Runtime Code Generation and Data Management for Heterogeneous Computing in Java

Juan José Fumero Toomas Remmelg Michel Steuwer Christophe Dubach
The University of Edinburgh
{juan.fumero, toomas.remmelg, michel.steuwer, christophe.dubach}@ed.ac.uk

Abstract

GPUs (Graphics Processing Unit) and other accelerators are nowadays commonly found in desktop machines, mobile devices and even data centres. While these highly parallel processors offer high raw performance, they also dramatically increase program complexity, requiring extra effort from programmers. This results in difficult-to-maintain and non-portable code due to the low-level nature of the languages used to program these devices.

This paper presents a high-level parallel programming approach for the popular Java programming language. Our goal is to revitalise the old Java slogan – *Write once, run anywhere* — in the context of modern heterogeneous systems. To enable the use of parallel accelerators from Java we introduce a new API for heterogeneous programming based on array and functional programming. Applications written with our API can then be transparently accelerated on a device such as a GPU using our runtime OpenCL code generator.

In order to ensure the highest level of performance, we present data management optimizations. Usually, data has to be translated (*marshalled*) between the Java representation and the representation accelerators use. This paper shows how marshal affects runtime and present a novel technique in Java to avoid this cost by implementing our own customised array data structure. Our design hides low level data management from the user making our approach applicable even for inexperienced Java programmers.

We evaluated our technique using a set of applications from different domains, including mathematical finance and machine learning. We achieve speedups of up to 500× over sequential and multi-threaded Java code when using an external GPU.

Categories and Subject Descriptors D Software [*D.1.3*]: Concurrent Programming

Keywords Algorithmic Skeletons,Parallel Patterns,Code Generation,Heterogeneous Systems,GPGPU

PPPJ '15, September 08-11, 2015, Melbourne, FL, USA.

http://dx.doi.org/10.1145/10.1145/2807426.2807428

1. Introduction

Computer systems are increasingly becoming more complex and heterogeneous. Systems ranging form workstations to smart phones are nowadays equipped with multiple parallel processors. These systems often feature accelerators like Graphics Processing Units (GPUs) or Intel Xeon PHI. These devices have evolved to general purpose hardware and can now be effectively used to accelerate many types of parallel workloads.

Specialized parallel programming approaches have been introduced targeting heterogeneous systems (e.g., OpenCL [30]) or GPUs in particular (e.g., CUDA [10]). However, programming such systems using these low-level languages is a challenging task even for experienced programmers: parallelism is expressed explicitly by the programmer and data has to be moved manually to and from the GPU memory. In recent years, this problem has been acknowledged by the research community and approaches reducing the programming effort for heterogeneous computing have been proposed. For instance OpenACC [29] and OpenMP 4.0 [32] allow programmers to target heterogeneous devices by adding annotations to sequential loops. This reduces the amount of boilerplate code, but still requires programmers to identify and explicitly decide how the parallelism should be exploited.

Unfortunately, incorporating heterogeneous computing in mainstream object oriented languages like Java has so far received little attention. Existing solutions based on bindings such as JOCL [23] for Java enable the use of OpenCL device from Java by exposing all the low-level details of OpenCL. The programmer is responsible for handling the complexity of programming the heterogeneous system by writing the actual device code in C and not in Java and orchestrating its execution. Other approaches such as Aparapi [1] requires the programmer to write in Java the computational in a low-level style similar to OpenCL.

Structured parallel programming [8, 27] is a promising approach where common parallel patterns (a.k.a., *algorithmic skeletons*) are used to easily express the algorithmic structure of the computation. Practical solutions targeting heterogeneous systems have been developed in form of high-level libraries [3, 14, 15, 21, 35] as well as programming languages like Nvidia's NOVA [9], Lime [12], or SAC [19]. In this paper, we present a high-level Java API [16] and runtime that is based on parallel patterns. It can be used to program parallel heterogeneous hardware transparently from Java. Our goal is to completely hide the underlying complexity from the programmer using a high-level interface based on well-known parallel patterns. While our API is fully Java compliant, we enable heterogeneous execution by recognizing calls to our API and compile parts of the Java application to OpenCL kernels at runtime.

In the context of heterogeneous computing and Java Virtual Machine, data management is an important aspect which has a

direct impact on performance. Usually, a time consuming explicit conversion between the Java data type and the C OpenCL data type via JNI is required. In this paper, we focus on addressing the challenge of seamlessly and efficiently managing data in Java as well as on the accelerator. We discuss how our implementation handles shortcomings of Java generics and avoids the cost of translating between different representations (*marshalling*) of data in Java and OpenCL. As we will see in the results section, our system is able to achieve a large end-to-end speedup compared with the execution of the original sequential application in Java.

To summarize, our paper makes the following contributions:

1. Presentation of a high-level pattern-based API and its implementation;
2. Presentation of a Java bytecode to OpenCL code generator for seamless heterogeneous computing in Java;
3. Presentation of optimization techniques for data management, including the avoidance of marshalling;
4. Experimental evaluation showing the benefits of heterogeneous computing and our optimization techniques using a set of benchmark applications.

The rest of the paper is organized as follows. In Section 2 we present how our pattern-based API can be used for high-level parallel programming in Java. Section 3 shows our overall system design. In Section 4 we discuss the design and implementation of our high-level pattern-based API and section 5 discusses how we generate OpenCL code from Java and execute it. Section 6 introduces and discusses our data management optimization techniques. Section 7 presents our benchmark applications and how they are expressed in our API. Section 8 evaluates our approach by showing performance numbers. Section 9 discusses related work and Section 10 concludes the paper and discuss the future work.

2. High-Level Programming with Parallel Patterns in Java

In this section we present our pattern-based Java API which extends our prior work [16] with the addition of our own array implementation discussed later. It is designed to be similar in style to the Stream API introduced in Java 8, both describing operations performed on arrays. Our design emphasizes composition and re-usability, allowing programmers to specify a computation once and apply it multiple times or reuse it as part of a larger computation. Computations are expressed by composing parallel *patterns* (a.k.a., algorithmic skeletons [8]), which are implemented in an object oriented style in our API but originally inspired from functional programming.

Listing 1 shows how the computation of dot product is expressed with our high-level API. On lines 2–5 the computation of the dot product is defined. The `AF` class contains factory methods (static) for creating all patterns known to our system. The `ArrayFunc` class implements the `Function` interface found in Java 8 where the first and second generic type parameter represent the element type of the input and output respectively. The `ArrayFunc` object representing the dot product implementation is defined by first, multiplying pairwise two vectors using the `zip2` and `map` patterns (lines 3 and 4). Then the intermediate result is summed up using the `reduce` pattern (line 5).

The `<Float,Float>` generic types before the `zip2` specify that the input to the `map` array function are two arrays whose elements are of type `Float`. This information is added so that the Java compiler can infer the correct type for the lambda expression used to customize the `map` pattern. The information is encoded as a tuple

```
1  // defining the dot product
2  ArrayFunc<Tuple2<Float, Float>, Float> dotP
3    = AF.<Float, Float>zip2
4        .map(x -> x._1() * x._2())
5        .reduce((x,y) -> x + y, 0.0f);
6
7  // input creation from two Java array a and b
8  PArray<Tuple2<Float, Float>> input
9    = PArray.from(a,b);
10
11 // executing the dot product on the input
12 PArray<Float> output = dotP.apply(input);
```

Listing 1: Dot product Java code with our pattern-based API

type, i.e., a fixed size collection of values from different types, which we added to our API. We use a `Tuple2` (a.k.a., as *Pair*) to represent the two input vectors of the dot product computation.

In order to manage data transfer between Java and an accelerator transparently for the user, we implemented our own portable array class: `PArray<T>`. The implementation of this class will be discussed later in the paper. Its usage can be seen in lines 8–9, where we have a factory method to create and initialize our `PArray` using two standard Java arrays (`a` and `b`). We automatically infer the generic type `T` of the `PArray<T>` class from the given Java arrays.

Once the computation has been defined and the input data is prepared, we can start executing the computation. Line 12 shows how the dot product computation is performed by applying the input data to the previously defined `dotP` array function. If a GPU is present in the system, our runtime will automatically move the data to the device, execute the computation there and read back the results. The resulting output is stored in the `output` array.

This example shows how our parallel patterns can be used to easily express parallel computations on arrays. Our API is tightly integrated with modern Java code and no additional programming language (e.g. OpenCL) has to be learned for targeting heterogeneous systems. We only use native Java features, like the newly introduced lambda expressions, which simplifies the programming, and we use generics to provide a strongly typed API. Currently, our API supports the well-known basic data-parallel patterns `zip`, `map`, and `reduce` which can be used as building blocks for expressing data parallelism. We will see in Section 7 how these patterns can be used to implement real-world applications in Java. We plan to extend this set of patterns in the future to support a wider range of applications (e.g. stencil code, iterative algorithms).

3. System Overview

Our system is designed to allow transparent use of a parallel accelerator (e.g. GPU) if available directly from Java. It is built around three major components: (1) a high-level Java Array programming API, (2) an OpenCL code generator and (3) a runtime that manages the execution of the computation on an OpenCL device. Our high-level API is inspired from the array programming model and provides the user with composable algorithmic patterns such as map or reduce as seen in the previous section. The OpenCL code generator extends the new Oracle Graal VM [13] (just in time compiler written in Java) to compile Java byte into OpenCL. Finally, our third component handles data transfer and device management.

Our code generator internally uses libraries provided by Graal for processing standard Java byte-code and generate OpenCL code from it. It is, therefore, not limited to run on the Graal VM, but

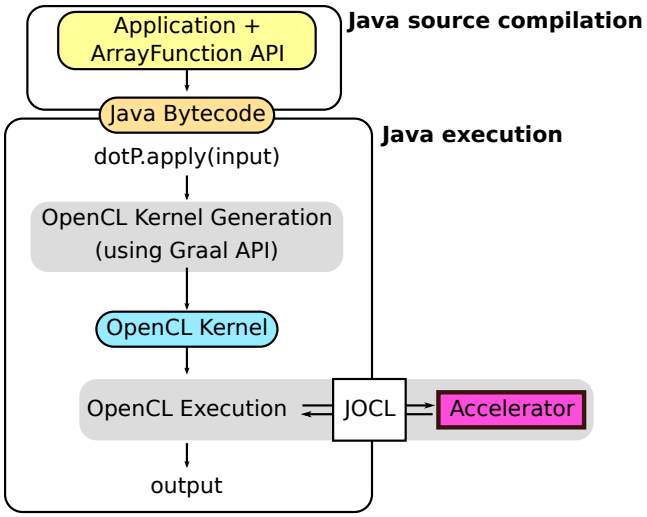

Figure 1: Overview of our system.

merely uses functionality and utilities provided by Graal which facilitates the compilation and runtime code generation.

Figure 1 gives an overview of our system and how the runtime handles the execution of the example presented in the previous section. As seen earlier, the application is written in pure Java where parallelism is expressed with the help of our Array Function API. We will describe the design and implementation of our API in Section 4. The application Java code is then compiled into standard Java byte-code and can be run as usual in a standard Java VM.

At runtime, when the application makes the first call to the application of an array function, our runtime checks the system configuration and determines whether we should attempt to run the computation on an OpenCL device (e.g. GPU). If no OpenCL device are available, the system will fall back to a pure Java implementation. If an OpenCL is available, our runtime will attempt to accelerate the computation using that device. We start the process of generating an OpenCL Kernel using our Graal OpenCL backend, which is described in Section 5. This requires running on a Java VM that supports the Graal API; if this is not available we fall back to pure Java execution. Note that we do not rely on any advance features offered by Graal at the moment but simply use it as a frontend for accessing the Java byte-code. Once the OpenCL kernel has been generated, we execute it on the accelerator using the JOCL OpenCL Java bindings, as described in Section 6. If there is any runtime error, the execution aborts and the computation resumes safely in pure Java.

In contrast with prior work such as LiquidMetal [12], our system generates the OpenCL kernel at runtime whenever it detects a usable OpenCL device. In our case the application is packaged as standard Java Bytecode without any device specific code. This is an important feature since we keep the philosophy of "compile-once run-everywhere". In addition, this offers the unique opportunity to specialise the code for the device available. The same design can be used to support another device-specific backend (e.g.. CUDA PTX) or to implement device specific optimizations.

4. API Design & Implementation

In this section we discuss the design of our Java API for heterogeneous systems. From a programming point of view, our API puts emphasis on composability and reuse of parallel patterns. From a performance point of view, our API allows applications to

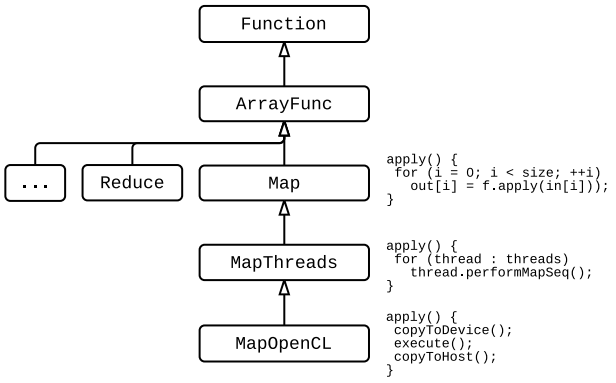

Figure 2: Inheritance of the implementation of `Map` pattern.

be portable and support execution on conventional Java as well as accelerators without requiring code changed from the programmer.

4.1 API Design

As Java is an object oriented programming language, our API is designed and implemented in an object oriented style, but still different from similar previous Java libraries like JMuesli [24] or Skandium [26]. Each parallel array operation in our API, like `Map` or `Reduce`, is represented as a class which inherits from the abstract `ArrayFunc<inT, outT>` super class. This class defines the `apply` method which is used for performing an operation on an input array and produce an output array.

A concrete operation, like `Map`, can be instantiated either directly using the `new` keyword, e.g., `new Map(f)`, or by invoking a corresponding factory method, e.g., `.map(f)`, as shown in Listing 1. All factory methods are defined in the `ArrayFunc` super-class, which allows the programmer to chain the instantiation of multiple operations. As all classes are implemented as generics our API ensures that only type safe compositions can be used.

Each parallel pattern is customized with an object satisfying the new Java `Function` interface. Most conveniently, a lambda expression is used to create this object, as shown in Listing 1. This design is fundamental different from similar previous Java libraries [24, 26], where the programmer customizes the patterns by implementing subclasses inheriting from predefined superclasses defining the interface used by the implementation of the pattern. This purely object oriented style requires a lot of boilerplate code, as new classes have to implemented and instantiated. In our API design, all of this can be avoided by using the new Java lambda expressions, which follow a more functional style and are easier to read and maintain.

4.2 Portable Implementation

Our design allows for a portable implementation which supports the execution on accelerators as well as conventional Java. Figure 2 shows the inheritance relationships for the implementation of the `Map` pattern.

The `Map` class inherits, like all parallel patterns, from the abstract `ArrayFunc` class. This class provides a straight forward sequential implementation of the map pattern in Java which can, for example, be used for testing and debugging purposes. A parallel implementation using Java threads is provided in the separate subclass `MapThreads`. This class can be customized by specifying the number of threads used for the parallelization, where our default is the number of available processor cores.

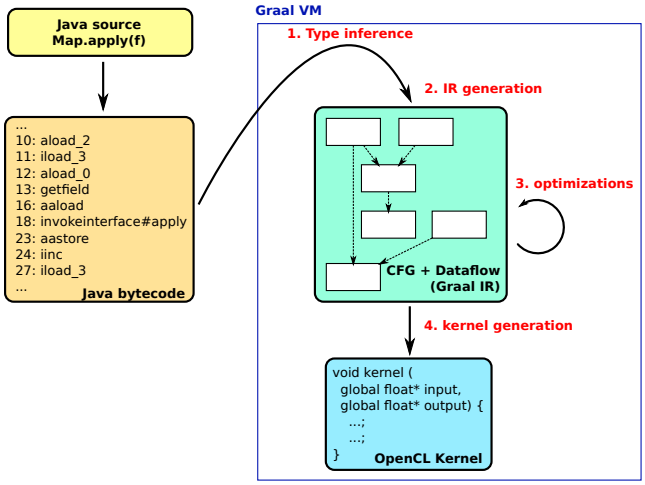

Figure 3: OpenCL code generation process from compiled Java byte-code run within the Graal VM. 1) First, we infer the input data types of the lambda expression. 2) Then the byte-code is converted into the Graal IR representation (control flow + data flow graph) and 3) various optimizations are performed. 4) Finally, our backend generates an OpenCL kernel

Finally, the OpenCL implementation of the map pattern is encapsulated in its own subclass `MapOpenCL`. This implementation performs three basic steps upon execution: 1) upload the input data to the accelerator; 2) perform the computation on the accelerator; 3) download the computed result back to Java.

The OpenCL kernel code generation happens just-in-time and will be explained in more detail in Section 5. Our implementation uses internally the Graal API to perform the OpenCL kernel generation and relies on the JOCL Java bindings for OpenCL to drive the execution. Note that if the Graal API is not present or if there is no OpenCL device detected, the execution of the computation via OpenCL is not possible. In this case the implementation of the super class is invoked as a fallback, which means the computation is performed in Java using multiple threads.

From a design point of view it is maybe suprising that `MapOpenCL` inherits from `MapThreads` instead directly from `Map`, but from an implementation point of view this design has the benefit of allowing for a straightforward implementation of the fallback behaviour.

The `map` factory method shown earlier in Listing 1 creates an instance of the `MapOpenCL` class, so that our implementation by default will first try to execute on an accelerator using OpenCL and only if that fails perform the computation in Java. Programmers also have the possibility to easily commit to a specific implementation by creating an instance of the appropriate class for performing the map computation sequentially, using multiple threads, or using OpenCL.

5. OpenCL Kernel Compilation

This section describes the process of generating OpenCL code from Java which allows us to execute computations seamlessly on GPUs. We use Oracle's Graal infrastructure [18] for our implementation and developed our own OpenCL backend for it.

5.1 Overview

Our compilation and code generation process is shown in Figure 3. The programmer writes the application by using our API as discussed earlier. A program written with our API is then compiled to ordinary Java byte-code using the standard Java compiler. This results in byte-code which represents parts of the implementation of the map parallel primitive. The byte-codes contain a virtual call to the `apply` method of the lambda `f` which implements the `Function` interface.

At runtime, when the map primitive is called, our implementation transforms the original byte-code into OpenCL code following four steps. First, we need to rediscover the types used in the lambda since all the generics have been erased. Once the type information has been recovered, the byte-code is converted to the Graal IR which is a CFG (Control Flow Graph). We invoke standard Graal IR optimizations such as inlining and node replacements to produce an optimized CFG. After optimization, we generate the OpenCL code by traversing the CFG and generating appropriate OpenCL source code for each node in the Graal IR. Once the kernel is created, we cache and reuse it if the user makes another call to the same map function.

5.2 Type Inference

Before we can actually generate an OpenCL kernel, we need to recover the type information in the lambda expression of the map function. The original type information is erased by the Java compiler since lambda functions use generics. To recover this information, our system execute a small part of the computation in pure Java for each high-level patterns (e.g. map, reduce, zip). For instance, in the case of the map pattern, we simply run the lambda expression on the first element of the input array. Using runtime reflection, we can infer both the input and output type of each expression. This information is then stored into the Graal IR to make it available when generating OpenCL code.

5.3 Graal IR Generation and Optimization

To generate the Graal IR we first identify the code regions in the program which should be compiled into an OpenCL kernel. For example, for the map pattern, the customizing lambda expression is handed over to Graal which parses the corresponding byte-code into its own internal IR [13].

Once the Graal IR is generated, we perform a series of transformations for preparing it for the OpenCL code generation. Each of these transformations operates on the control flow graph (CFG), by removing existing nodes or adding new nodes. We start the optimization process by applying common compiler optimizations:

- Canonicalizing: local optimizations such as constant folding and local strength reduction are applied.
- Inlining: method calls are replaced in the graph by the implementation of the corresponding method.
- Dead code elimination: code segments which are never executed are removed from the graph.

We apply an additional set of OpenCL specific optimizations to remove and transform nodes which do not require a corresponding implementation in OpenCL. For example, we remove null pointer checks on the inputs and output arguments to the function since we can check for these outside of the kernel. We also eliminate the (un)box nodes by converting all the wrapper objects such as `Float` or `Integer` to their corresponding primitives equivalent (`float`, `int`). In addition, we extended the existing node replacement strategy in Graal for dealing with OpenCL specific math operations. Method calls to the Java math library are transparently replaced with OpenCL specific nodes representing calls to equivalent OpenCL built-in math functions. Note that if at any point during this process we encounter a problem (e.g. a feature

used in the lambda not supported in OpenCL), we abort the OpenCL kernel generation process and fall back to pure Java execution.

5.4 OpenCL Code Generation Example

We now illustrate the optimization and code generation process with a small example. Consider the following program which multiplies all the integer input elements by two and cast them to a double value:

```
ArrayFunc<Integer,Double> mul2 =
  AF.map(x -> (double)x*2);
```

Figure 4 shows the Graal IR (leftmost) corresponding to our example. The Param node corresponds to the input parameter. The `MethodCallTarget` node in the IR together with the `Invoke#Integer.intValue` node performs the unboxing of the `Integer` object into an integer primitive. After applying the inlining and dead code optimization passes we obtain the graph in the middle, where the `invoke` nodes representing method calls have been replaced. As a result, the `intValue` invocation is lowered to an Unbox node and a null pointer check is added. Similarly the `Double.valueOf` node is lowered into a `Box` node. After applying our OpenCL specific optimizations the graph on the right is produced where exceptions and (un)boxing are removed, as there are not required in our OpenCL implementation. The graph on the right is then passed to our OpenCL kernel generator, which produces the final OpenCL code as shown in the figure.

```
1  inline double lambda0(int p0) {
2    double cast_1 = (double) p0;
3    double result_2 = cast_1 * 2.0;
4    return result_2;
5  }
6  kernel void lambdaComputationKernel (
7    global int * p0,
8    global int *p0_index_data,
9    global double  *p1,
10   global int *p1_index_data)
11 {
12   int p0_dim_1 = 0; int p1_dim_1 = 0;
13   int gs = get_global_size(0);
14   int loop_1 = get_global_id(0);
15   for ( ; ; loop_1 += gs) {
16     int p0_len_dim_1 = p0_index_data[p0_dim_1];
17     bool cond_2 = loop_1 < p0_len_dim_1;
18     if (cond_2) {
19       int auxVar0 = p0[loop_1];
20       double res = lambd0(auxVar0);
21       p1[p1_index_data[p1_dim_1 + 1] + loop_1]
22         = res;
23     } else { break; }
24   }
25 }
```

Listing 2: OpenCL Kernel automatically generated by the Graal-OpenCL runtime

Listing 2 shows the complete kernel generated automatically for the example above. Lines 1–5 show the code generated for the lambda expression, which multiplies the vector element by a constant and performs the int to double conversion. Lines 6–25 show the implementation of map in OpenCL. The global ID is obtained in line 14 and is used to access the elements from the arrays passed to the main kernel which are passed to the generated function in line 20.

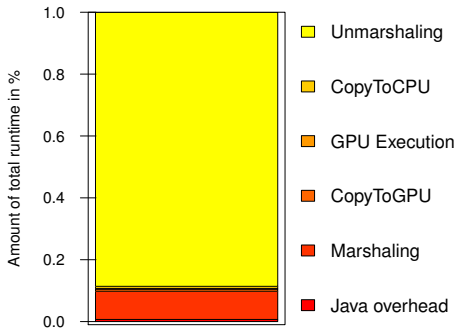

Figure 5: Breakdown of the total runtime of the Black Scholes application into single steps. The marshalling and unmarshalling steps are constituting over half of the total runtime.

6. OpenCL Data Management

In this section we discuss the challenge of efficiently managing data in a heterogeneous setting with Java and the CPU on the one side and OpenCL and the GPU on the other. We start the section by describing the challenges involved and providing some evidence of the scale of the problem. Then we discuss our optimization techniques applied to address these challenges.

6.1 The Challenge of Data Management

Efficient data management is essential in heterogeneous systems. Usually the CPU and GPU do not share the same physical memory and, therefore, data has to be moved between them which can limit the performance benefits of the GPU. The usage of Java introduces an additional challenge for efficiently managing data. Java uses *reference semantics*, i.e., references to objects are managed instead of the objects itself. When a collection of objects is stored in an array the references to the objects are stored in the array. This data management is fundamentally different from *value semantics* used in OpenCL (which it inherits from C) where not references to objects but instead the objects themselves are stored in the array.

For primitive types like `int` and `float`, Java uses the same value semantics as C but unfortunately these types cannot be used together with Java generics. As our API relies on generics for ensuring strong type safety, we cannot use primitives types directly. To circumvent this drawback Java provides corresponding wrapper types (`Integer` and `Float`) which follow Java conventional reference semantics and can be used together with generics.

The process of translating between the two data representation of Java and OpenCL is known as *(un)marshalling*. Unfortunately, marshalling is an expensive task. Figure 5 shows the runtime for the Black Scholes application (discussed later in Section 7) implemented using our API without applying any data management optimizations. We can see that the marshalling and specially the unmarshalling steps take up 90% of the runtime. In the next subsection we are going to discuss optimizations to avoid performing marshalling altogether.

6.2 A Portable Array Class

To take control of the data management and to be able to apply optimizations, we implemented our own array class which we named "PArray" for Portable Array (portable across devices). `PArray<T>` is a generic Java class, where `T` is the type of the elements stored in the array. `T` can either be the wrapper of a primitive type, e.g., `Float`, our own tuple type class, e.g.,

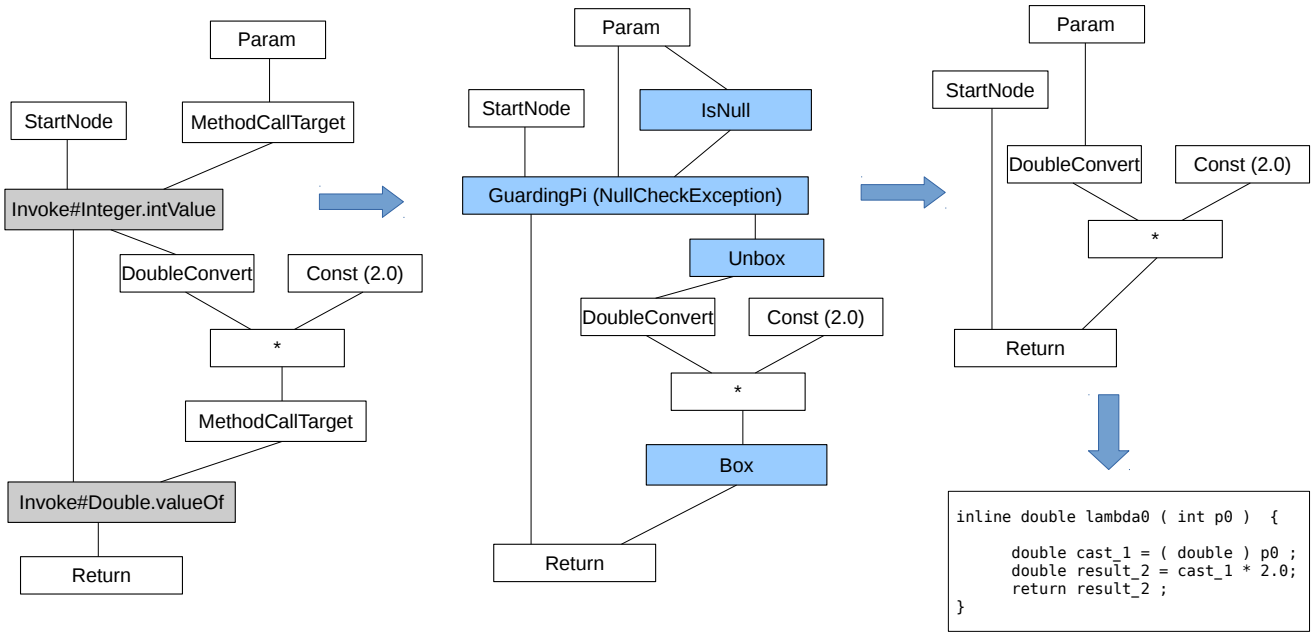

Figure 4: Graal IR optimisation phases and C OpenCL code generation. The left side shows the Graal IR for the input code in Java (lambda expression). Then, in-line Graal IR optimisation is invoked and it produces the second graph. As we produce C OpenCL code, we do not need to support null pointers, box and unbox, therefore this graph is optimised given the graph in the right side.

`Tuple2<Float, Float>` or a nested array type for representing multidimensional data. Our array class follows a value semantics, i.e., instead of references to values copies of the values are stored in the array. This enables us to avoid the marshalling step and directly pass a pointer to our internal storage to the OpenCL implementation, as no Java code can hold a reference to an element in our arrays.

6.2.1 Handling Generics

Our implementation of the `PArray<T>` class uses the *Java Reflection API* at runtime to inspect the type of `T` the first time the array is filled up with data. If `T` is a wrapper type like `Float` our implementation directly stores a buffer of the underlying primitive type; for an `PArray<Float>` internally we store a Java `FloatBuffer` from the `Java.nio` package which is a low-level data storage of primitive `float` values. This implementation strategy helps us to circumvent the drawback that Java generics cannot be used together with primitive types, which has been reported as a major performance disadvantage of JMuesli [24].

6.2.2 Tuples Support

For arrays of tuples we internally store multiple buffers, one for each component of the tuple since this layout is generally more efficient on accelerators. For example for an `PArray<Tuple2<Float, Double>>` we store one `FloatBuffer` and one `DoubleBuffer` as can be seen in figure 6. When an element of such an array is accessed from Java code we build an object of type `Tuple2<Float, Double>` on the fly. Using this data layout we can avoid the marshalling cost for tuples, as our generated OpenCL kernels will accept one array per component of the tuple as arguments.

It is possible to use our `PArray` and `Tuple` types in the lambda expressions used for customizing the patterns. For example the lambda expression might return a tuple (as it is the case for the Black-Scholes benchmark discussed later in Section 7). For instance the following code show a simple example which stores the same value twice in a `Tuple2`:

```
AF.<Float> map(x -> new Tuple2<>(x, x));
```

This is perfectly legal in our system. When generating the OpenCL kernel code for this lambda expression we generate a C `struct` representing the `Tuple2` instance.

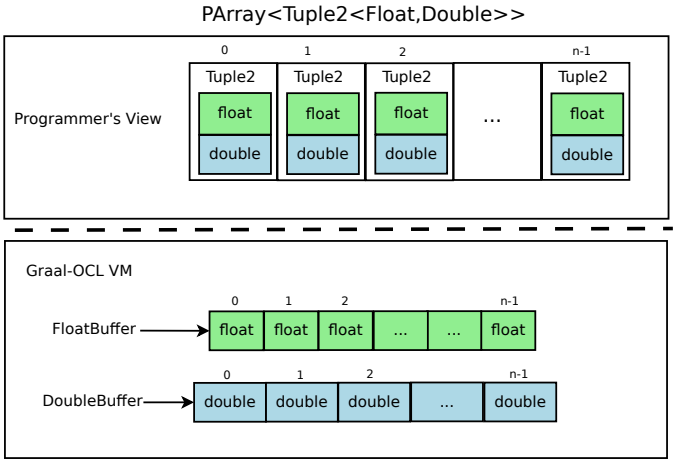

Figure 6: Strategy to avoid marshalling in the VM for the OpenCL component.

6.3 Data Optimizations

We now discuss two optimizations that our system perform specifically to reduce data management overheads (marshalling and transfer to/from the device).

6.3.1 Avoiding Marshalling

The main motivation for implementing our own array class is that we can completely avoid the cost of marshalling. The low-level buffers, like `FloatBuffer`, store their values in the same data layout as expected by C and OpenCL. When executing an OpenCL kernel we can directly move the data from the `FloatBuffer` to the GPU without the marshalling step which can add significant overhead, as we showed in our previous paper [16]. We provide custom `get` and `put` methods for accessing the elements in our arrays.

When a lambda expression reads or return tuples, the generated OpenCL kernel code for such a lambda expression will read or store its results directly from/into many distinct OpenCL buffers – one for each component of the tuple. Since our code generator knows about our special Tuple classes, we can generate the correct OpenCL code. When copied back from the GPU to the CPU these OpenCL buffers are copied directly into the allocated low-level Java buffers, thus, avoiding the unmarshalling step.

For multidimensional arrays, e.g., `PArray<PArray<Float>>`, we allocate a single flattened buffer and perform the appropriate index calculations when accessing the array. Again this allows us to circumvent the marshalling step, as in OpenCL multidimensional data is usually handled in the same manner.

6.3.2 Pinned Memory

Implementing our own array class also offers us the ability to decide precisely where to allocate the storage for the data. In the case where the computation involved the use of an OpenCL device (e.g., GPU), we allocate the data in pinned memory at the OS level (i.e., the memory pages involved will never be swapped). This enables faster transfer time between the host and the device since the GPU DMA engine can be used, freeing the CPU from managing the data transfer.

7. Benchmark Applications

In this section we present our four benchmark applications ranging from linear algebra (*saxpy*), to applications from the fields of mathematical finance (*Black-Scholes*), physics (*N-Body* simulation), data analysis (*Monte Carlo* simulation) and machine-learning with *k-means* clustering). We discuss how they can be implemented by an application developer using our pattern based API.

7.1 Single-Precision alpha X Plus Y (*saxpy*)

Our first benchmark application is a simple linear algebra benchmark which scales a vector with a constant α and then adds it too another vector. Listing 3 shows the implementation of *saxpy* using our API. The two input vectors are combined into an array of pairs using the `zip` pattern (in line 3). The following `map` pattern is then used to specify how to process each pair of elements. The computation is specified using a lambda expression (see line 4), where the two corresponding elements of both vectors are added together after the first vector has been scaled with `alpha`. This example shows how ordinary variables defined outside of the patterns can naturally be accessed inside an lambda expression.

7.2 Black-Scholes

The Black-Scholes financial mathematics application is a standard method used is high frequency trading and computes so called *call* and *put* options for stock prices. Listing 4 shows the implementation in Java using our API. We use the `map` pattern (see line 2) to compute the call and put options for a given stock price. The `map` returns the call and put options encoded in a tuple (see line 5). Finally, in line 8 computation is applied to the array of stock prices.

```
1 float a = 1.5f;
2 ArrayFunc<Tuple2<Float, Float>, Float> saxpy
3   = AF.<Float, Float> zip2()
4       .map(p -> a * p._1() + p._2());
5
6 PArray<Float> result = saxpy.apply(left,right);
```

Listing 3: Saxpy Java code with our `ArrayFunction` API

```
1 ArrayFunc<Float, Tuple2<Float, Float>> bs
2   = AF.<Float> map( stockPrice -> {
3       float call = computeCallOption(...);
4       float put  = computePutOption(...);
5       return new Tuple2<>(call, put); });
6
7 PArray<Tuple2<Float, Float>> result
8   = bs.apply(sPrices);
```

Listing 4: Implementation of the Black-Scholes application using our pattern-based API

```
1 ArrayFunc<Tuple7<...>, Tuple7<...>> nBody
2   = AF.<...> zip7().map( b -> {
3       float f[] = computeForce(b, bs);
4       updateForce(b, f);
5       updateVelocity(b, f);
6       return b; });
7
8 do { bs = nBody.apply(bs); } while (condition);
```

Listing 5: N-Body simulation implemented using our API.

7.3 N-Body Simulation

N-Body simulations are widely used in physics and astronomy to simulate how a system of particles (a.k.a., *bodies*) behaves over time. In an iterative process, forces between each combination of pairs of bodies are computed from which the velocity of each body is derived.

Our implemented (shown in Listing 5) encodes the bodies as tuples with seven elements: the first three elements encode the position; the next three elements encode the velocity; and the last element encodes the mass of the body. In each step of the iteration (line 8) we update the array of bodies `bs` by applying the computation to each body `b` using the `map` pattern (line 2). Inside the `map` we first compute the force which act upon body `b` from all other bodies before we use this to update its force and velocity (see lines 3–5).

7.4 Monte Carlo Simulation

Monte Carlo simulations perform approximations using statistical sampling, i.e., by generating and analyzing a large number of random numbers. This basic technique is useful in a number of applications ranging from risk management, physics simulation, to data analysis.

In our implementation we generate a large number of random values to approximate the value of π. We generate an array of random numbers in Java on which we apply the `map` pattern. Each random number serves as a random seed to initialize the pseudo-random number generator on the GPU which generates a large number of random numbers in parallel.

```
ArrayFunc<Float, Float> mc
  = AF.<Float> map( seed -> {
    for (int i = 0; i < iter; ++i) {
      seed = updateSeed(seed);
      float x = generateRandomNumber(seed);
      float y = generateRandomNumber(seed);
      if (dist(x, y) <= 1.0f) { sum++; } }
    return sum / iter; } );

PArray<Float> result = mc.apply(seeds);
```

Listing 6: Monte Carlo simulation implemented with our `ArrayFunction` API

```
do { // step 1
    membership = AF.<Float,Float>zip2().map( p
     -> {
            return computeClusterID(p, centers);
          } ).apply(points);
      // step 2
    centers = computeAndUpdateCentres(points,
     memberships);
} while (condition);
```

Listing 7: Kmeans classification by using the `ArrayFunction` API

7.5 K-means Clustering

K-means clustering is a method from the domain of machine-learning to group (or *cluster*) similar data points. Its an iterative refinement process which contains of two steps: 1) each data point is assigned to its nearest cluster; 2) the new center point of the cluster is computed based on all points in the cluster.

Listing 7 shows a sketch of the implementation in Java using our API. We only express the first step using our API. This shows how our API can naturally integrate with ordinary Java code, which implements the second step. The computation is executed as an iterative process until a termination condition is met (see line 7). A data point is represented as a tuple of two float values, therefore, we use the `zip2` pattern together with the `map` pattern (see line 2) to compute the nearest new cluster id for each point `p`. Because this computational step requires access to the cluster centers, we defined the computation inside the iteration where the `centers` variable can be access inside the lambda expression. Our caching mechanism explained in section 5 ensures that we only generate the OpenCL kernel for this computation once.

After the points have been assigned to clusters we perform the second step of updating the cluster centers in pure Java by calling a static method (see line 6). This shows the tight and fluid integration of our API with Java, where one computational step is performed on the GPU and the next one in Java, both steps defined using the same language.

8. Evaluation

Experimental setup We evaluated our approach using two separate systems. The first system comprising a four core Intel i7 4770K processor with 16GB of RAM and a AMD Radeon R9 295X2 GPU with 2 × 4GB of graphics memory. The second system has the same Intel CPU also with 16GB RAM and a Nvidia GeForce GTX Titan Black with 6GB graphics memory. We use OpenCL 1.2 and the latest GPU drivers available (AMD: 1598.5, Nvidia: 331.79). The Java version used is Oracle JRE 8 (1.8.0.25).

We executed every benchmark 100 times and present the median runtime measured. We measured the execution time for a sequential version of each benchmark in pure Java and compare it against parallel versions implemented using our API running on the multi core CPU and the GPU. We include all overhead in our measurements, especially the time required for transferring data to and from the GPU. We do not include the time for performing the JIT compilation of the Java byte-code, neither for the Java Hotspot compiler nor for our Graal based implementation.

Generating the OpenCL code for our benchmarks takes between 70 and 300 milliseconds. We use a caching system to store the generated GPU code similar to the handling of CPU code by normal JVMs. Therefore, if the same GPU computation is executed multiple times, we generate the code only once and obatin the generated code from the cache afterwards.

We measure performance using two input data sizes for each benchmark. For *saxpy* the smaller input size is an array of 64 MB and the larger of 512 MB of float values. K-means processes 8 million (small) and 33 million (large) points. Black-Scholes computed options for 200 thousand (small) and 6 million (large) stock prices. N-Body simulates 64 and 256 thousand particles and Monte Carlo is initialized with 256 thousand random numbers for the small input size and with 1 million random numbers for the large input size.

Runtime Results Java Multithreaded Execution We evaluate all our benchmarks against sequential Java implementations which operate on arrays of primitive Java types. This is how Java programmers traditionally implement performance oriented applications reaching performance close to native written C code.

On our system uses a four core CPU with hyper-threading. We uses our API as shown in the previous section to implement our benchmarks and performed measurements using 1, 2, 4, 8, and 16 Java threads. The results are shown in Figure 7. The y-axis shows the speedup of our implementation over the sequential Java implementation.

The first benchmark *saxpy* is a very memory intensive benchmark with few computations performed per memory operation. Therefore, we did not expect to see a speedup as the multiple threads executing all wait for the data to be read from memory. The sequential Java application already uses the perfect memory layout, a primitive `float` array, using our API we introduce an overhead by using a `FloatBuffer` as our storage and passing the `float` values through some layers of our API.

The other four benchmarks show substantially better results, as these are benchmarks more suited for parallel execution. For the K-Means application we compare the runtime of the first step, where using our API gives a speedup of over 3× as compared to sequential Java for the larger data size. For Black-Scholes, N-Body and Monte Carlo, we can even see higher speedups of up to 5×.

We can especially see, that the single threaded implementation using our API introduces no, or only a moderate overhead, as compared to sequential Java for these four benchmarks. Overall, these results show, that our API is well-suited for implementing data-parallel applications in Java even when only using the power of a multi core CPU.

Runtime Results GPU Execution For execution on the GPU we can reuse the unmodified application we used for our runtime measurements with Java threads. Figure 8 shows the speedups we obtain over sequential Java when we perform the computations on two different GPUs: a Nvidia GeForce GTX Titan Black and a AMD Radeon R9 295X2 GPU.

In Figure 8 we use a logarithmic scale on the y-axis, because of the large speedups we obtain for some of the benchmarks. For each benchmark, input data size, and GPU we show two bars: one

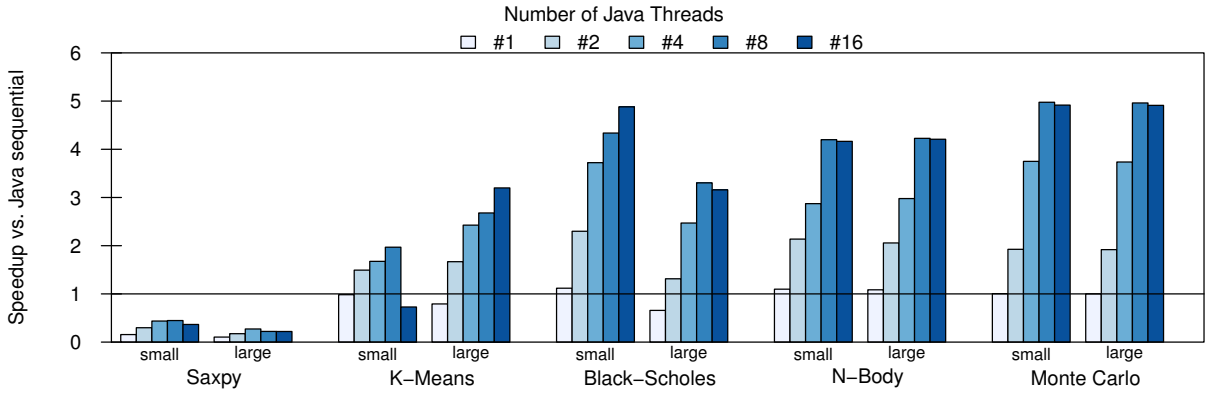

Figure 7: Speedup for all benchmark applications over sequential Java execution with multiple Java threads.

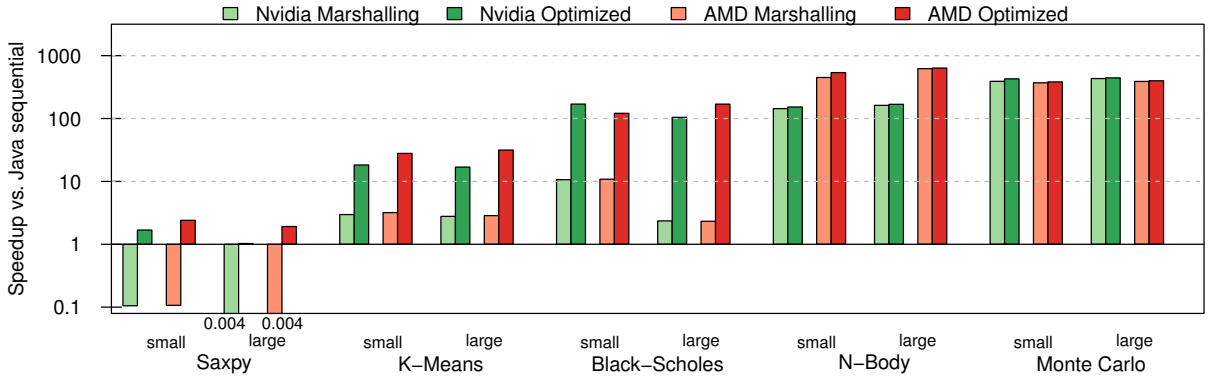

Figure 8: Speedup for all benchmark applications over sequential Java execution with GPU.

showing the application runtime with marshalling and the second bar showing the runtime with our optimizations applied as described in Section 6 which avoid marshalling.

We can see that the effect of marshalling can be very significant and our optimization technique avoiding marshalling enables large additional speedups. For the *saxpy* application we can see, that GPU execution with marshalling introduces a significant slowdown over sequential Java execution up to three orders of magnitude. By avoiding marshalling we can even obtain a small speedup for this memory-intensive benchmark.

For K-Means we can see that avoiding marshalling allows the speedups to jump by a factor of $\approx 10\times$, from about $3\times$ to $30\times$. For Black-Scholes the speedup improves from $10.0\times$ to $121.0\times$ for the small data size and from $2.3\times$ to $169.4\times$ for the large data size. These two applications are examples of applications which offer substantial parallelism, but require a significant amount of data to be transferred between Java and the GPU. For these type of applications our optimization can offer a runtime improvement of up to $70\times$ times, as observed for the large data size of Black-Scholes executed on the AMD GPU.

The last two applications N-Body and Monte Carlo are examples of applications which are very computational intensive, therefore, these applications very effectively harness the power of the GPU resulting in massive speedups of up to 600 over sequential Java for the N-Body application executed on the AMD GPU. For these applications little data is moved between Java and the GPU, therefore, our data management optimizations have only a very minor effect on the overall application runtime.

Overall the results show that our API offers high performance for data parallel application executed on the GPU. Massive performance improvements can be achieved using application written in pure Java using our API. In addition, for some applications our data management optimizations significantly improve performance.

9. Related Work

Other projects have been proposed to simplify parallel and especially GPU programming using high-level abstractions in Java and other programming languages.

9.1 Skeleton- and Pattern-Based Approaches

Parallel pattern based programming approaches have been studied in theory and practice for some time [8, 27]. Cole [8] introduced functional parallel patterns as *algorithmic skeletons*. Multiple implementations of skeleton frameworks have been developed since for clusters and multi-core CPUs, e.g., FastFlow [4] and Muesli [15]. For a comprehensive overview see [17]. SkelCL [35] and SkePU [14] are recent skeleton frameworks offering high-level patterns similar to the once presented in this paper for programming multi-GPU systems in C++.

Intels Threading Building Blocks allows programmers to express parallel programs in a structured way [27]. Predefined

building blocks are customized and used similar to the patterns presented in this paper. TBB can be used to program multi core CPUs as well as Intels Xeon Phi accelerator. Thrust [21] and Bolt [2] are C++ libraries developed by NVIDIA and AMD which offer sets of customizable patterns to simplify GPU programming. While some ideas are similar to our approach, non of these projects target Java as we do.

JMuesli [24] and Skandium [26] are skeleton libraries for programming clusters and multi-core CPUs from Java. Different to our approach these projects do not target GPUs and do not make use of the most recent Java 8 features, like lambda expressions, which greatly simplifies the programming and allows the programmer to avoid writing extensive boilerplate code.

9.2 Low-level Language Extensions

There exists also lower level extensions for existing languages which include directive approaches such as OpenMP and OpenACC. OpenACC, as well as OpenMP in its latest version 4.0, introduce directives and clauses for heterogeneous computing as a extension for C/C++ and Fortran Programmers. Both approaches defines a set of directives which the programmer uses to annotate loops in the program. The directives add information how the loops can be parallelized and executed on GPUs. These approach have been used to target GPUs by converting OpenMP to CUDA [25], hiCUDA [20] which translates sequential C code to CUDA or `accULL`, an OpenACC implementation which generates OpenCL and CUDA source code [34]. Although these approaches saves a lot of engineering work, they remain low level: programmers have to express the parallelism explicitly and must control the data transfer to and form the GPU explicitly as well.

9.3 High-Level Languages for GPUs Programming

Many similar projects outside of the Java world have been proposed for simplifying GPU programming. Accelerate is a functional domain specific language built within Haskell to support GPU acceleration [6]. Copperhead [5] is a data parallel programming language by NVIDIA integrated in Python. NOVA [9] is a functional programming language developed by NVIDIA offering parallel patterns as primitive in the language to express parallelism. HiDP [28] is a parallel programming language which uses patterns to hierarchical structure data parallel computations. Obsidian [7] is an embedded language for Haskell which allows to program in GPU by using high level constructions. Obsidian uses an API as a intermediate layer for optimizations with CUDA. X10 [36] is a language for high performance computing that can also be used to program GPUs [11]. However, the programming style often remains low-level since the programmer has to express the same low-level operations found in CUDA or OpenCL.

Lime [12] is a Java based programming language which adds new operators to the Java programming language to easily expressing parallelism for stream computations. It has been defined by IBMs for their Liquid Metal project and can be compiled to OpenCL code for GPU execution. Programmers have to commit to learn a new language, whereas they can use our API directly in Java. Furthermore, switching the programming language is often not an option, as it requires a major investment for rewriting the entire software project.

9.4 Java-Specific Approaches for GPU programming

For programming GPUs in Java there exists simple language bindings for OpenCL (JOCL [23]) or CUDA (JavaCL [22]). However, by using these bindings programmers are forced to switch programming languages and encode their computations directly in OpenCL or CUDA. Rootbeer [33] and Aparapi [1], both developed by AMD, address this issue by converting Java bytecode to OpenCL at runtime. Nevertheless, these projects remain low-level as they still require the programmer to explicitly exploit the parallelism and do not raise the level of abstraction like our approach does.

Sumatra [31] is a new OpenJDK project aiming to use the Java 8 Stream API as input and compile it to HSAIL code which can be executed by AMD GPUs. This approach is very similar to ours in spirit, but our API design has a stronger emphasize on re usability and composability in contrast to the Java 8 Stream API. In addition, we generate OpenCL code which allows us to target a broad range of hardware which is currently not the case with HSAIL.

10. Conclusion and future work

In this paper, we have presented a high-level API for easily expressing parallel programs in Java. Our implementation compiles at runtime the Java bytcode to OpenCL and executes it on GPUs. We presented optimizations techniques which reduce the overhead introduced by Java and are applied automatically inside our compiler and runtime system.

We evaluated our approach using five benchmarks from different application domains. Our runtime experiments show, that using the GPU speedups of over 500× (including overheads) are possible with our approach. We show that our data management optimizations avoid the cost of marashalling and can substantially improve performance by up to 70x when executing on GPUs.

For future work, we plan to extend our API with more patterns, e.g., to enable nearest neighbor (a. k. a. stencil) computations. We plan to choose our patterns with care, because we want to keep the number of patterns small as they should act a set of basic building block. We want the users to leverage composition to build larger computational patterns from these building blocks.

Based on the presented infrastructure for executing efficient GPU applications in Java, we want to explore advanced memory optimizations such as usage of the fast local memory on GPUs from the high-level API.

Besides, we want to extend our API to support multi GPU execution and simultaneous execution in Java and on GPUs. We intend to exploit machine learning techniques for predicting for a given combination of application and hardware the best way to execute the application, using Java, a GPU, or both.

Acknowledgments

We would like to thank Oracle Labs for their support of this work. The authors would also like to thank the anonymous rewiers, as well as Thibaut Lutz, Alberto Magni and Andrew McLeod for fruitful discussions and their help regarding our implementation and benchmarks.

References

[1] AMD. Aparapi. http://aparapi.github.io/.

[2] AMD. Bolt C++ template library. http://github.com/HSA-Libraries/Bolt.

[3] Buono, D., Danelutto, M., Lametti, S., and Torquati, M. Parallel Patterns for General Purpose Many-Core. In *21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2013, Belfast, United Kingdom* (2013), IEEE Computer Society, pp. 131–139.

[4] Campa, S., Danelutto, M., Goli, M., González-Vélez, H., Popescu, A. M., and Torquati, M. Parallel Patterns for Heterogeneous CPU/GPU Architectures: Structured Parallelism from Cluster to Cloud. *Future Generation Comp. Syst. 37* (2014), 354–366.

[5] Catanzaro, B., Garland, M., and Keutzer, K. Copperhead: Compiling an Embedded Data Parallel Language. In *Proceedings*

of the 16th ACM Symposium on Principles and Practice of Parallel Programming (2011), PPoPP.

[6] CHAKRAVARTY, M. M., KELLER, G., LEE, S., MCDONELL, T. L., AND GROVER, V. Accelerating Haskell array codes with multicore GPUs. In *Proc. of the 6th workshop on Declarative Aspects of Multicore Programming* (2011), DAMP.

[7] CLAESSEN, K., SHEERAN, M., AND SVENSSON, J. Obsidian: GPU Programming in Haskell. In *In Proc. of 20th International Symposium on the Implementation and Application of Functional Languages* (2008), IFL.

[8] COLE, M. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, MA, USA, 1991.

[9] COLLINS, A., GREWE, D., GROVER, V., LEE, S., AND SUSNEA, A. NOVA: A functional language for data parallelism. In *Proceedings of the 2014 ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming* (2014), ARRAY.

[10] Nvidia CUDA, 2015. http://developer.nvidia.com/.

[11] CUNNINGHAM, D., BORDAWEKAR, R., AND SARASWAT, V. GPU Programming in a High Level Language: Compiling X10 to CUDA. In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop* (2011), X10.

[12] DUBACH, C., CHENG, P., RABBAH, R., BACON, D. F., AND FINK, S. J. Compiling a High-level Language for GPUs: (via Language Support for Architectures and Compilers). In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (2012), PLDI.

[13] DUBOSCQ, G., WÜRTHINGER, T., STADLER, L., WIMMER, C., SIMON, D., AND MÖSSENBÖCK, H. Graal IR: An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages* (New York, NY, USA, 2013), VMIL, ACM, pp. 1–10.

[14] ENMYREN, J., AND KESSLER, C. W. SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems. In *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications* (2010), HLPP.

[15] ERNSTING, S., AND KUCHEN, H. Algorithmic Skeletons for Multi-core, Multi-GPU Systems and Clusters. *IJHPCN 7*, 2 (2012), 129–138.

[16] FUMERO, J. J., STEUWER, M., AND DUBACH, C. 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* (2014), ARRAY.

[17] GONZÁLEZ-VÉLEZ, H., AND LEYTON, M. A Survey of Algorithmic Skeleton Frameworks: High-level Structured Parallel Programming Enablers. *Softw. Pract. Exper. 40*, 12 (Nov. 2010), 1135–1160.

[18] GRAAL. Oracle Graal VM. http://openjdk.java.net/projects/graal/.

[19] GUO, J., RODRIGUES, W., THIYAGALINGAM, J., GUYOMARC'H, F., BOULET, P., AND SCHOLZ, S.-B. Harnessing the Power of GPUs without Losing Abstractions in SAC and ArrayOL: A Comparative Study. In *Proceedings of the IPDPS 2011 Workshop on High-Level Parallel Programming Models and Supportive Environments* (2011), pp. 1183–1190.

[20] HAN, T. D., AND ABDELRAHMAN, T. S. hiCUDA: High-level GPGPU programming. *IEEE Trans. Parallel Distrib. Syst. 22*, 1 (Jan. 2011).

[21] HOBEROCK, J., AND BELL, N. Thrust: A Parallel Template Library. http://developer.nvidia.com/thrust.

[22] Java bindings for OpenCL, 2015. http://javacl.googlecode.com.

[23] Java bindings for OpenCL. http://www.jocl.org/.

[24] KUCHEN, H., AND ERNSTING, S. Data Parallel Skeletons in Java. In *Proceedings of the International Conference on Computational Science, ICCS 2012, Omaha, Nebraska, USA, 4-6 June, 2012* (2012), H. H. Ali, Y. Shi, D. Khazanchi, M. Lees, G. D. van Albada, J. Dongarra, and P. M. A. Sloot, Eds., vol. 9 of *Procedia Computer Science*, Elsevier, pp. 1817–1826.

[25] LEE, S., MIN, S.-J., AND EIGENMANN, R. OpenMP to GPGPU: a Compiler Framework for Automatic Translation and Optimization. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming* (1999), PPoPP.

[26] LEYTON, M., AND PIQUER, J. M. Skandium: Multi-core Programming with Algorithmic Skeletons. In *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, PDP 2010, Pisa, Italy, February 17-19, 2010* (2010), M. Danelutto, J. Bourgeois, and T. Gross, Eds., IEEE Computer Society, pp. 289–296.

[27] MCCOOL, M., ROBISON, A. D., AND REINDERS, J. *Structured Parallel Programming*. Morgan Kaufmann, 2012.

[28] MUELLER, F., AND ZHANG, Y. HIDP: A Hierarchical Data Parallel Language. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (2013), CGO.

[29] OpenACC. http://www.openacc-standard.org/.

[30] OpenCL. http://www.khronos.org/opencl/.

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

[32] OpenMP 4.0. http://openmp.org/wp/.

[33] PRATT-SZELIGA, P. C., FAWCETT, J. W., AND WELCH, R. D. Rootbeer: Seamlessly Using GPUs from Java. In *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on* (2012), G. Min, J. Hu, L. C. Liu, L. T. Yang, S. Seelam, and L. Lefevre, Eds., HPCC-ICESS.

[34] REYES, R., LÓPEZ-RODRÍGUEZ, I., FUMERO, J. J., AND DE SANDE, F. accULL: An OpenACC Implementation with CUDA and OpenCL Support. In *European Conference on Parallel Processing* (2012), Euro-Par.

[35] STEUWER, M., KEGEL, P., AND GORLATCH, S. Skelcl - A portable skeleton library for high-level GPU programming. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May 2011 - Workshop Proceedings* (2011), IEEE, pp. 1176–1182.

[36] TARDIEU, O., HERTA, B., CUNNINGHAM, D., GROVE, D., KAMBADUR, P., SARASWAT, V., SHINNAR, A., TAKEUCHI, M., AND VAZIRI, M. X10 and APGAS at Petascale. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2014), PPoPP.