1. An example program

```
1 class Par {
2   float bar(float f, int r) { return f * (1000 / (r - 7)); }
3   void foo(float[] a, float[] b, float[] c, int n, Par p) {
4     IntStream.range(0, n).parallel().forEach(i -> {
5       b[i] = p.bar(a[i], n);
6       c[i] = b[i];
7     });
8   }
9 }
```

```
IntStream.range(low, up).parallel().forEach(i -> <lambda>)
IntStream.rangeClosed(low, up).parallel().forEach(i -> <lambda>)
```

The function `rangeClosed(low, up)` generates a sequence within the range of $low \leq i \leq up$, where i is a lambda parameter, up is an upper inclusion limit and low is a lower inclusion limit. $\langle lambda \rangle$ refers to a valid Java lambda body with a lambda parameter i whose input is the above sequence of integer values. The following constructs are supported in $\langle lambda \rangle$:

- **types**: all of the Java primitive types
- **variables**: local, parameters, and instance variables and one-dimensional arrays of primitive types whose references are loop invariant
- **expressions**: all of the Java expressions for primitive types
- **statements**: all of the Java statements except the `throw` and `try-catch-finally`, `synchronized`, `interface` and `JNI` method calls, and other aggregate operations of the streams such as `reduce()`
- **exceptions**: `ArrayIndexOutOfBoundsException`, `NullPointerException`, and `ArithmeticException` (only division by zero)

Even though our compiler only supports one-dimensional arrays, programmers can still use multi-dimensional arrays by calculating an index of each dimension themselves. Although object creations such as `new` are not supported, most computation-intensive applications can be executed under this restriction. This is because they would avoid allocating objects in parallel streams code in order to achieve high performance.

An extended version of IBM Java 8 runtime supports machine-learning-based performance heuristics for runtime CPU/GPU selection for a given parallel streams code [13]. However, we disabled this feature in order to focus on our study on compilation and optimization.

Listing 1 contains a running example program that we will refer to throughout this paper.

IV. How to support exception checks, virtual method calls, and array aliasing

This section describes how we addressed the technical challenges posed by Java language features in the context of GPUs. The first challenge is how to support exception checks on GPUs to ensure the safety of Java programs. The second one is how to support virtual method calls that are frequently used in Java programs. The last challenge is how to handle array aliasing among lexically different variables that hold the same array reference.

A. Exception checks

The Java specification [7] states that all Java exceptions must be thrown precisely: all effects of the statements and expressions executed prior to the exception must appear to have taken place. For lambda expressions in parallel streams API, exceptions thrown by all iterations concurrently should be thrown out of the lambda expression. The order of exceptions in each iteration must still be precise.

To preserve these exception semantics even on a GPU, our compiler embeds exception checking code into the GPU code. A program execution falls back to the original parallel streams code that is executed on the CPU when exceptions occurs during GPU execution. During execution of the original parallel streams code, an exception will occur while correctly maintaining all of the side effects such as updating global variables. However, this approach may not always be efficient, since executing exception checks on a GPU may add some overhead. Beyond this approach, Section V will describe how applying loop versioning helps to remove redundant exception checks on GPUs [12].

B. Virtual method calls

It is important to support virtual method calls within a lambda expression used in parallel streams APIs since virtual calls are frequently used in Java programs.

A native implementation of a virtual method call requires a receiver object to identify the target method. Performing a dynamic dispatch on a GPU [37] requires transferring a receiver object together with a virtual method table to the GPU. This transfer is considerably more complex than transferring a Java array of a primitive type. This is because it requires an address translation of pointers in the receiver object and a virtual table to be sent from the CPU to the GPU.

To avoid this overhead, our compiler applies either direct or guarded devirtualization. Direct devirtualization replaces a virtual method call with a non-virtual method call if the receiver of the call is loop invariant, and can be uniquely identified at JIT-compile time. If a loop-invariant guard is required, a target method is chosen based on runtime profiling [4]. Then, our compiler moves the guard out of the parallel loop by loop versioning [2]. We generate two versions of code consisting of 1) the optimized parallel loop with a non-virtual call that does not access the receiver object and 2) the original parallel streams code with the original virtual call. Since the guarded code is a conditional branch, the branch taken goes to the optimized parallel loop and the branch not taken goes to the original parallel streams code.

Our JIT compiler applies method inlining to the devirtualized method call in the optimized parallel loop. As a result,

Listing 2. Pseudo code for versioned loops to support a virtual method

```
1 ...
2 float bar(float f, int r) { return f * (1000 / (r - 7)); }
3 void foo(float[] a, float[] b, float[] c, int n, Par p) {
4   // precheck before executing a parallel loop
5   if (isNonOverridden(p, Par.bar())) {
6     // an optimized parallel loop with inlined code for GPU
7     IntStream.range(0, n).parallel().forEach(i -> {
8       b[i] = a[i] * (1000 / (n - 7)); // inlined Par.bar()
9       c[i] = b[i];
10    });
11  } else {
12    // the original parallel streams code for CPU
13    IntStream.range(0, n).parallel().forEach(i -> {
14      b[i] = p.bar(a[i], n); // with a virtual method call
15      c[i] = b[i];
16    });
17  }
18 }
```

an optimized parallel loop does not require transferring a receiver object and a virtual table to the GPU, and is executed on the GPU. Since previous work [15] revealed that more than 70% of virtual calls are monomorphic and inline code would be executed in many cases, this approach is quite effective in practice..

Listing 2 shows the pseudocode obtained after applying guarded devirtualization, method inlining, loop versioning to the virtual method call `p.bar()` in Listing 1. The guard `isNonOverridden(obj, method)` checks whether the `method` in the `obj` is overridden in the class hierarchy. If this check ensures that the method `Par.bar` is not overridden, the inline code is executed on the GPU. Otherwise, the original parallel streams code is executed on the CPU.

C. Array aliasing

Array aliasing may occur among lexically different variables that hold references to the same array. It is not easy to detect this situation by using only static analysis. If a runtime system cannot recognize array aliasing, an allocation unit on the GPU memory that corresponds to the Java array is allocated as many times as there are variables holding a reference to that array. Therefore, it may happen that the same Java array is represented by different allocation units in different parts of the program which will either cause incorrect results, or require runtime overhead to merge the results (e.g., by using Array SSA form [21]). Our implementation handles array aliasing by using a runtime approach [30]. This approach ensures that one Java array corresponds to a single GPU allocation unit. We check whether any pair of Java arrays to be transferred to a GPU alias each other at runtime. If an alias is detected, aliased array shared the same GPU allocation unit. This is especially convenient for Java arrays, since (unlike C arrays) it is not possible to create an array alias in Java that starts in the middle of another array.

V. JIT Compiler Optimizations

This section describes several optimizations that we implemented in our JIT compiler in support of GPU execution. These optimizations are automatically applied without the programmer having to make decisions such as when to insert annotations or directives. The following are the optimizations performed by our compiler:

1) aligning the starting address of the Java array body in GPU memory with the memory transaction boundary, which avoids overfetching due to misaligned accesses.
2) utilizing read-only cache, which exploits data locality.
3) optimizing data transfers, which can reduce the amount of data transferred between the host and the GPU.
4) eliminating redundant exception checks on GPUs by using loop versioning.

Items 1 and 2 represent new optimizations that are specific to GPU execution. Item 3 combines a new GPU optimization with existing CPU optimiations. Item 4 is an extension of an existing CPU optimization.

A. Aligning the starting address of a Java array body in GPU memory with a memory transaction boundary

Global memory access in GPUs is done at a granularity of 32 consecutive threads called a *warp*. In a case where 32 consecutive global memory locations are accessed by a *warp* and the starting address is aligned, these memory accesses are coalesced into a single memory transaction. This increases memory bandwidth utilization in GPUs and leads to higher performance. In the Tesla K40m GPU, the granularity of a memory access transaction to a global memory is 128 bytes, when a L2 cache miss happens. For example, 32 consecutive 4-byte accesses are coalesced and use 100% of the bus if the starting address is aligned. Otherwise, multiple memory transactions may overfetch redundant words due to misaligned accesses.

Arrays in managed languages such as Java and Python have small (typically 8-16 bytes) headers in front of the actual data regions, e.g., objects or arrays. We devised a new object alignment strategy that aligns the starting address of an array body with a memory transaction boundary bacause the array's body is generally accessed more frequently than its header. For each array in a parallel loop, our compiler 1) calculates a range of read and write elements, 2) identifies an element with the minimum array index, and 3) aligns the address of the element with a memory transaction boundary. In practice, the first element of the array would be aligned with a memory transaction boundary. Since GPU memory allocation routines such as `cudaMalloc` always return a 128-byte aligned address, we use the address in GPU memory access instructions with an address offset . The offset is calculated as $128-((hSize+minIdx)\ mod\ 128)$, where $hSize$ is the size of the array header in bytes and $minIdx$ is the size of the elements in bytes, which is identified by 2). This is different from previous alignment strategies work [33], [34] that allocate and transfer an array for Java to GPU memory by aligning its header with the memory access

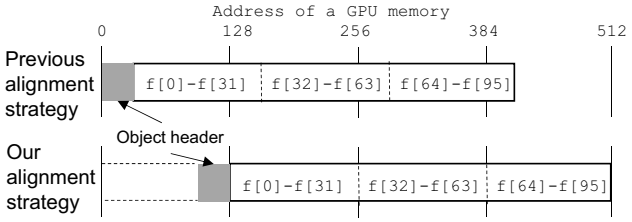

Figure 2. Comparison of previous and our object alignment strategies. The type of the array f[] is float

boundary. The previous strategies would have thus more memory transactions compared with ours. Figure 2 compares the memory layouts. While the existing strategy generates two memory transactions for accessing `f[0]-f[31]` in a *warp*, our approach generates only one memory transaction for these accesses.

B. Utilizing read-only cache

A read-only cache is not coherent enough with other memory accesses to shorten memory latency within a streaming multiprocessors extreme (SMX). A runtime system must ensure that nothing is written into an array that is placed into read-only cache.

Our approach combines static analysis and dynamic analysis to ensure that there is no data that may be written. Static analysis is used for detecting *possibly read-only* arrays in parallel streams. This can be done by inspecting ILs and searching arrays not having write access. Our JIT compiler then generates a guard conditions that checks if *possibly read-only* arrays are *definitely read-only* by including runtime alias checking code. Thus, it generates two versions of code that consist of a parallel loop with and without read-only cache (See Listing 3). The current implementation executes the latter code on the CPU to reduce the size of GPU code. Algorithm 1 presents a pseudocode of the `isUseROC` function that performs the dynamic analysis. `allocInfo` contains information about an object regarding the pointer such as the access types of the object. If lines 6 or 7 in Algorithm 1 detect an alias between a read-only array and an array to be written to, the code that does not use the read-only cache will be executed to ensure correct results. If no alias is detected between a read-only array and an array to be written, the code that uses the read-only cache will be executed in order to reduce the number of memory transactions to the global memory. This process is similar to that of preserving alias relations in GPU memory [17]. In addition, our algorithm takes in account access type associated with each pointer. It checks whether there is an alias between pointers with different access types.

C. Optimizing data transfers

Data transfers between the host system and GPUs incurs significant communication overheads due to the limited data

Algorithm 1: `isUseROC`(ptr, isReadOnly, isWrite)

input : *ptr*: An address that holds a pointer to a Java array
input : *isReadOnly*: Is the array *ptr* read-only?
input : *isWrite*: Has the array *ptr* been updated?
output: whether the code that uses ROC can be executed

1 $info \leftarrow getAllocInfo(allocInfo, ptr)$;
2 $setAllocInfo(allocInfo, ptr, isReadOnly, isWrite)$;
3 **if** $info == NULL$ **then**
4 | **return true**
5 **end**
6 $b1 \leftarrow (info.isReadOnly \wedge isWrite)$;
7 $b2 \leftarrow (info.isWrite \wedge isReadOnly)$;
8 **return** $\neg(b1 \vee b2)$;

Listing 3. Pseudo code for versioned loops to utilize read-only cache

```
1  void foo(float[] a, float[] b, float[] c, int n, Par p) {
2    if (isNonOverridden(p.bar)) {
3      if (isUseLoc(a, T, F) && isUseLoc(b, F, T) &&
4          isUseLoc(c, F, T)) {
5        // ensure: (a != b) and (b != c) and (c != a)
6        // optimized parallel loop w/ read-only cache for GPU
7        IntStream.range(0, n).parallel().forEach(i -> {
8          // _a[] means to load thru read-only cache
9          b[i] = _a[i] * (1000 / (n - 7));
10         c[i] = b[i];
11       });
12     } else {
13       // the loop without read-only cache
14       ...
15     }
16   } else { ... } // the original parallel streams code
17                  // with a virtual method call
18 }
```

transfer rate through PCI-Express Gen 3. To alleviate this overhead, our JIT compiler performs two types of optimizations for reducing unnecessary data transfers.

One optimization is to try and transfer part of an array rather than the entire array. Specifically, a read set of an array reference on the right hand side or a write set of an array reference on the left hand side can be transferred. The current CUDA runtime supports transfers of contiguous regions, and our compiler enables this optimization when the read set or the write set is a contiguous region. The exemple array access targeted by our optimization has the form, `x[i+b]`, where `i` is a loop induction variable and `b` is loop invariant. This optimization can reduce the data transfer overhead compared with the previous work [30] that transfers the whole array.

The other is to eliminate data transfers from the GPU to the CPU if an array is read-only on the GPU. If our JIT compiler detects a lexical read-only array in lambda expressions with parallel streams, the JIT compiler lets a runtime helper know a data transfer for this array may be eliminated. At runtime, if the runtime helper recognizes that the lexical read-only array does not alias with any of the other arrays to be written, it eliminates the data transfer regarding the read-only array from the GPU to CPU. Otherwise, the transfer is performed. Note that this elimination of data transfers for read-only arrays is supported in previous implementations [17], [30] for C, C++, and X10.

Listing 4. Pseudo code for versioned loops to eliminate exception checks

```
1  void foo(float[] a, float[] b, float[] c, int n, Par p) {
2    if (isNonOverridden(t.bar)) {
3      if (isUseRoc(a,true,false) && isUseRoc(b,false,true) &&
4          isUseRoc(c,false,true)) {
5        // pre-exception checks
6        if (a!=null && b!=null && c!=null && // nullpointer
7            0 <= a.length && a.length < n && // array bound
8            0 <= b.length && b.length < n &&
9            0 <= c.length && c.length < n &&
10           (n-7) != 0} // division by zero
11         // optimized GPU parallel loop w/o exception checks
12         IntStream.range(0, n).parallel().forEach(i -> {
13           // _a[] means to load thru read-only cache
14           b[i] = _a[i] * (1000 / (n-7));// no exception check
15           c[i] = b[i]; // no exception check
16         });
17       } else {
18         // original parallel streams code executed on CPU
19           ... // with exception check
20       }
21     } else { ... } // loop without read-only cache
22   } else { ... } // original parallel streams code on CPU
23 }
```

D. Eliminating exception checks in a parallel loop

Our JIT compiler currently supports three exceptions during GPU execution, i.e., `NullPointerException`, `ArithmeticException` (division by zero), and `ArrayIndexOutOfBoundsException`, in lambda expressions with parallel streams. These possible exceptions pose two performance issues. One is increasing the path length of instruction in a parallel loop by inserting exception checks. The other is the introduction of control dependences between every pair of exceptions. To alleviate these issues, it is important to reduce the number of exception checks in lambda expressions with parallel streams. Our JIT compiler creates an optimized safe region that cannot throw any exception by using loop versioning [2].

To eliminate checks for `NullPointerException` or `ArithmeticException` in a parallel loop, the compiler generates checks with a loop invariant variable to ensure an array reference is `non-null` or a divisor is not zero. For an optimized parallel loop, these checks are speculatively executed on the CPU. Moreover, to eliminate checks for `ArrayIndexOutOfBoundsException` in a parallel loop, the compiler generates checks with a loop induction variable to ensure that the array index expression is within the array length. Similarly, these checks are speculatively executed on the CPU. Our JIT compiler supports array accesses of the form `x[a*i+b]`, where `i` is a loop induction variable, and `a` and `b` are loop invariant. This loop versioning can also be applied to a `for` loop in a lambda expression. In this case, checks are speculatively executed before executing the `for` loop on the GPU.

Listing 4 shows the pseudocode after applying loop versioning to the running example in Listing 1. `Pre-exception checks` code are used to check the conditions of three kinds of exceptions before executing the optimized parallel loop. One is to check whether an array reference is `non-null` in line 6. The other is to check whether an index is within a range of an array length in lines 7-9. The last one is to check whether a divisor is non-zero in line 10. If any of these conditions is not satisfied, the original parallel streams code is executed on the CPU. Since there may be several unoptimized loops after applying the optimizations, these loops can be grouped into one or a few loops to reduce the size of the generated code. For example, an unoptimized loop executed on a CPU can handle all of the unoptimized loops in the `else` clauses.

Listing 5. NVVM IR corresponds to lines 14 and 15 in Listing 4

```
1  define void @foo(i8* %p0, i8* %p1, i8* %p2, ...) {
2    // eliminated any exception checks in this method
3    // by the optimization in Section V-D
4
5    %p0.addr = alloca i8*, align 8
6    %p0.hdrptr = getelementptr inbounds i8* %p0, i32 112
7    store i8* %p0.hdrptr i8** %p0.addr, align 8
8    %p1.addr = alloca i8*, align 8
9    %p1.hdrptr = getelementptr inbounds i8* %p1, i32 112
10   store i8* %p2.hdrptr, i8** %p1.addr, align %78
11   %p2.addr = alloca i8*, align 8
12   %p2.hdrptr = getelementptr inbounds i8* %p2, i32 112
13   store i8* %p2.hdrptr, i8** %p2.addr, align 8
14   ...
15   %7 = load i8** %p1.addr, align 8
16   %8 = load i32* %a5.addr, align 4 // i = ...
17   %9 = sext i32 %8 to i64
18   %10 = mul i64 %9, 4
19   %11 = add i64 %10, 16
20   %12 = getelementptr inbounds i8* %7, i64 %11// &b[i]
21
22   %13 = load i32* %p3.addr, align 4 // n = ...
23   %14 = add i32 %13, -7
24   %15 = sdiv i32 1000, %14
25   %16 = sitofp i32 %15 to float // (1000 / (n-7))
26
27   %17 = load i8** %p0.addr, align 8
28   %18 = getelementptr inbounds i8* %17, i64 %11 // &a[i]
29   %19 = bitcast i8* %18 to float addrspace(1)*
30   // ... = a[i] thru read-only cache
31   %20 = tail call float @llvm.nvvm.ldg.global.f.f32.p1f32
32     (float addrspace(1)* %19), ...
33
34   %21 = fmul float %16, %20
35   %22 = bitcast i8* %12 to float *
36   store float %21, float * %22, align 4  // b[i] = ...
37   %23 = load i8** %p2.addr, align 8
38   %24 = getelementptr inbounds i8* %23, i64 %11 // &c[i]
39   %25 = bitcast i8* %24 to float *
40   store float %21, float * %25, align 4 // c[i] = ...
41   ...
       // for alignment optimization in Section V-A
       // for read-only cache optimization in Section V-B
```

The same approach is used to eliminate checks for `NullPointerException` and `ArrayIndexOutOfBoundsException` in a loop [22]. This paper also applies loop versioning to `ArithmeticException`. Unlike the previous work [22], our approach, which combines native exception checks described in Section IV-A and optimistic loop versioning, can ensure precise exception semantics even if loop versioning cannot be applied. Our experiments in Section VI also reveal that eliminating exception checks encourages optimizations, such as loop unrolling, in a low-level compiler for GPUs.

Listings 5 shows the NVVM IR corresponding to a GPU kernel on lines 14 and 15 in Listings 4. In the NVVM IR, there is no conditional branch since loop versioning moves all of the exception checks out of the parallel loop and they are executed on the CPU. Instructions to add an offset to a

register in lines 6, 9, 12, and 19 adjust the starting address of the Java array and are generated by the optimization in Section V-A. The load instruction in lines 31-32 obtains a float element from the read-only cache, which is generated by the optimization in Section V-B.

VI. Performance Evaluation

This section presents the results of an experimental evaluation of our JIT compiler on an IBM POWER8 and NVIDIA Tesla GPU platform with the Ubuntu 14.10 operating system and CUDA 5.5. The platform has two 10-core IBM POWER8 CPUs (3.69GHz with 256GB of RAM). Each core is capable of running eight SMT threads, resulting in 160 threads per platform. The NVIDIA K40m GPU with 2880 CUDA cores operating at 876MHz with 12GB of global memory is connected to the POWER8 by using PCI-Express Gen 3. The error-correcting code (ECC) feature was turned off in the experiment. The L1 cache of the GPU was only used for local working sets such as register spill areas.

The eight benchmarks shown in Table II were used in the experiments. Each benchmark was tested in a parallel Java version and a sequential Java version. The parallel Java version employed parallel streams with lambda expressions to mark parallel loops which could be run either on the CPU using the Java fork/join framework or on GPU devices. The appendix shows the code style that illustrates array access patterns, based on the lambda parameter, for each of the benchmark programs. In the sequential Java version, the parallel streams were replaced by sequential Java for-loops.

The parallel Java version was executed on two different configurations:

- **160 worker threads :** Executed in the Java fork/join framework on the Java Virtual Machine (JVM) running on the CPUs; the number of threads was not specified explicitly, which means that the maximum available worker threads were used by default (160 threads on this platform).
- **GPU :** Executed on GPUs using the proposed code generation and runtime (1024 CUDA threads per block were assigned).

Performance was measured in terms of elapsed nanoseconds from the start of one or two parallel streams to the completion of all iterations of those loops in each benchmark program. (see the Appendix for details.) The Java system call *System.nanoTime()* was used. This measurement included the overhead of fork/join for parallel Java. For GPU execution, it included any overhead from CUDA Driver API calls such as GPU memory allocation and data transfer between the host and the GPU. For each configuration, each benchmark was executed 30 times in a single JVM invocation, and the average execution time of the last ten executions was reported as a *steady-state* execution time.

optimizations	instructions per iteration of inner most loop	execution time of the GPU kernel
`ALL` without `LV`	31	36.6ms
`ALL` without unrolling	18	23.8ms
`ALL`	17	20.1ms

Table III
Instructions and execution time of **MM**

A. Performance improvements over sequential Java

Figure 3 shows a maximum speedup numbers with 95% confidence interval error bars relative to the sequential Java version. The results show a speedup of up to 81.4 × for the 160 worker threads (fork/join) case for **SpMM**, and a maximum speedup of 2067.7 on the GPU for **Series**. However, **SpMM** and **Gesummv** both demonstrated higher speedups on the CPU than the GPU due to large data transfer overheads. These overheads are expected to be lower on future hardware platforms due to enhancements such as NVlink and unified memory. Overall, the results show a geometric mean performance improvement of 127.9 for GPU execution relative to the sequential Java version, and of 3.3 (geometric mean) relative to the parallel CPU version on 160 hardware threads (fork/join).

B. Breakdown for GPU Optimizations

This section analyzes the GPU execution to show the impact of the four optimizations described in Section V. In the following, loop versioning is referred to as `LV`, data transfer optimization is referred to as `DT`, buffer alignment optimization is referred to as `ALIGN`, and read-only cache array utilization is referred to as `ROC`. `BASE` is a baseline and `ALL` denotes `BASE` plus all optimizations (=`BASE` + `LV` + `DT` + `ALIGN` + `ROC`). Figure 4 illustrates the impact of each optimization. Each bar consists of the host-to-device data transfer time (H2D), kernel execution time (Kernel), and device-to-host time (D2H). These bars are normalized to the sum of the H2D, Kernel, and D2H times [1] with `ALL` optimizations, which is the right-most bar for each benchmark.

For **BlackScholes**, `ALL` shows a 153.7% performance improvement compared with `BASE`. This benchmark particularly benefits from `DT` since one array is not updated (D2H optimization) and two arrays are written but not read on the GPU (H2D optimization), resulting in a 55.3% DT performance improvement. For **Crypt**, `ALL` shows a 147.0% performance improvement compared with `BASE`. `DT` shortens D2H data transfers by 97.2%. `ALIGN` degrades kernel performance by 6.3% due to non-coalesced array accesses but `ROC` eventually shows a 50.3% kernel performance improvements

[1] We don't include other overheads such as device initialization because these overheads are relatively much smaller than the H2D, Kernel, and D2H times.

Benchmark	Summary	Data Size	Data Type
Blackscholes	Financial application which calculates the price of European put and call options	4,194,304 virtual options	double
Crypt	Cryptographic application from the Java Grande Benchmarks [18]	Size C with N= 50,000,000	byte
SpMM	Sparse matrix multiplication from the Java Grande Benchmarks [18]	Size C with N = 500,000	double
Series	Series from the Java Grande Benchmarks [18]	Size C with N = 1,000,000	double
MRIQ	Three-dimensional medical benchmark from Parboil [31], ported to Java	large size(64×64×64)	float
MM	A standard dense matrix multiplication: $C = A.B$	1,024×1,024	double
Gemm	Matrix multiplication: $C = \alpha.A.B + \beta.C$ from PolyBench [32], ported to Java	1,024×1,024	int
Gesummv	Scalar, Vector and Matrix Multiplication from PolyBench [32], ported to Java	4,096×4,096	int

Table II
DETAILS ON THE BENCHMARKS USED TO EVALUATE THE PROPOSED JIT COMPILER

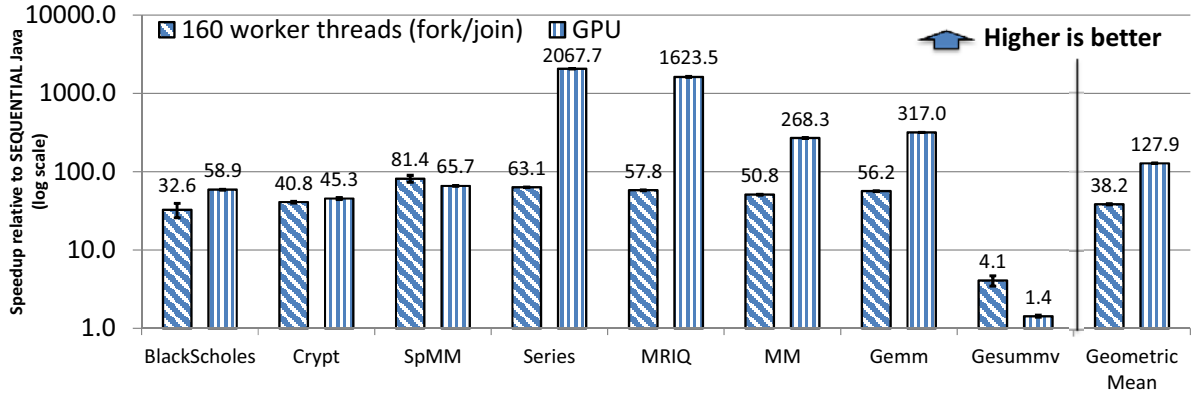

Figure 3. Performance improvements over sequential Java on IBM POWER8 + NVIDIA Tesla GPU

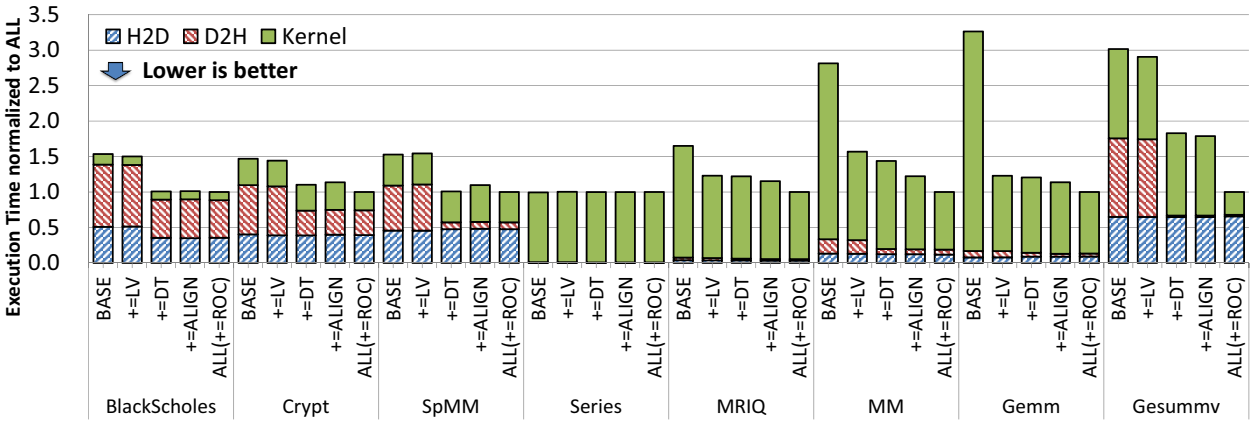

Figure 4. Performance breakdown for GPU execution. Each bar is normalized to the right-most bar (ALL) for each benchmark.

	Blacksholes	Crypt	SpMM	Series	MRIQ	MM	Gemm	Gesummv	Geometric Mean
Expansion ratio of GPU binary size	0.93	0.88	1.00	0.96	1.30	1.47	1.30	1.10	1.10

Table IV
EFFECT OF APPLYING LOOP VERSIONING (LOOP VERSIONING INCREASES CODE SIZE IF RATIO IS MORE THAN 1.00)

	Blacksholes	Crypt	SpMM	Series	MRIQ	MM	Gemm	Gesummv	Geometric Mean
Our Kernel time	2.0 msec	13.4 msec	4.6 msec	415.2 msec	44.6 msec	19.0 msec	17.6 msec	6.4 msec	-
Aparapi Kernel time	2.3 msec	19.9 msec	2.0 msec	Fails	53.6 msec	28.3 msec	20.2 msec	11.3 msec	-
CUDA Kernel time	1.7 msec	20.2 msec	3.4 msec	380.6 msec	5.1 msec	8.5 msec	20.9 msec	22.2 msec	-
speedup over Aparapi	1.15x	1.49x	0.43x	-	1.20x	1.48x	1.14x	1.77x	1.15x
speedup over CUDA	0.85x	1.51x	0.74x	0.92x	0.11x	0.45x	1.19x	3.47x	0.81x

Table V
PERFORMANCE IMPROVEMENTS OVER APARAPI AND CUDA CODE

over +=ALIGN. For **SpMM**, ALL shows a 152.8% performance improvement compared with BASE. A 581% D2H data transfer speed up is achieved by DT. Like **Crypt**, ALIGN degrades kernel performance by 20.0% due to indirect array accesses but this degradation is canceled by ROC. For **Series**, GPU performance is bounded by the kernel computation but the kernel does not benefit from the optimizations due to there being fewer operations on arrays. Performance for **MRIQ** is also bounded by the kernel computation. LV, ALIGN, and ROC improve kernel performance by 27.6%, 2.5%, and 6.6%, respectively, and thus ALL shows a 164.8% performance improvement over BASE. Similarly, for **MM**, LV, ALIGN, and ROC improve kernel performance by 98.7%, 20.3%, and 26.7%, respectively; This is in addition to the 161.1% H2D performance improvement by DT (i.e., a 280% performance improvement in total). **Gemm** also benefits from the optimizations. Performance improvements numbers include 192.2% for LV, 5.1% for ALIGN, 15.9% for ROC, and the overall performance improvement had by ALL is 326.4%. For **Gesummv**, DT eliminates a significant number of redundant D2H data transfers (i.e., a 4500% D2H performance improvement), and ROC shows a 349% kernel performance improvement, resulting in a 301.4% performance improvement for ALL compared with BASE.

We also investigated how LV contributes to performance improvements. Here, we will focus on **MM**. Table III shows the instructions per iteration and kernel execution time of the three optimizations. Since ALL without LV includes exception checks in the loop, its execution is slow. ALL is the fastest, since it has no exception checks. The PTX2binary module applied loop unrolling to the generated instructions once in the inner most loop. This loop unrolling yielded a 15.5% performance improvement, compared with the version disabling loop rolling. In general, loop versioning can be applied when the original loop body is small. Thus, LV is effective at exploiting additional optimization opportunities by reducing the number of instructions in a loop.

We also examined the effect of applying loop versioning in terms of the expansion ratio of the binary size of program on the GPU. Table IV compares the GPU binary sizes with and without applying loop versioning. While the binary sizes of some programs are increased by loop versioning, the binary size of other programs are reduced by it.

For those applications that have no inner loop such as **Blackscholes**, **Crypt**, and **Series** (see Listing. 6 in the appendix for more details), most of the exception checking code are moved out of the parallel loop in their CPU binaries and their GPU binaries do not contain the original parallel loop as shown in Listing. 4. That is why the ratio of code expansion is less than 1.00. In contrast, the ratio may exceed 1.00 when an application has an inner loop (e.g. **MRIQ**, **MM**, **Gemm**, and **Gesummv**) since loop versioning generates a dual version of code consisting of the original code and the optimized code for each inner loop in their GPU binary. Note that loop versioning is not applied to **SpMM**.

C. Comparison with the state-of-the-art approach

Aparapi is a state-of-the-art approach which uses Java 8 lambda expressions to generate GPU code by compiling Java bytecode to OpenCL code (see Section II). The experiment hence assessed the performance of the lambda branch of Aparapi[2] and our JIT compiler. It used exactly the same lambda expression, but for Aparapi, a parallel region was specified by `Device.firstGPU(length, i ->{...});` instead of the standard Java 8 `IntStream.range(start, end).parallel().forEach(i ->{...});`.

Unfortunately, OpenCL is not supported in the POWER8 platform. Thus, the Aparapi version was executed on an Intel Core i7-4820K CPU at 3.70GHz and K40c GPU. The K40c GPU is exactly the same as K40m but has a cooling system. We used the Oracle Java SE Development Kit 8 (1.8.0_40) on the Ubuntu 14.04 operating system. For fair comparison, we used simply the elapsed time for the kernel execution because the data transfer time depends on the host OS and CPUs.

Table V shows the execution times and performance improvement of the JIT compiler relative to Aparapi on 7 benchmarks.[3] Each benchmark was executed 30 times in a single JVM invocation and we took the average kernel execution time of the last ten executions as the *steady-state* kernel execution time. Kernel execution times were measured as follows:

- For the JIT compiler, the kernel time was computed by using the C `gettimeofday` function to measure the elapsed time for the CUDA `cuLaunchKernel` and `cudaDeviceSynchronize` APIs.
- For Aparapi, the kernel time was measured by using the OpenCL `clGetEventProfilingInfo` function to get information on when `clEnqueueNDRangeKernel` commands actually began execution and when they completed execution.

Table V show that the JIT compiler generally faster than Aparapi. For **SpMM**, the performance difference is caused by a loop unrolling factor for an important loop than ran until PTX was generated. Aparapi itself does not aggresively optimize kernel and the optimization is delegated to the OpenCL compiler, which narrows the compiler's scope of optimizations due to the lack of information for the whole program. On the other hand, our JIT compiler could perform several optimizations on IRs from the Java bytecode and applied several of the NVIDIA GPU-aware optimizations discussed in Section V. In addition, Aparapi does not optimize data transfers as it would incur large data transfer overheads.

[2]To the best of our knowledge, Aparapi-1.0.0 in github has no Java 8 lambda support.

[3]Aparapi failed to compile JGF-Series.

D. Comparison with hand-written code

This section compares the hand-written CUDA programs and GPU code generated by our JIT compiler. We executed the versions of **BlackScholes** and **MM** that are in the CUDA SDK samples [24], **Gemm** and **Gesummv** from PolyBench [32], **MRIQ** from Parboil [31], and **Crypt**, **SpMM**, and **Series** implemented in [36].

We used the default optimization option level (`-O3`) of `nvcc` to compile these CUDA programs. Table V shows the execution times and the relative performance of the JIT compiler in relation to the hand-written code. Each benchmark is executed 30 times and we computed the average kernel execution time of the last ten executions.

In the case of **SpMM**, the JIT compiler was slower than the hand-written CUDA code since it could not eliminate exception checks and the CUDA code does not perform exception checks. The CUDA version of **BlackScholes** chooses 128 CUDA threads per block while the JIT compiler always uses 1024 CUDA threads per block. On **Crypt**, **Gemm**, and **Gesummv**, the JIT compiler outperformed CUDA code, since the optimization of utilizing the read-only cache array contributed to a performance improvement. However, this was not the case with **MRIQ** and **MM**, due to the lack of GPU-aware optimizations including –use-fast-math for float values and the lack of the use of shared memory in our JIT compiler. The code generated by the JIT compiler performed additional operations for preserving Java's exception semantics while the hand-written CUDA code did not perform any exception checking operations on the GPUs.

VII. Related Work

A. GPU code generation from JVM-compatible languages

GPU code generation is supported by several JVM-compatible language compilation systems.

Automatic parallelization of Java programs for GPUs would be an ideal goal. However, it has been widely recognized that auto-parallelization faces obstacles such as aliasing. A previous study [22] addressed these issues but some restrictions still remain. For automatic parallelization, an appropriate device selection (e.g. CPU or GPU) is also an important issue even if we recognize that the given loop can be parallelized. There are a few studies [13], [22] including our work that address this issue.

Many previous studies support *explicit parallel programming* by programmers. JCUDA [36] allows programmers to write Java codes that call user-written CUDA kernels with a special interface. The JCUDA compiler generates the JNI glue code between the JVM and CUDA runtime by using this interface. Some studies provide Java-based explicit GPU parallel programming models with GPU-specific class libraries. JaBEE [37], RootBeer [33], and Aparapi [1] compile Java bytecode to either CUDA or OpenCL code by including a code region within a method declared inside a specific class / interface (e.g. run() method of the `Kernel` class / interface). While these approaches provide impressive support for executing Java programs on GPUs, the programming model lacks portability and significantly decreases programmability relative to the Java 8 parallel steam APIs.

To increase programmability, other studies provide higher-level abstraction of GPU programming. Hadoop-CL [9] is built on top of Aparapi and adds OpenCL generation into the Hadoop system. Lime [5] is a JVM compatible language with Java extensions that express map/reduce operations. HJ-OpenCL [11], [12] generates OpenCL from Habanero-Java language, which provides high-level language constructs such as parallel loop (`forall`), barrier synchronization (`next`), and high-level multi-dimensional array (`ArrayView`). One study [6] provides a high level array programming model built on top of the Java 8 parallel streams API.

Overall, these approaches rely on external compilers or class libraries and some of them have no support for Java exception semantics. Our approach, however, supports the Java 8 language.

B. Kernel / Communication Optimizations

Lime [5] tries to use several types of OpenCL memory. It does so by utilizing Lime's strongly typed system without sophisticated data dependence analysis or alias analysis. However, their approach does not support pure Java programs.

Pluto [3] and PPCG [35] are polyhedral compilation frameworks that particularly improve locality through loop tiling for imperfectly nested loops in C programs with GPU's shared memory utilization. These approaches generally assume that there is no aliasing in a region enclosed by a specific directive.

Some studies try to minimize data transfer overheads with dynamic analysis. X10CUDA+AMM [30] maintains coherence between the host memory and GPU memory at the granularity of a whole array and tries to eliminate redundant data transfers between CPUs and GPUs. CGCM [17] and DyManD [16] inspect user-written CUDA kernels and C/C++ host code and insert run-time library calls which track data usage throughout program execution at the granularity of an *allocation unit* (e.g. a memory region allocated by malloc). Unlike these approaches, ours supports partial array copying with static analysis.

C. Precise Exception Semantics

To the best of knowledge, only a few approaches maintain Java's exception semantics on GPUs.

One previous study [22] checks for `NullPointerException` and `ArrayIndexOutOfBoundsException` before executing GPU loop to ensure no exception occurs during GPU execution [2]. However, it cannot handle

`ArithmeticException`; nor can it deal with a case where an array subscript is non-affine or not loop invariant (e.g. indirect access in **SpMM**).

RootBeer [33] runs checking for `NullPointer Exception` and `ArrayIndexOutOfBoundsException` during GPU execution. This approach, however, may add non-trivial overheads in a case where exception checking code can be moved out of the GPU loop as we discussed in Section VI.

HJ-OpenCL [11], [12] offers two exclusive modes to maintain Java's exception semantics on GPUs. One is that programmers can specify conditions which ensure no exception will be thrown during GPU execution with the `safe` language construct; thereby, the construct is used for manual loop versioning and exception checking is done by CPUs before GPU execution, like in [22]. Another way is speculative exception checking. HJ-OpenCL's runtime speculatively executes OpenCL kernels without exception checking on GPUs in parallel while executing automatically generated special code intended for exception checking on CPUs. Unlike HJ-OpenCL, our approach uses both static [11], [22] and dynamic [12], [33] approaches in order to move exception checking code out of GPU loop to avoid dynamic overheads.

VIII. Conclusion

This paper described the first JIT compiler that compiles and optimizes parallel streams APIs for GPUs implemented on top of a production Java 8 runtime system. The JIT compiler approach facilitates ease of programming, safety, and platform portability on a GPU. It supports precise exception and virtual method calls in lambda expressions with parallel streams. It also automatically optimizes lambda expressions by aligning the body of the Java array on the GPU with the memory transaction boundary, by utilizing the read-only cache for array accesses, and by eliminating unnecessary data transfers between the host and the GPU. In addition, it also performs loop versioning to reduce the number of exception checks in GPU code. Most of these optimizations are already integrated into the IBM SDK Java Technology Edition, Version 8. Our JIT compiler outperforms sequential and parallel executions of the original Java 8 programs by 127.9 × and 3.3 × geometric mean speedups, showed a 1.15 × geometric mean speed up over the state-of-the-art approach (Aparapi), and performed comparably to hand-written CUDA programs.

Appendix

Listing 6 shows the code style that illustrates array access patterns, based on the lambda parameter, for each of the benchmark programs.

Listing 6. Code style of benchmark programs

```
//BlackScholes
IntStream.range(0, N).parallel().forEach(i -> {
  ...
  c[i] = ...;
  p[i] = ...;
});


//Crypt
IntStream.range(0, N / 8).parallel().forEach(idx -> {
  i = idx * 8;
  ... = t1[i]; ...; ... = t1[i+7];
  ...
  t2[i] = ...; ...; t2[i+7] = ...;
});


//SpMM
IntStream.range(0, N).parallel().forEach(id -> {
  int rbegin = row[id], rend = row[id+1];
  for (int i = 0; i < rend - rbegin; i++) {
    for (int j = 0; j < inter; j++) {
      s += ...;
    }
  }
  y[id] = s;
});


//Series
IntStream.range(1, 2*N).parallel().forEach(i -> {
  ...
  testArray[i]= ...;
});


//MRIQ
IntStream.range(0, K).parallel().forEach(i -> {
  ... = phiR[i] ... phiI[i];
  phiMag[i] = ...
});
IntStream.range(0, X).parallel().forEach(i -> {
  for (int j = 0; j < K; j++) {
    ... = (kx[j] * x[i] + ky[j] * y[i] + kz[j] * z[i]);
    ... = phiMag[j];
  }
  Qr[i] = ...;
  Qi[i] = ...;
});


//MM
IntStream.range(0,N*N).parallel().forEach(idx - >{
  int i = idx / N, j = idx % N;
  for (int k = 0; k < ROWS; k++) {
    s += b[i*N+k] * c[k*N+j];
  }
  a[idx] = s;
});

//Gemm
IntStream.range(0, N*N).parallel().forEach(idx -> {
  int i = idx / N, j = idx % ROWS;
  C[idx] *= beta;
  for (int k = 0; k < ROWS; k++) {
    C[idx] += A[i*N+k] * B[k*N+j] ...;
  }
});


//Gesummv
IntStream.range(0, N).parallel().forEach(i -> {
  t[i] = 0; y[i] = 0;
  for (int j = 0; j < ROWS; j++) {
    t[i] = A[i*N+j] * ... + t[i];
    y[i] = B[i*N+j] * ... + y[i];
  }
  y[i] = t[i] ... + y[i] ...;
});
```

Acknowledgments

Part of this research is supported by the IBM Centre for Advanced Studies. We thank Marcel Mitran for his encouragement and support in pursuing the parallel streams API and lambda approach, and thank Jimmy Kwa for his extensive contribution to the implementation. We also thank Max Grossman for providing us the CUDA programs used as part of the evaluation. Finally, we would like to thank the anonymous reviewers for their helpful comments and suggestions.

References

[1] Aparapi. API for Data Parallel Java. http://code.google.com/p/aparapi/.

[2] P. V. Artigas, M. Gupta, S. P. Midkiff, and J. E. Moreira. Automatic Loop Transformations and Parallelization for Java. ICS '00, pages 1–10, 2000.

[3] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs. ICS '08, pages 225–234, 2008.

[4] B. Calder and D. Grunwald. Reducing Indirect Function Call Overhead in C++ Programs. POPL '94, pages 397–408, 1994.

[5] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink. Compiling a high-level language for GPUs: (via language support for architectures and compilers). PLDI '12, pages 1–12, 2012.

[6] J. J. Fumero, M. Steuwer, and C. Dubach. A Composable Array Function Interface for Heterogeneous Computing in Java. ARRAY '14, pages 44:44–44:49, 2014.

[7] J. Gosling, B. Joy, G. L. Steele, G. Bracha, and A. Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 2014.

[8] N. Grcevski, A. Kielstra, K. Stoodley, M. Stoodley, and V. Sundaresan. Java Just-in-time Compiler and Virtual Machine Improvements for Server and Middleware Applications. VM '04, pages 151–162, 2004.

[9] M. Grossman, M. Breternitz, and V. Sarkar. HadoopCL: MapReduce on Distributed Heterogeneous Platforms Through Seamless Integration of Hadoop and OpenCL. IPDPSW '13, pages 1918–1927, 2013.

[10] M. Grossman, S. Imam, and S. Vivek. Hjlib-cl: Reducing the gap between the jvm and accelerators. PPPJ '15. 2015.

[11] A. Hayashi, M. Grossman, J. Zhao, J. Shirako, and V. Sarkar. Accelerating Habanero-Java Programs with OpenCL Generation. PPPJ '13, pages 124–134, 2013.

[12] A. Hayashi, M. Grossman, J. Zhao, J. Shirako, and V. Sarkar. Speculative execution of parallel programs with precise exception semantics on gpus. LCPC '13, pages 342–356. 2014.

[13] A. Hayashi, K. Ishizaki, G. Koblents, and S. Vivek. Machine-learning-based performance heuristics for runtime cpu/gpu selection. PPPJ '15. 2015.

[14] IBM Corporation. IBM SDK, Java Technology Edition, Version 8. http://www.ibm.com/developerworks/java/jdk/.

[15] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A Study of Devirtualization Techniques for a Java Just-In-Time Compiler. OOPSLA '00, pages 294–310, 2000.

[16] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August. Dynamically managed data for CPU-GPU architectures. CGO '12, 2012.

[17] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August. Automatic CPU-GPU Communication Management and Optimization. PLDI '11, pages 142–151, 2011.

[18] JGF. The Java Grande Forum benchmark suite. http://www.epcc.ed.ac.uk/javagrande/javag.html.

[19] KHRONOS GROUP. OpenCL 2.1 Provisional API Specification, Version 2.1, 2013. https://www.khronos.org/registry/cl/.

[20] Khronos Group. SPIR SPECIFICATION 1.2, 2014. https://www.khronos.org/registry/spir/specs/spir_spec-1.2.pdf.

[21] K. Knobe and V. Sarkar. Array SSA form and its use in Parallelization. POPL '98, 1998.

[22] A. Leung, O. Lhoták, and G. Lashari. Automatic parallelization for graphics processing units. PPPJ '09, pages 91–100, 2009.

[23] P. Micikevicius. Performance optimization. http://on-demand.gputechconf.com/gtc/2013/presentations/S3466-Programming-Guidelines-GPU-Architecture.pdf.

[24] NVIDIA Corporation. CUDA Samples, 2013. http://docs.nvidia.com/cuda/cuda-samples/.

[25] NVIDIA Corporation. CUDA C PROGRAMMING GUIDE 7.0, 2014. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[26] NVIDIA Corporation. NVVM IR SPECIFICATION 1.1, 2014. http://docs.nvidia.com/cuda/pdf/NVVM_IR_Specification.pdf.

[27] NVIDIA Corporation. PARALLEL THREAD EXECUTION ISA v4.1, 2014. http://docs.nvidia.com/cuda/pdf/ptx_isa_4.1.pdf.

[28] OpenACC forum. The OpenACC Application Programming Interface, Version 2.0, 2013. http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf.

[29] Oracle Corporation. The java tutorial. https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html.

[30] S. Pai, R. Govindarajan, and M. J. Thazhuthaveetil. Fast and Efficient Automatic Memory Management for GPUs Using Compiler-assisted Runtime Coherence Scheme. PACT '12, pages 33–42, 2012.

[31] Parboil. Parboil benchmarks. http://impact.crhc.illinois.edu/parboil.aspx.

[32] PolyBench. The polyhedral benchmark suite. http://www.cse.ohio-state.edu/~pouchet/software/polybench.

[33] P. Pratt-Szeliga, J. Fawcett, and R. Welch. Rootbeer: Seamlessly Using GPUs from Java. HPCC-ICESS '12, pages 375–380, 2012.

[34] Sumatra. Project Sumatra. http://openjdk.java.net/projects/sumatra/.

[35] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, 2013.

[36] Y. Yan, M. Grossman, and V. Sarkar. JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA. Euro-Par '09, pages 887–899, 2009.

[37] W. Zaremba, Y. Lin, and V. Grover. JaBEE: Framework for Object-oriented Java Bytecode Compilation and Execution on Graphics Processor Units. GPGPU-5, pages 74–83, 2012.