Using Compiler Snippets to Exploit Parallelism on Heterogeneous Hardware

[Link to publication record in Manchester Research Explorer](#)

Citation for published version (APA):

Fumero Alfonso, J., & Kotselidis, C-E. (2018). *Using Compiler Snippets to Exploit Parallelism on Heterogeneous Hardware: A Java Reduction Case Study*. Paper presented at VMIL 2018 Virtual Machines and Language Implementations, Boston, Massachusetts, United States.

Citing this paper

Please note that where the full-text provided on Manchester Research Explorer is the Author Accepted Manuscript or Proof version this may differ from the final Published version. If citing, it is advised that you check and use the publisher's definitive version.

General rights

Copyright and moral rights for the publications made accessible in the Research Explorer are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Takedown policy

If you believe that this document breaches copyright please refer to the University of Manchester's Takedown Procedures [<http://man.ac.uk/04Y6Bo>] or contact uml.scholarlycommunications@manchester.ac.uk providing relevant details, so we can investigate your claim.

Using Compiler Snippets to Exploit Parallelism on Heterogeneous Hardware: A Java Reduction Case Study

Juan Fumero

Advanced Processor Technologies Group
The University of Manchester
Manchester, M13 9PL, United Kingdom
juan.fumero@manchester.ac.uk

Christos Kotselidis

Advanced Processor Technologies Group
The University of Manchester
Manchester, M13 9PL, United Kingdom
christos.kotselidis@manchester.ac.uk

Abstract

Parallel skeletons are essential structured design patterns for efficient heterogeneous and parallel programming. They allow programmers to express common algorithms in such a way that it is much easier to read, maintain, debug and implement for different parallel programming models and parallel architectures. Reductions are one of the most common parallel skeletons. Many programming frameworks have been proposed for accelerating reduction operations on heterogeneous hardware. However, for the Java programming language, little work has been done for automatically compiling and exploiting reductions in Java applications on GPUs.

In this paper we present our work in progress in utilizing compiler snippets to express parallelism on heterogeneous hardware. In detail, we demonstrate the usage of Graal's snippets, in the context of the Tornado compiler, to express a set of Java reduction operations for GPU acceleration. The snippets are expressed in pure Java with OpenCL semantics, simplifying the JIT compiler optimizations and code generation. We showcase that with our technique we are able to execute a predefined set of reductions on GPUs within 85% of the performance of the native code and reach up to 20x over the Java sequential execution.

CCS Concepts • Software and its engineering → Patterns; Just-in-time compilers; Source code generation;

Keywords GPGPUs, JIT Compilation, Reductions

ACM Reference Format:

Juan Fumero and Christos Kotselidis. 2018. Using Compiler Snippets to Exploit Parallelism on Heterogeneous Hardware: A Java

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

VMIL '18, November 4, 2018, Boston, MA, USA

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-6071-5/18/11...\$15.00

<https://doi.org/10.1145/3281287.3281292>

Reduction Case Study. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '18), November 4, 2018, Boston, MA, USA*. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3281287.3281292>

1 Introduction

Parallel programming skeletons such as map-reduce [8] and fork-join [17] have become essential tools for programmers to achieve higher performance of their applications, with ease in programmability. In particular, the map-reduce paradigm, since its conception, has been adopted by many applications that span from Big Data frameworks to desktop computing in various programming languages [21, 28, 32]. In addition, a number of such parallel skeletons have been combined to enable new usages as in the case of MR4J [3] that enables map-reduce operations in Java by employing the fork-join framework to achieve parallelism.

The introduction of heterogeneous hardware resources, such as GPUs and FPGAs into mainstream computing, creates new opportunities to increase the performance of such parallel skeletons. In the context of programming languages that have been designed specifically for heterogeneous programming like OpenCL [19], significant work has been done to implement high-performance reductions on GPUs leveraging the underlying architecture [24, 25]. However, the Java programming language, which is the backbone of Big Data frameworks (e.g. Hadoop [30], Spark [33], and Flink [4]), lacks implementations of reduction operations mainly due to the fact that reduce operations can not be expressed, and hence optimized, inside the language itself. Consequently, the omission of this feature limits not only the application of map/reduce operations on desktop configurations, but also in Big Data processing in large scale deployments as they execute on top of Java Virtual Machines (JVMs).

In this paper we present our work in progress towards supporting Java reductions on heterogeneous hardware. To achieve that, we leverage the Tornado framework [7, 15] that enables Java execution on heterogeneous hardware. In addition, we employ the Graal compiler [9] and its snippets [22] to enable the automatic generation of reduce-operation at runtime. We showcase that the capabilities of snippets can

extend beyond node replacements and prebuilt Intermediate Representation (IR) graphs' introduction inside a method's IR, and can be used to express parallel skeletons during the compilation process completely transparently to the users. Finally, to enable the introduced GPU-based Java reductions, programmers need only to add one annotation to their code. In detail, this paper:

- Demonstrates how OpenCL implements reductions and explains the challenges in implementing them in a managed programming language like Java.
- Introduces a technique for expressing parallelism by utilizing compiler snippets.
- Showcases that with the introduced technique, we are able to express a pre-defined set of Java reduction operations and execute them on GPUs via OpenCL.
- Evaluates the performance of our proposed solution against hand-tuned OpenCL C code and a sequential Java implementation. We showcase that our approach achieves, in average, 85% the performance of the native code and executes up to 20x faster than the sequential Java implementation.

2 Background

This section briefly explains the GPU architecture, the OpenCL programming model, and how to implement efficient reductions on GPUs.

2.1 Reductions

Reductions are operations that compress an input array into a single element [18]. To illustrate how reductions are implemented in OpenCL, we first show a simple reduction implemented in Java (see Listing 1). The reduce method sums up all elements from an input array and returns the result as a scalar value.

```

1 public float reduce(float[] input) {
2     float result = 0.0f;
3     for (int i = 0; i < input.length; i++) {
4         result += input[i];
5     }
6     return result;
7 }
```

Listing 1. Example of a Java reduction.

2.2 Overview of the GPU Architecture

GPUs can be regarded as general purpose accelerators, initially designed for computer graphics. They contain hundreds of cores grouped into blocks called Stream Multiprocessors (SMs). Each block contains its own set of schedulers (NVIDIA GPUs contain up to 4 thread schedulers) that assign physical GPU cores to input threads. Each core contains a set of functional units (integer and float precision) as well as

special functional units for other math operations, such as square root. Each SM contains its local memory, which in the case of NVIDIA refers to private memory while in the case of OpenCL refers to actual shared memory¹. Only threads running on the same SM can share memory. This is a crucial hardware detail in order to understand how reductions work on GPUs and OpenCL.

Furthermore, each SM also contains a set of registers for keeping private variables, and a space of global memory in which threads can read and write. However, the global memory is much slower than the local memory of the GPU. The number of SMs within a GPU varies depending on the GPU model. The GPU we used for our experiments in this paper (NVIDIA GP100 Pascal [1]) contains 60 SMs with 64 cores each, with a total of 3840 single precision CUDA cores. This type of hardware is ideal for exploiting highly parallel and regular applications by running thousands of threads simultaneously on the GPU.

2.3 Brief Overview of OpenCL

The Open Computing Language (OpenCL) is a standard for heterogeneous programming [19, 26] and is composed of a programming language and a runtime system that facilitates programming and execution on heterogeneous devices (e.g., GPUs, FPGAs, and CPUs).

OpenCL execution on GPUs OpenCL programmers write *compute-kernels* as functions to be executed on the heterogeneous devices. Kernels are implemented using an extension of the C programming language (C with OpenCL modifiers), which are dynamically compiled at runtime by the host (e.g., a CPU) and sent to a target device for execution (e.g., a GPU). Parallelization in OpenCL is implicit by mapping kernels into an N-dimensional index space. This means that OpenCL programmers work with the index space to obtain a single element from the input space. GPU execution follows the SIMT (Single Instruction Multiple Thread) model, a variance of the SIMD (Single Instruction Multiple Data) model, in which a single instruction is executed in parallel by many threads using a different input index from the iteration space. The host program launches the kernel typically with a large number of threads. The target device (e.g., a GPU), receives the threads, partitions them into groups (called *warps* or *wavefronts*), and assigns them to SMs on the GPU.

2.4 Reductions in OpenCL

Figure 1 shows a representation of how to perform reductions on a GPU using OpenCL. The iteration space is divided into groups (called work-groups). In the example shown in Figure 1, there are two groups of eight threads. All threads within the same work-group will perform a full reduction.

¹In the context of this paper, we follow the OpenCL terminology to define shared memory as local memory.

Figure 1. Representation of a reduction on GPUs using OpenCL. Each thread will compute a reduction inside a work-group. The host side will compute the final result by reducing all elements from all the work-groups.

```

1  kernel void reduce(global float* input,
2                      global float* partialSums,
3                      local float* localSums) {
4      int idx = get_global_id(0);
5      uint localIdx = get_local_id(0);
6      uint group_size = get_local_size(0);
7      localSums[localIdx] = input[idx];
8      for (uint stride = group_size / 2;
9           stride > 0; stride /= 2) {
10         barrier(CLK_LOCAL_MEM_FENCE);
11         if (localIdx < stride) {
12             localSums[localIdx]
13             += localSums[localIdx + stride];
14         }
15     }
16     if (localIdx == 0) {
17         partialSums[get_group_id(0)] = localSums[0];
18     }
19 }

```

Listing 2. Reduction in OpenCL

A reduction in OpenCL also divides each work-group into two parts. The algorithm on the GPU will compute the reduction using these two parts. For example, as Figure 1 illustrates, the result from the first iteration in position 0 of the first work-group takes the input elements indexed in positions 0 and 3 (first of the first half with first of the second half). At the end of each iteration, an OpenCL barrier operation is needed in order to guarantee that all threads have written their new values. The process will iterate until reducing just the last two elements within the same work-group. The final reduction occurs on the host side, in which a final reduction across all results from each work-group is performed.

In order to achieve parallelism in reductions with OpenCL, threads have to be organized in such a way that each work-group can compute a full reduction. OpenCL programmers can use the OpenCL runtime information to obtain the maximum number of work-groups and threads per work-group; information that varies depending on the device (e.g., CPU and GPU models).

OpenCL Kernel Listing 2 shows an OpenCL kernel that implements a reduction. This code follows the representation explained in Figure 1. The keyword **kernel** is an OpenCL modifier that indicates to the compiler that this is the main function to run on the GPU. The keywords **global** and **local** from the list of parameters indicate that variables are stored in global and local memory respectively. Note that OpenCL programmers have full control of the GPU memory hierarchy. All variables declared within the kernel are private and stored in private registers of the GPU.

Line 7 copies data from the GPU global memory to the GPU local memory, which is around 100x times faster. The loop in lines 8-15 performs the reduction within a block of threads. Since all threads within the same work-group store the result into the same variable, we need to add a barrier to guarantee the correctness of the result. Note that OpenCL barriers have to be manually inserted by the programmer. As we show in Figure 1, the reduction sums up the values within the same work-group. Once the reduction within a work-group finishes, the partial reduction needs to be copied back from local memory to global memory (line 17).

Java challenges to efficient reductions Java currently has no support for executing reductions on a GPU automatically without using external libraries and wrappers. This limits the hardware in which Java programs could run on.

Java 8 introduced the use of common parallel skeletons through the use of streams for Java collections. These streams, although they provide parallel operations, they not guarantee parallelism, and in some cases, they may slow down

Figure 2. Overview of the current Tornado system.

applications² [29]. Moreover, Java streams can not currently exploit GPUs transparently. We address this problem by adding automatic JIT compilation for transforming sequential reductions to OpenCL.

3 System Overview

We implemented reductions on top of Tornado [6, 15], a heterogeneous programming framework to automatically compile Java programs into OpenCL C. Tornado makes use of the Graal compiler [9], a new open-source Java JIT compiler implemented in Java that has been recently integrated into JDK 10 as an experimental compiler. We augmented the Tornado compiler to enable GPU JIT compilation for reduce-operations within Java. This section provides an overview of Tornado and its JIT compiler, while Section 4 presents our technique to perform automatic reductions.

Figure 2 shows an overview of the Tornado framework. The light-green components highlight the Tornado’s sub-systems across the software stack. As shown, Tornado is composed of a task-based parallel API, a runtime system, an OpenCL JIT compiler and a lightweight layer for interacting with the OpenCL drivers.

Tornado API The parallel API allows programmers to identify parallel sections of the input Java code and compose tasks to be executed on the parallel hardware. The API currently provides a Java annotation, `@Parallel`, that programmers can use to annotate sequential loops. This annotation is then used by the Tornado JIT compiler to generate OpenCL C code. The API also exposes a set of operations to a pipeline of tasks, called `TaskSchedule`. Each task references an existing Java method.

Listing 3 shows an example of a map-reduce computation within Tornado. Line 12 creates a group of tasks called `TaskSchedule`, while lines 13-14 create the parallel tasks that reference existing Java methods with their corresponding parameters. Later, the Tornado JIT compiler will transform these methods into OpenCL C. Note that the Java code is

```

1 public class Compute {
2     public void map(float[] in, float[] out) {
3         for (@Parallel int i = 0; i < n; i++) {
4             out[i] = in[i] * in[i];
5         }
6     public void reduce(float[] in, float[] out) {
7         for (int i = 0; i < n; i++) {
8             out[0] += in[i];
9         }
10    public void compute(float[] in, float[] out,
11        float[] temp) {
12        TaskSchedule t0 = new TaskSchedule("s0")
13            .task("t0", this::map, in, temp)
14            .task("t1", this::reduce, temp, out)
15            .execute();
16    }

```

Listing 3. Example of the Tornado Task Parallel API.

in the form of pure Java sequential code with the addition of the `@Parallel` annotation. Moreover, the reduction implemented with this unmodified Tornado version computes the sequential implementation. Finally, line 15 invokes the `execute` method that runs the method on the GPU.

Tornado Runtime Once the `execute` method is invoked, Tornado builds a data flow graph (DFG) that models a task schedule. Tornado uses this new DFG to optimize and automate data transfers to and from the GPU. Tornado is constrained to the OpenCL compute and memory model. Tornado currently does not support dynamic object allocation on GPUs due to the lack of support in pure OpenCL. However, the Tornado runtime keeps the input and output variables consistent across the Java heap and the device heap (e.g., the GPU heap), and knows exactly which buffers are allocated and copied to each device through the DFG built from the task schedule.

Tornado OCL JIT Compiler and Driver Tornado OpenCL JIT compiler generates OpenCL code from Java bytecodes, that represent the input tasks, by using Graal. The current version of Tornado optimizes map computations and exploits the `@Parallel` Java annotation.

Figure 3 shows a representation of the OpenCL JIT compilation process within Tornado. The top of the figure shows an example of a parallel map computation while the right side depicts the output (the OpenCL C generated code). The input Java code is compiled with a standard Java compiler. Then, when the application is running, Tornado compiles the input tasks to OpenCL. To achieve that, Tornado builds a Control Flow Graph (CFG) and a Data Flow Graph (DFG) using the same representation of the Graal IR [9] from the Java bytecode. In addition, Tornado applies a set of common compiler optimization phases over this IR, such as loop

²<https://goo.gl/yHnxSK>

Figure 3. OpenCL JIT Compilation process within Tornado.

unrolling, partial escape analysis [23], and constant propagation. Furthermore, Tornado applies a set of compiler passes for optimizing the code for heterogeneous architectures, such as parallel-loop exploration, and task specialization (IR specialization depending on the target device).

Tornado has three different types of IRs (high-IR, mid-IR, and low-IR), in which the compiler applies different types of optimizations. For example, high-IR is used to apply non-hardware dependent optimizations; meanwhile, on the low-IR Tornado will apply hardware-specific optimizations. *Lowering* are transitions between these IRs, in which the compiler can also apply optimizations, such as **snippets** [22] and node replacements. Finally, the Tornado driver interacts with the corresponding OpenCL platform to execute the code.

3.1 Compiler Snippets

Compiler snippets are pre-compiled and optimized code regions that can be used by a JIT compiler to replace common operations. Snippets are usually implemented in low-level languages like assembly code. However, in Graal, code snippets are implemented in Java [22], and they express low-level operations in a high-level programming language. Since Graal compiles Java, snippets are also inserted, at runtime, into the same compile graph, and therefore, re-optimized.

Graal snippets are commonly used to replace functions such as array and math operations, insert write barriers, and perform allocations. All these operations are low-level within the VM. *In this paper, we showcase enhanced usability of compiler-snippets inside the compiler. We abstract and express high-level common structured parallel design patterns as snippets to automatically compile Java code to efficient OpenCL for running on GPUs.* The next Section explains, in detail, how we use the introduced reduction snippets to automatically enable reduction operations from Java to OpenCL.

```

1 public void reduce(float[] input,
2                   @Reduce float[] result) {
3     result[0] = 0.0f;
4     for (@Parallel int i = 0; i < input.length; i++) {
5         result[0] += input[i];
6     }
7 }

```

Listing 4. Example of reductions with Tornado

4 Enabling Automatic OpenCL Reductions

This section presents the compilation process as well as an API that enables the JIT compiler to automatically exploit parallelism for reduction operations. We first present the changes in the API and then we show the IR extensions for supporting atomics that allow compiler snippets to perform node replacement for reductions.

4.1 Expressing Reductions within Tornado

We introduce and expose to developers a new **@Reduce** Java annotation for expressing reductions within Tornado. The **@Reduce** annotation does not force parallelism. Instead, it is taken by the Tornado compiler as a hint for parallelization, providing relaxed parallel semantics. In combination with the existing **@Parallel** annotation, programmers can express parallelism for many Java applications with minimal changes in their source code.

Listing 4 shows an example of a reduce-computation using Tornado. The code is similar to the Listing 1 with the difference that the result is returned into an array. The new code version is annotated with **@Reduce** on the `result` array. Furthermore, the loop is also annotated with **@Parallel**. These two annotations are essentially compiler-hints to the Tornado JIT compiler to translate, at runtime, this input method

Figure 4. New IR nodes introduced after reduce-detection.

to OpenCL C. The rest of this section presents, in detail, how this translation process works.

4.2 OpenCL IR Nodes

As described in Section 3, Tornado starts its compilation process by building a CFG and a DFG of the input bytecodes. This graph is then used to apply compiler optimizations.

We introduce a new compiler phase to explore Tornado API annotations and perform node replacements. This phase traverses the CFG and obtains node usages from a node annotated with `@Reduce`. Note that reduce-variables are annotated at the parameter list of the methods. If we detect parameters with the reduce annotation, the new Tornado phase also performs an analysis to detect if the actual annotated parameters correspond to a reduction or not. This means that, as mentioned earlier, the Java annotation is taken as a hint by the Tornado compiler and it does not force parallelism if the compiler does not detect that the input variable is used to perform a reduction.

Reduction detection Tornado is able to detect simple reductions automatically from the CFG. First, it gets the usages from the list of the input parameters. Then, it will check data flow dependencies for all of these usages to check if the output value is actually computing and writing values into the same position that it is loading data from. This is a simple technique that allows automatic identification of simple reduce-operations. If the reduction detection for an input parameter returns `true`, then it performs node replacement with the new corresponding IR nodes.

OpenCL IR Nodes for Reductions If the reduction detection phase succeeds, Tornado applies node replacement to identify sections in which reductions should be applied. Figure 4 shows an example of this compiler transformation for the input Java code of Listing 4. The left side of the Figure shows the Graal IR before applying our compiler transformations. The right side of the Figure shows the new nodes introduced by the Tornado’s compiler-phase. Dash arrows represent data flow and solid top-down arrows represent control flow.

Figure 5. OpenCL reduce-snippet replacement during lowering.

We introduced two types of nodes in the IR: a Reduce operation node and a store atomic node. Since an addition is represented as a data flow node in our CFG, we do not consider this operation as an atomic in the graph. Only when we perform a store (store index or store field), Tornado applies that operation as atomic. This is because the code generator will traverse the control flow and will obtain all dependencies needed from the data flow nodes.

Our implementation provides reduction operations and stores for multiple data types (such as `int`, `float` and `double` Java types). These new nodes are then used by the Tornado compiler in later phases to perform the final node replacement with the actual reduction.

4.3 Snippets for Reductions

When Tornado performs the lowering from HIR to LIR, it applies new node replacements (preparing the IR for the final code generation) and inserts new code snippets. In this process, Tornado applies the pre-defined reduction snippets to the current compiler-graph.

Figure 5 shows a representation of the compiler snippet transformation during lowering. If the input graph contains the node `StoreAtomic` (could be an index or a field), then our OpenCL JIT compiler creates a reduction snippet and performs the node substitution for the pre-defined reduction. The left side of Figure 5 shows the input for the lowering phase before applying the substitution. The middle graph shows a representation of the pre-defined reduce snippet. After this transformation, the snippet is inlined into the compilation graph, and Tornado continues with the optimization and lowering pipeline to generate OpenCL C code (right side of Figure 5).

Java Snippets for Reductions The reduction snippets are fully implemented in Java. These snippets implement the reduction parallel skeletons following the OpenCL semantics. Listing 5 shows an example of one of the pre-defined snippets in the Tornado compiler. Depending on the input data type and the type of reduction operation involved (e.g., an addition), Tornado invokes a different pre-defined compiler snippet. The Java code shown in Listing 5 shows the snippet for performing reduction using `float` Java arrays.

```

1  @Snippet
2  public static void reductionFloat(float[] inputArray, float[] outputArray, int gidx, float value) {
3      int localIdx = OpenCLIntrinsics.get_local_id(0);
4      int localGroupSize = OpenCLIntrinsics.get_local_size(0);
5      int groupID = OpenCLIntrinsics.get_group_id(0);
6      // Obtain the thread-id
7      int myID = localIdx + (localGroupSize * groupID);
8      inputArray[myID] = value;
9      // Performs the reduction within the work-group
10     for (int stride = (localGroupSize / 2); stride > 0; stride /= 2) {
11         OpenCLIntrinsics.localBarrier();
12         if (localIdx < stride) {
13             inputArray[myID] += inputArray[myID + stride];
14         }
15     }
16     // Copy partial results to the output
17     OpenCLIntrinsics.globalBarrier();
18     if (localIdx == 0) {
19         outputArray[groupID] = inputArray[myID];
20     }
21 }

```

Listing 5. Reduction compiler snippet for the Java `float` data type and the plus operator.

Note that this snippet is similar to the OpenCL C native code we introduced in Section 2 but without using local memory. Therefore, the whole computation is currently performed on the GPU’s global memory. In the future, we plan to augment the Tornado compiler to also use local memory.

As shown in Listing 5, the snippet first obtains the thread information, a group identifier, and a local identifier by querying the OpenCL runtime API (lines 3-7). We achieve this by using compiler intrinsics inside the compiler snippets. For example, `OpenCLIntrinsics.get_local_id` obtains the local identifier within the work-group. Then the snippet performs the actual reduction within the work-group (loop in lines 10-15). It first applies a local barrier (in which threads within the same work-group are synchronized) and then it performs the reduction. Once reductions within each work-group are finished, we copy the data back to the result array (`outputArray`).

IR Nodes for OpenCL Intrinsics This snippet is automatically inlined into the compiler graph after the first lowering phase. Since we have also other compiler intrinsics, such as the barriers and queries to obtain the thread’s information, we also perform a new node replacement in the mid-tier compilation pipeline in Tornado. In this new phase, we substitute the invoke nodes corresponding to the OpenCL intrinsics with control flow nodes that match each operation. For instance, when we find an invoke node for the `localBarrier` operation, we insert a control flow node called `OCCLocalBarrier`. This new node is used during the final code generation, producing the local barrier OpenCL builtin.

5 Evaluation

This section presents a performance evaluation our work in progress towards exploiting Java reductions on GPUs. We run our set of benchmarks on a server with a CPU and a GPU. The CPU is an Intel i7-7700 @4.2GHz with 64 of RAM, while the GPU is an NVIDIA Quadro GP100 GPU with 16GB of RAM (NVIDIA driver 384.111).

In the software side, we use CentOS 7.4 with the Linux Kernel 3.10. We also use OpenCL 1.2 provided with the OpenCL tools and compilers by NVIDIA. Tornado is compiled and executed with Java 1.8.131 with JVMCI³ support⁴.

5.1 Benchmarks

To evaluate the code quality of the generated code by the Tornado JIT compiler when using reductions, we ported two benchmarks from OpenCL to Java using the Tornado API. The applications are sum, and reductions using multiplication (`mul`).

Measurements In order to make a fair comparison between the Java managed code and the statically compiled C++ code, we show peak performance by reporting the median time of 101 iterations of the OpenCL kernel execution time. We run our set of experiments with 12GB of Java heap memory. However, since we measured the OpenCL kernel times directly reported from the OpenCL driver, Java GC does not have any influence in the measurements.

³<http://openjdk.java.net/jeps/243>

⁴<https://github.com/beehive-lab/graal-jvmci-8/tree/tornado>

Figure 6. Performance of the sum benchmark. The left side shows kernel execution time (the lower the better). The right sides shows speedup compared to Tornado (the higher, the better).

Figure 7. Performance of the mul benchmark. The left side shows kernel execution time (the lower the better). The right sides shows speedup compared to Tornado (the higher, the better).

Figure 8. Speedup of sum and mul benchmarks over Java sequential.

Data Size We study and evaluate the performance of each benchmark for multiple data sizes. We varied the input data in power of two from 4096 to 67108864 elements. This means that the evaluated datasets occupy between 30KB and 268MB.

5.2 Performance Analysis

Figures 6 and 7 show the performance evaluation results for the sum and mul benchmarks. The left side of these two figures shows the kernel execution times of Tornado and

the different versions of pure OpenCL, which we call OCL-64 to OCL-1024; each version corresponds to a different work-group size. The X-axis shows the input data size while y-axis shows the kernels' execution times for the figures on the left (the lower, the better), and speedup over Tornado for the figures on the right (the higher, the better). As shown in Figures 6 and 7, the work-group sizes influence the performance. Since Tornado selects 256 threads per block-size by default, we also studied the performance of the native code using different block-sizes.

As shown in Figures 6 and 7, Tornado's performance is almost on par with that of the native OpenCL code. As illustrated in the speedups graphs (right side), we can see that Tornado achieves almost the same performance as native code by using the 1024 block-size (only 3% slowdown for the sum benchmark and 18% slowdown for the multiplication benchmark). If we compare Tornado (which uses 256 block size) against the OCL-256 configuration, Tornado achieves up to 85% of the native performance for both benchmarks (sum and multiplication).

Figure 8 shows the speedup of Tornado compared to the Java sequential implementation. X-axis shows the input data size while the y-axis shows the actual speedup. Each bar represents a different benchmark (one for the sum benchmark and other for the mul benchmark). As shown, Tornado achieves a minimum of 1.4x and a maximum of 20.5x over Java sequential code.

6 Related Work

Parallel skeletons are extensively used by numerous parallel programming frameworks, and modern programming languages. Java is one of the few programming languages that does not include parallel skeletons in the language definition. Stream and parallel operations such as `map` and `reduce` were introduced in JDK 8 for Java collections. However, none of these operations can be transparently executed on GPUs. To the best of our knowledge, no prior work exists that automatically accelerates reductions on GPUs for Java programs.

GPU JIT Compilation for Java The most related projects are Aparapi [2], and IBM J9 [14]. Aparapi is a parallel programming framework and a compiler that can dynamically compile Java code to OpenCL and execute it on GPUs. Compilation with Aparapi takes place at runtime from the Java bytecode. Aparapi programmers express GPU code by extending the base class and overriding a runner method. Aparapi, although it is programmed using a high-level programming language, it remains low-level because developers need to know hardware details such as GPU threads, barriers, and GPU memory hierarchies. Tornado does not expose low-level hardware details to programmers and everything is automatically managed by the runtime and the GPU JIT compiler.

IBM J9 [14] is also a parallel framework and a JIT compiler for running Java 8 streams on GPUs via CUDA PTX. This compiler is limited to the `forEach` method of the stream API to compile at runtime on the GPU. CUDA code generation within IBM J9 is directly mapped from the Java bytecodes. On the contrary, Tornado uses different IR levels that are progressively lowered from the Java bytecode to OpenCL C. This allows us to perform several compiler optimizations as well as apply pre-defined snippets enabling advanced compiler optimizations.

Marawacc [10–12] introduced a GPU JIT compiler based on Graal to automatically compile input Java programs into OpenCL C. Marawacc also includes a functional API, using `map` and `reduce`. However, reductions are only supported for parallel CPUs. Marawacc differs from Tornado in that snippets cannot be applied. OpenCL code is directly generated from the High-IR of Graal, losing opportunities for applying more compiler optimizations.

Sumatra [27], Rootbeer [20], and JaBEE [34] are also similar projects that compile, at runtime, Java programs to HSAIL, PTX and CUDA respectively. In contrast to Tornado, none of these projects supports reductions.

GPU JIT Compilers for other Programming Languages

Similarly to our proposal for supporting reductions within Tornado, Numba [16] has also introduced an annotation system for Python programs. Developers annotate methods and the Numba JIT compiler creates a CUDA parallel version of the code. The Numba compiler transforms the Python input code to LLVM, which is then used to compile to CUDA

PTX. On the contrary, Tornado JIT compiler as well as all snippets are fully implemented in Java.

Copperhead [5] is another JIT compiler that translates a subset of Python to CUDA C. It uses Python decorators (Python annotations that are able to inject and modify Python code) as a way to identify source code regions to be executed on GPUs. These decorators are similar in behavior to compiler snippets, with the exception that our approach works in the IR level instead of the source level, and therefore, they are language agnostic.

Other approaches such as RiverTrail [13] and ParallelJS [31] compile JavaScript at runtime to OpenCL C. However, they use new data types as collections that have to be ported on GPUs. On the contrary, Tornado compiles to OpenCL C existing Java primitive types and certain Java objects.

7 Conclusions

Despite the fact that parallel skeletons are widely used for parallel and heterogeneous programming, there is little work on how to automatically generate parallel reductions on GPUs for Java programs. In this paper, we present our work in progress towards generating and exploiting efficient parallel reductions on GPUs for Java programs. We first introduce the `@Reduce` annotation for Java programmers as a way to instruct the compiler where reductions are located. With our approach, we exploit the parallelism of reductions through JIT compilation. We demonstrate that the combination of the addition of new nodes in the compiler IR graph with the compiler snippets is a powerful tool to express compiler optimizations and reductions in OpenCL semantics from the Java perspective. Our results demonstrate that we are able to execute reductions within 85% of the performance of the best native code version, while achieving a speedup of up to 20x compared to the Java sequential implementations.

Future Work For future work, we plan to combine the presented technique with GPU local memory in order to push the performance boundaries even further. In addition, we plan to investigate heuristics to decide the right amount of block-sizes and work-groups to achieve maximum performance per application.

Acknowledgments

This work is partially supported by the EU Horizon 2020 E2Data 780245 grant. Authors would also like to thank David Leopoldseder and Foivos Zakkak for fruitful discussions and feedback.

References

- [1] NVIDIA Tesla P100, The Most Advanced Datacenter Accelerator Ever Built, 2016. <https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf>.
- [2] AMD. Aparapi, 2016. <http://aparapi.github.io/>.
- [3] C. Barrett, C. Kotselidis, and M. Luján. Towards Co-designed Optimizations in Parallel Frameworks: A MapReduce Case Study. In

- Proceedings of the ACM International Conference on Computing Frontiers*, CF '16, pages 172–179, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4128-8. doi: 10.1145/2903150.2903162.
- [4] Carbone and Asterios Katsifodimos and Stephan Ewen and Volker Markl and Seif Haridi and Kostas Tzoumas. Apache Flink TM: Stream and Batch Processing in a Single Engine Paris. 2016.
 - [5] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: Compiling an Embedded Data Parallel Language. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, 2011.
 - [6] J. Clarkson, J. Fumero, M. Papadimitriou, M. Xekalaki, and C. Kotselidis. Towards Practical Heterogeneous Virtual Machines. In *Conference Companion of the 2Nd International Conference on Art, Science, and Engineering of Programming*, Programming'18 Companion, pages 46–48, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5513-1. doi: 10.1145/3191697.3191730.
 - [7] J. Clarkson, J. Fumero, M. Papadimitriou, F. S. Zakkak, M. Xekalaki, C. Kotselidis, and M. Luján. Exploiting High-Performance Heterogeneous Hardware for Java Programs using Graal. In *Proceedings of the 15th International Conference on Managed Languages and Runtimes*, ManLang '18, 2018.
 - [8] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.
 - [9] G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '13, pages 1–10, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2601-8. doi: 10.1145/2542142.2542143.
 - [10] J. Fumero. *Accelerating Interpreted Programming Languages on GPUs with Just-In-Time and Runtime Optimisations*. PhD thesis, The University of Edinburgh, UK, 2017.
 - [11] J. J. Fumero, M. Steuwer, and C. Dubach. A Composable Array Function Interface for Heterogeneous Computing in Java. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, pages 44:44–44:49, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2937-8. doi: 10.1145/2627373.2627381.
 - [12] J. J. Fumero, T. Rummelg, M. Steuwer, and C. Dubach. Runtime Code Generation and Data Management for Heterogeneous Computing in Java. In *Proceedings of the Principles and Practices of Programming on the Java Platform*, PPPJ '15, pages 16–26, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3712-0. doi: 10.1145/2807426.2807428.
 - [13] S. Herhut, R. L. Hudson, T. Shpeisman, and J. Sreeram. River Trail: A Path to Parallelism in JavaScript. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 729–744, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2374-1. doi: 10.1145/2509136.2509516.
 - [14] K. Ishizaki, A. Hayashi, G. Koblenst, and V. Sarkar. Compiling and Optimizing Java 8 Programs for GPU Execution. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 419–431, Oct 2015. doi: 10.1109/PACT.2015.46.
 - [15] C. Kotselidis, J. Clarkson, A. Rodchenko, A. Nisbet, J. Mawer, and M. Luján. Heterogeneous Managed Runtime Systems: A Computer Vision Case Study. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '17, pages 74–82, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4948-2. doi: 10.1145/3050748.3050764.
 - [16] S. K. Lam, A. Pitrou, and S. Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pages 7:1–7:6, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-4005-2. doi: 10.1145/2833157.2833162.
 - [17] D. Lea. A Java Fork/Join Framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, pages 36–43, 2000.
 - [18] M. McCool, J. Reinders, and A. Robison. Structured Parallel Programming: Patterns for Efficient Computation. 2012.
 - [19] OpenCL. OpenCL, 2009. <http://www.khronos.org/opencl/>.
 - [20] P. Pratt-Szeliga, J. Fawcett, and R. Welch. Rootbeer: Seamlessly Using GPUs from Java. In *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICES)*, 2012 IEEE 14th International Conference on, pages 375–380, June 2012.
 - [21] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture*, pages 13–24, 2007.
 - [22] D. Simon, C. Wimmer, B. Urban, G. Duboscq, L. Stadler, and T. Würthinger. Snippets: Taking the High Road to a Low Level. *ACM Trans. Archit. Code Optim.*, 12(2):20:20:1–20:20:25, June 2015. ISSN 1544-3566. doi: 10.1145/2764907.
 - [23] L. Stadler, T. Würthinger, and H. Mössenböck. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 165:165–165:174, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2670-4. doi: 10.1145/2544137.2544157.
 - [24] M. Steuwer and S. Gorlatch. Skelcl: a high-level extension of opencl for multi-gpu systems. *The Journal of Supercomputing*, 69(1):25–33, 2014.
 - [25] M. Steuwer, T. Rummelg, and C. Dubach. LIFT: A functional data-parallel IR for high-performance GPU code generation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 74–85, Feb 2017. doi: 10.1109/CGO.2017.7863730.
 - [26] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *IEEE Des. Test*, 12(3):66–73, May 2010. ISSN 0740-7475. doi: 10.1109/MCSE.2010.69.
 - [27] Sumatra. Sumatra OpenJDK, 2015. <http://openjdk.java.net/projects/sumatra/>.
 - [28] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: Modular MapReduce for Shared-Memory Systems. In *Proceedings of the 2nd International Workshop on MapReduce and its Applications*, pages 9–16, 2011.
 - [29] Y. Tang, R. Khatchadourian, M. Bagherzadeh, and S. Ahmed. Towards safe refactoring for intelligent parallelization of java 8 streams. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*, ICSE '18, pages 206–207, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5663-3. doi: 10.1145/3183440.3195098.
 - [30] The Apache Software Foundation. Hadoop Project Website. <http://hadoop.apache.org/>.
 - [31] J. Wang, N. Rubin, and S. Yalamanchili. ParallelJS: An Execution Framework for JavaScript on Heterogeneous Systems. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, GPGPU-7, pages 72:72–72:80, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2766-4. doi: 10.1145/2576779.2576788.
 - [32] R. M. Yoo, A. Romano, and C. Kozyrakis. Phoenix Rebirth: Scalable MapReduce on a Large-scale Shared-Memory System. In *IEEE International Symposium on Workload Characterization*, pages 198–207, 2009.
 - [33] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.
 - [34] W. Zaremba, Y. Lin, and V. Grover. JaBEE: Framework for Object-oriented Java Bytecode Compilation and Execution on Graphics Processor Units. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 74–83, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1233-2. doi: 10.1145/2159430.2159439.