A Composable Array Function Interface for Heterogeneous Computing in Java

Juan José Fumero
University of Edinburgh, UK
juan.fumero@ed.ac.uk

Michel Steuwer
University of Münster, Germany
michel.steuwer@uni-muenster.de

Christophe Dubach
University of Edinburgh, UK
christophe.dubach@ed.ac.uk

Abstract

Heterogeneous computing has now become mainstream with virtually every desktop machines featuring accelerators such as Graphics Processing Units (GPUs). While heterogeneity offers the promise of high-performance and high-efficiency, it comes at the cost of huge programming difficulties. Languages and interfaces for programming such system tend to be low-level and require expert knowledge of the hardware in order to achieve its potential.

A promising approach for programming such heterogeneous systems is the use of array programming. This style of programming relies on well known parallel patterns that can be easily translated into GPU or other accelerator code. However, only little work has been done on integrating such concepts in mainstream languages such as Java.

In this work, we propose a new Array Function interface implemented with the new features from Java 8. While similar in spirit to the new Stream API of Java, our API follows a different design based on reusability and composability. We demonstrate that this API can be used to generate OpenCL code for a simple application. We present encouraging preliminary performance results showing the potential of our approach.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Patterns; D.3.4 [*Processors*]: Code generation, Compilers

Keywords Array programming, Patterns, GPGPU

1. Introduction

Computer systems have become increasingly complex and heterogeneous in recent years. The introduction of multicore processors and the widespread use of Graphics Processing Units (GPUs) for general purpose computations imposes huge challenges on programmers.

Several programming models have been introduced for writing code for GPUs, like CUDA [6] and OpenCL [18]. Writing efficient parallel code using these models requires a deep understanding and intimate knowledge of the underlying hardware. Multiple research projects aim at simplifying programming of parallel heterogeneous systems in general and GPUs in particular. For instance OpenACC [17] simplifies programming of GPUs by allowing the programmer to annotate loops to be executed in parallel on a GPU but this task remains relatively low-level. Higher level abstractions are necessary to enable non expert programmer to make use of these modern parallel computer systems.

A better approach consists of using structured parallel programming [15] where common patterns are used to easily express algorithmic ideas as parallel operations on arrays. This style of programming is heavily influenced by functional programming and has been successfully applied to practical use, e.g., in Google's MapReduce framework [13]. Structured parallel programming helps simplifying the programming of parallel hardware and – at the same time – enable the compiler to produce efficient parallel code by exploiting the semantic information embedded in the pattern. This idea is not new and has already been applied to GPUs by high-level libraries like Thrust [11] or SkelCL [20] as well as in functional languages like Haskell [4], SAC [9], or NVIDIA's NOVA language [5]. This style of programming is also the foundation of languages like StreamIt [21] or Lime [7] where the application is expressed as a stream of computation.

Most projects have targeted either functional programming languages or languages common in the high performance computing community like C or C++. Only little work has been done to integrate these ideas for programming GPUs into existing mainstream object oriented languages like Java. The challenge is to design a system which combines the familiarity of the object oriented language with the power and flexibility of parallel patterns to enable Java programmers to easily and efficiently use GPUs.

Java 8 introduces new features which support a more functional programming style such as lambda expressions. In addition, the new Java 8 Stream API allows for writing applications in an array programming style, which is definitively a step in the right direction. In this paper we describe our initial design and implementation of an array based programming interface in Java which makes extensive use of the new features of Java 8. This paper makes the following contributions:

- Design of a new API for supporting array programming in Java;
- Development of an OpenCL code generator for our API.

The rest of the paper is organised as follows: Section 2 discusses some background of parallel array programming and introduces some of the new features of Java 8. Section 3 presents our new array programming interface and section 4 explains how we generate GPU code from Java and execute it. Section 5 evaluates our approach using an example application. Section 6 discusses related work and section 7 concludes the paper.

ARRAY'14, June 11 2014, Edinburgh, United Kingdom
Copyright is held by the owner/author(s). Publication rights licensed to ACM..
Copyright © 2014 ACM 978-1-1-4503-2937-8/14/06. . . $15.00.
http://dx.doi.org/10.1145/2627373.2627381

2. Background

Parallel Array Programming Parallel array programming is a style of programming where programs are expressed as a composition of *patterns* or *array functions* which perform parallel operations on arrays. These array functions are customised by the application programmer using so called *user functions* which are application specific. The most basic array function is *map* which applies a user function to each element of an input array. The semantics of map enable the processing of each element of the array in parallel. Other commonly known array functions include *reduce*, which performs a parallel reduction of an array based on its binary user function, or *scan*, which can be used to compute a parallel prefix-sum. These array function are well known operations in functional programming languages like Haskell, SAC or Scala but are not necessary implemented in a parallel way. Patterns such as map can also be implemented in other programming languages such as Cilk Plus, Intel TBB, Intel ARB, and OpenCL [15].

Java GPU Programming Our goal is to offer an easy way to exploit external devices such as GPUs directly from Java and abstract away the details of the hardware. Accessing a GPU from Java can be done using low-level bindings for CUDA or OpenCL [12] which requires to write a low-level OpenCL kernel. Aparapi [1] and Rootbeer [19] are two libraries that provides a high-level abstraction by allowing the user to write the kernel directly in Java. However, this style of programming is still low-level since the hardware features are directly exposed to the programmer at the Java level.

Stream API and Functional Interface Project Sumatra [2] is a new approach based on new features of Java 8; the Stream API and supports for lambdas. It lets the user express the program with the new Stream API and automatically generates HSAIL low-level code for AMD GPUs using Graal [8]. The Stream API implements an array programming model where the user expresses the application as a series of stream operations. These stream operations can take a function as an argument represented by the Function interface in Java 8 which can be expressed with a user function or lambda expression. Streams are always created from a collection (arrays for instance) and only executes once a so called terminal operator is encountered (e.g. reduction or forEach). This design means that it is not possible to reuse stream once created as they are tied to a particular collection.

Array Function Interface This paper proposes a new design of an array programming interface for Java, named ArrayFunction API, that lifts some of the limitations of the Stream API. All of our array functions extend the Java function interface, allowing them be easily composed and passed as input to other array functions. In addition, we do not rely on terminal operations to start the computation which means that any composed array function is reusable. Finally, we have designed our implementation in such a way that any array function can be run on an OpenCL device (e.g. a GPU) automatically (although we currently only support the map function).

3. Array Programming in Java

We now present our ArrayFunction API that enables array programming in Java. We first present an example to show how Java programmers use our API, afterwards we discuss our API design.

3.1 Example: Dot product

We use the dot product computation of two vectors as a simple example. The dot product is computed by first multiplying the two vectors pairwise and then sum up all the intermediate results. Listing 1 show the implementation in our ArrayFunction API. As can be seen we first use the zip2 ArrayFunction of our API to pairwise combine the two input arrays together (line 3). The <Integer ,Interger> type parameters specifies that the input should be two arrays of Integer. The output type of the zip2 pattern is automatically inferred as an array of Tuple2<Integer, Integer>. Then, we map the multiplication operation with a lambda expression (x -> x._1 * x._2) to every pair of Integers which are represented as tuples (line 4). Finally, the reduce pattern is used with a lambda expression to sum up all the elements (line 5). Once the function representing the dot product computation has been built, the user can simply run it on an input by calling the apply function which expects two arrays of Integer in the form of a Tuple2 (line 10).

```
1 ArrayFunction dotProduct =
2   ArrayFunction.<Integer,Integer>zip2()
3                  .map(x -> x._1 * x._2)
4                  .reduce(x,y -> x+y);
5
6 Integer[] in1 = new Integer[size];
7 Integer[] in2 = new Integer[size];
8 Integer[] out =
9   dotProduct.apply(new Tuple2(in1, in2));
```

Listing 1. Dot product Java code with our ArrayFunction API

This example illustrates how we can chain patterns easily with our API. In addition, the expression is strongly typed as the type checker automatically infers and verifies at each step that all the types match.

3.2 Class Hierarchy

Our design is inspired by the Stream API and uses lambda expressions which are part of Java 8. However, in contrast to the Stream API, we implemented all array operations as functions which inherit from the same functional interface. This enables the creation of reusable composed functions – as seen in the example – which is fundamentally different from the Stream API where a stream is tightly coupled with its input data. In addition, this allows us to nest functions by passing an array function as input to other array functions such as map.

The UML diagram representing our array function class hierarchy is shown in figure 1. The symbols before the signature of each method in the UML diagram indicates the visibility of each method. The symbol ”+” indicates a method or field is public whereas "-" indicates it is private. In the case of the classes, the middle section represents the fields while the bottom part represents the methods. The parametric type (or generics) representation is similar to the one used in Java.

We decided to build our class hierarchy starting with the Function interface which is part of Java 8. This Function interface is used to represent unary functions and lambda expressions. Since Java 8 does not support variadic generics (i.e. a variable number of type parameter), we implemented several Tuple classes (1 to 8 currently) to be able to support functions with multiple arguments such as Zip. Consequently, we have to also duplicates several method dealing with the zip function as we will see. The addition of variadic generics in Java would certainly be a welcome addition.

The ArrayFunction interface extends the Function interface and is intended to be used as the top-level interface by the programmer. As its name suggests, this interface represents functions that are applied on arrays. The input elements are of type inT while the output elements are of type ouT. As can be seen this corresponds to the inT[] and ouT[] parameter type of the Function interface. The benefit of extending the Function interface to implement our array

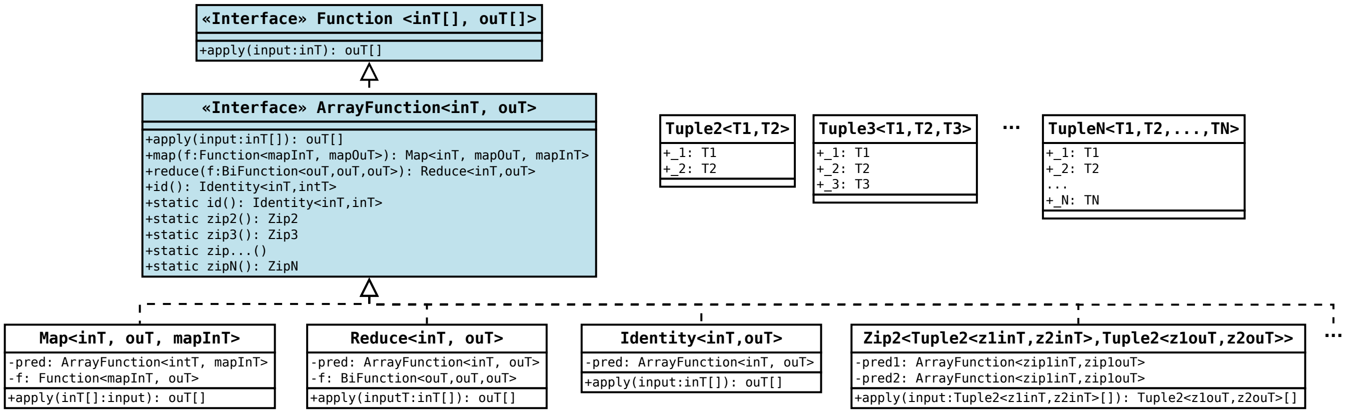
«Interface» Function <inT[], ouT[]>
+apply(input:inT): ouT[]
«Interface» ArrayFunction<inT, ouT>
+apply(input:inT[]): ouT[]
+map(f:Function<mapInT, mapOuT>): Map<inT, mapOuT, mapInT>
+reduce(f:BiFunction<ouT,ouT,ouT>): Reduce<inT,ouT>
+id(): Identity<inT,intT>
+static id(): Identity<inT,inT>
+static zip2(): Zip2
+static zip3(): Zip3
+static zip...()
+static zipN(): ZipN
Tuple2<T1,T2>
+_1: T1
+_2: T2
Tuple3<T1,T2,T3>
+_1: T1
+_2: T2
+_3: T3
...
TupleN<T1,T2,...,TN>
+_1: T1
+_2: T2
...
+_N: TN
Map<inT, ouT, mapInT>
-pred: ArrayFunction<intT, mapInT>
-f: Function<mapInT, ouT>
+apply(inT[]:input): ouT[]
Reduce<inT, ouT>
-pred: ArrayFunction<inT, ouT>
-f: BiFunction<ouT,ouT,ouT>
+apply(inputT:inT[]): ouT[]
Identity<inT,ouT>
-pred: ArrayFunction<inT, ouT>
+apply(input:inT[]): ouT[]
Zip2<Tuple2<z1inT,z2inT>,Tuple2<z1ouT,z2ouT>>
-pred1: ArrayFunction<zip1inT,zip1ouT>
-pred2: ArrayFunction<zip1inT,zip1ouT>
+apply(input:Tuple2<z1inT,z2inT>[]): Tuple2<z1ouT,z2ouT>[]
...

Figure 1. Simplified view of the class hierarchy of our ArrayFunction API

functions is that it is always possible to pass an array function as an input to another array function like map.

The ArrayFunction interface comprises several methods corresponding to the different available functions operating on arrays. The set of methods is currently limited to map, reduce, id and zip but we plan to add more functions in the future to support operations such as flattening, partitioning or stencil computation. These methods create ArrayFunction subclasses that implement the different operations. For instance the map method, which takes a function f that is applied to each element of the input array, returns a Map ArrayFunction object. This design allows us to chain array functions with one another easily.

3.3 Array Functions Composition

The creation of a new ArrayFunction starts with either the static id() method or one of the different static zip() methods. The id() method permits the creation of functions that takes one input array whereas the zip() methods enable the creation of functions that operate on multiple input arrays at once. Once created, it is possible to easily compose array functions together by simply calling one of the array function methods from the ArrayFunction interface. We have taken special care to ensure that the results of composing functions remain strongly typed.

To achieve strong typing, each class implementing the ArrayFunction interface is responsible for storing a reference to the ArrayFunction that created it in the first place. This is done by having a predecessor (pred) field in each class which is initialised at construction time when one of the map, reduce, id or zip methods are called in the ArrayFunction interface. The type information of the predecessor function is recorded in the predecessor field to enable static type checking and propagate the input type inT of the composed function. Note that the inT type represents the input type of the whole composed function and not the input type of the specific array operator (map, reduce, ...) being applied. For example, in case of the Map subclass, the type of the predecessor is <intT, mapInT> where intT is the input element type and mapInT the output element type of the predecessor which corresponds to the input type of the "mapped" function f. The ouT type, which is the output type of the "mapped" function f will become the new element output type of the newly created Map ArrayFunction.

3.4 Array Function Execution

Once an array function has been created, we can execute it by calling the apply method which takes an array as an input of the element type of the ArrayFunction. In our design, the apply can either execute the computation in pure Java or execute the code on an OpenCL device like a GPU. The decision is currently made with the use of a global variable that controls the behaviour of all array functions. Our long term goal is to automatically determine where to execute the function based on the type of device available, input size and other characteristics.

```
1  public outT[] apply(inT[] input) {
2    mapInT[] predResult = pred.apply(in);
3    ouT[] result = new ouT[predResult.length];
4    if (JavaExecution)
5      for (int i=0; i<predResult.length; i++)
6        result[i] = f.apply(predResult[i]);
7    else
8      // OpenCL device execution
9    return result;
10 }
```

Listing 2. Implementation (pseudo-Java code) of the apply method of the Map class.

The apply method is implemented in each of the ArrayFunction subclasses using a similar mechanism. An example for the Map class is shown in listing 2. First, we execute the apply method of the predecessor array function (line 2). Then, we create the result array for this current array function (line 3). If the ArrayFunction interface is configured for Java execution we apply the "mapped" function f to each element of the input array (line 4-6). Otherwise, we execute the function on an OpenCL device (line 7-8) the details of which will be discussed in the next section.

4. OpenCL Compilation and Execution

This section describes the process leading to the execution of our array functions on an OpenCL parallel device, like a GPU. Our code generator is written in pure Java using the Graal API [8] which exposes an interface to the JavaVM JIT compiler. We currently only support the map ArrayFunction in our code generator which means that all the other ArrayFunctions are executed in pure Java. We intend to lift this limitation in the future by adding supports for all array function in our code generator.

When we wish to execute an array function on an OpenCL device, the following events take place at runtime: the code generator produces the OpenCL kernel code which is immediately compiled to binary code by the OpenCL vendor compiler; the data is marshalled from Java to C via the Java Native Interface (JNI) and is transferred onto the device; finally the computation is executed on the device and the data is returned to Java via JNI. Note that we im-

```
1  typedef struct { int _1; int _2; } tuple2_t;
2
3  int f(tuple2_t in) { return in_1 * in_2; }
4
5  kernel void map(global tuple2_t* in, uint n,
6                  global int* out) {
7    uint gid = get_global_id(0);
8    uint gsz = get_global_size(0);
9    for (uint i=0; i+gid<n; i+= gsz)
10     out[i+gid] = f(in[i+gid]._1, in[i+gid]._2);
11 }
```

Listing 3. Vector multiplication generated OpenCL code

```
1  ArrayFunction bs =
2    ArrayFunction.<Float>id().map(
3      stockPrice -> {
4        float call = computeCallOption(...);
5        float put  = computePutOption(...);
6        return new Tuple2<>(call, put);
7      });
8
9  Tuple2<Float, Float>[] result =
10   bs.apply(stockPrices);
```

Listing 4. Implementation of the Black-Scholes Model using the `ArrayFunction` API

plemented a cache for already generated kernels so that we do not need to re-generate and re-compile an OpenCL kernel next time the array function is executed.

4.1 OpenCL Code Generation

The first stage before being able to execute an array function of our API on an OpenCL device consists of generating OpenCL code. As this happens at runtime we can access the internal runtime representation of the program.

Function Code Generation To produce an OpenCL kernel for the map array function, we use a template approach that generates a for loop that distributes work among the different global threads. In the body of the loop the user provided function is called on every element of the input array and the results are stored in the output array. The OpenCL code for the user function, which is written in Java, is generated by a visitor that traverses the high-level nodes of the control flow and data flow graph and emits corresponding OpenCL code. If an error is detected at runtime, the code generator throws an exception which our library catches. At this moment the execution is switched from the OpenCL to pure Java code. In this way, the user code is guaranteed to always be executed. We currently only support a basic subset of the Java language but we are planning to extend our work to cover more advanced Java features in the future (e.g. instance method calls, Object creation).

Data Structure Generation Our code generator currently supports primitive types, arrays of primitive and tuples. In order to ensure high performance in the code generated, all the class types are automatically converted to primitive types whenever possible. For instance, `Integer[]` will be converted during code generation to `int[]`. We also automatically flatten multi-dimensional arrays by creating an index array and an array of values. Tuples are implicitly known to the backend and converted to equivalent OpenCL/C structs. We intend to extend our backend in the near future to deal with arbitrary Objects.

Example Listing 3 shows the OpenCL code generated for the `map(x -> x._1+x._2)` array function from the dot product example (listing 1). The first line is the data structure declaration corresponding to the type of `x` (`Tuple2<Integer, Integer>`). For the lambda expression the function f is generated (line 3) which performs the multiplication. The kernel code starts on line 5 and represents the whole array function which takes an array of tuples as an input and produces an array of int. The for loop on lines 9-10 maps each thread to one or more elements of the input array and calls the function f.

4.2 OpenCL Execution

Once the kernel has been generated, we execute it on the OpenCL device using the JOCL [12] OpenCL Java Bindings. The first step consists of preparing (a.k.a. marshalling) the data for the execution on the OpenCL device. This involves converting the various Java data types to primitive types, flattening the array and converting the tuples. The Tuple classes are lowered to structs using the Struct class from JOCL.

Once the marshalling has taken place, we can send the data to the device, execute the kernel and start the unmarshalling process. This last step converts back the low-level data structures into Java classes. While these steps currently executes synchronously in our implementation, we plan to overlap data marshalling, transfer and execution in the future to reduce the overhead.

All possible errors from the JOCL binding are catched. In the same way as in the code generator, if an error is detected, a deoptimisation is performed and the pure Java code is executed.

5. Evaluation

We evaluate our library in a workstation with an AMD Radeon HD 7970 GPU running OpenCL 1.2. The driver version (1348.5) is the last one available at the moment of writing this paper. For the code generator, we have extended the Graal compiler with a new backend for OpenCL. Graal is compiled with the latest version of Java 8 (JDK 1.8.0).

Expressing Black-Scholes with the Array Function API For evaluating our approach we implemented the Black-Scholes model, which is an application from the field of financial mathematics. Given an array of stock prices the Black-Scholes model computes for each stock price a pair of call and put options. Listing 4 shows how this problem can be implemented with our `ArrayFunction` API. The `ArrayFunction` computing the Black-Scholes model is defined in lines 1 – 7 using the map array function: for a given stock price (line 3) the call and put options are computed (line 4 and 5). The map then returns a pair of these two values represented as a `Tuple2` (line 6). Finally, the computation is applied to the array of stock prices and produces an array containing the call and put options (line 9 and 10).

Runtime Experiments Figure 2 shows a comparison between the runtime of the Black-Scholes application implemented using our API and pure Java. The benchmark is run with `bootStraping` enabled in Graal. This means that the Graal-jit compiler (Graal is written in Java) first compiles itself before running the application. We run the application 100 times per size and we record the median time.

The first bar show the execution time using the `ArrayFunction` API, the second and third bar show two different Java implementations: in one version the floating point values are represented as arrays of objects (`Float[]`) and in the other one as array of primitives (`float[]`).

We can draw two important conclusions from this comparison. First, our API introduces almost no overhead in comparison to a pure Java version using low-level loops to process the data;

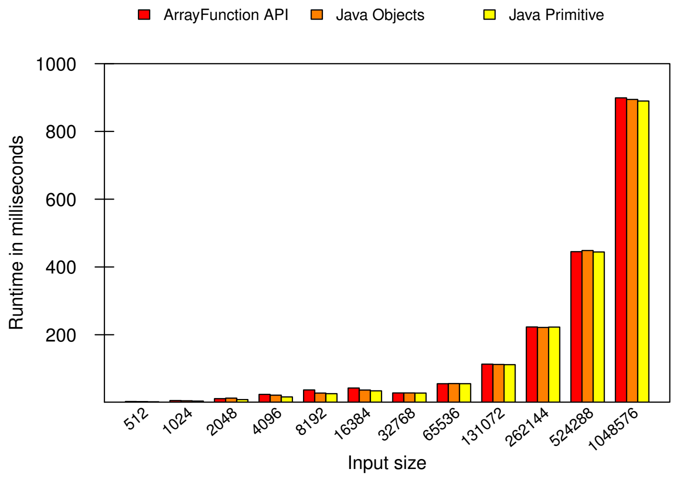

Figure 2. Runtime comparison of the Black-Scholes application implemented with `ArrayFunction` API with Java using objects and primitive types.

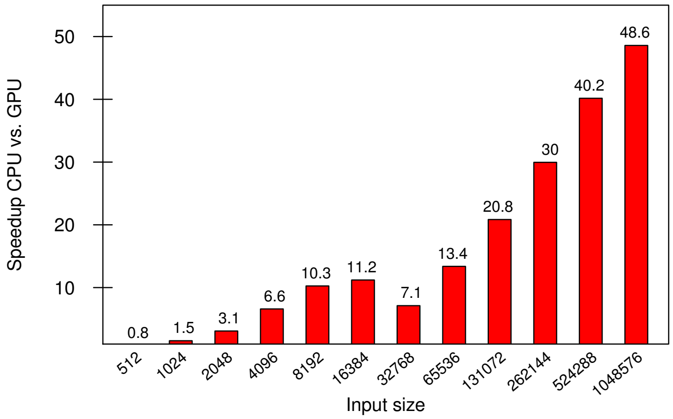

Figure 3. Speedup of overall execution time in GPU over CPU.

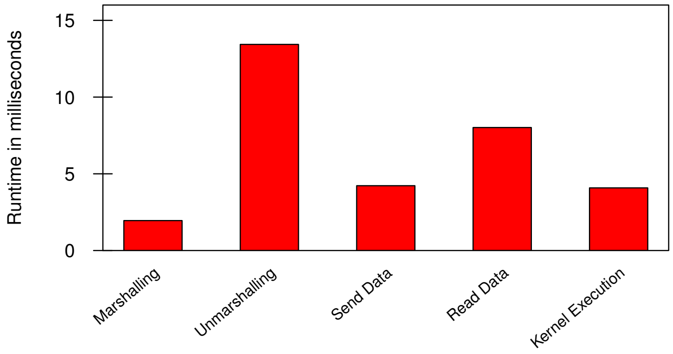

Figure 4. Breakdown of the GPU execution time for an input size of 1048576 elements.

Secondly, for our application the difference between working with objects or primitive types is small. This is possibly due to the computational nature of the Black-Scholes application which reads a data item once and then performs several computations on it. Therefore, the conversion between object and primitive type only happens once per data item and is hidden by the computation time. For this kind of applications our `ArrayFunction` API is competitive with primitive-based Java implementation.

Speedup The Figure 3 shows the speedup when our implementation is executed on the GPU as compared to the execution on the CPU.

The runtime for the GPU version, includes the times for the steps described in section 4 like the preparation and transfer of the input data. The time for generating and building the OpenCL kernel is not included since the kernel is read from the kernel cache we implemented. Furthermore, the generation and compilation of the OpenCL kernel can be hidden when being performed concurrently to the execution of other code as is the case with most modern JIT compiler. As can be seen, the version using the `ArrayFunction` API on the GPU version outperforms the Java versions. With increasing data sizes the advantage of the GPU version increases as compared to the CPU. The slight drop at 32768 elements is due to the Java version being jit'ed as the code becomes hotter with larger input sizes.

Breakdown of the GPU execution time Figure 4 shows a breakdown of the execution time when executing the Black-Scholes application on the GPU. The Figure does not contain the kernel generation and build time, which takes between 100 and 200 milliseconds in the first iteration. The runtime for marshalling and unmarshalling stage are introduced by the transformation between objects (structs in OpenCL to Tuple2 Objects in Java). Transferring the data to and from the GPU also takes some time. The actual execution only takes a fraction of the overall runtime. Nevertheless, as the impressive speedup in figure 3 shows, its still beneficial to use the GPU for this application. Future work will focus on reducing the overhead by overlapping marshalling and data transfer with kernel execution.

6. Related Work

Array Programming for GPUs A number of new programming languages have been proposed which generate GPU code based on array functions. Copperhead [3] for instance is a data parallel programming language by NVIDIA integrated in Python. HiDP [16] is a parallel programming language which uses patterns to hierarchical structure data parallel computations. NOVA [5] is a functional programming language developed by NVIDIA offering parallel patterns as primitive in the language to express parallelism. StreamIt [21] is an architect independent language for stream programming which can be used to generate CUDA code. Lime [7] is a programming language developed by IBM that extends Java with concepts similar to StreamIt and can generate OpenCL code.

In contrast to our API all these approaches require programmers to learn a new programming language which arguably limits their adoption.

High-level Library-based Approach Several high-level pattern based programming libraries have been implemented in popular programming languages. Thrust [11] is a C++ library developed by NVIDIA which offers a set of customisable patterns to simplify GPU programming. SkelCL [20] allows to program multi-GPU systems using high-level patterns in C++.

Java Specification Resquest (JSR) 166 [14] is a framework based on the fork/join programming model for parallel computation in Java. The JSR-166 contains a ParallelArray class that provides parallel operations such as map, reduce, select or apply in a set of elements. Similarly, River Trail [10] is a JavaScript library which exposes parallel computation with map and reduce among others. River Trail has its own data structure to represent parallel arrays.

We differ to these approaches as we build the computation pipeline independently of the data input, thus allowing multiple computation in the pipeline with different data sets.

GPU Computing in Java We are not the first to enable GPU computing in Java. There exists several bindings to OpenCL like JOCL [12] which require the user to write the function to be executed on the GPU to be written in OpenCL. Both Rootbeer [19] and Aparapi [1], which is an project by AMD, convert Java bytecode to OpenCL at runtime and, thus, allow for writing all the code in Java. Nevertheless, these projects are low-level as they require the programmer to explicitly exploit the parallelism and do not raise the level of abstraction like our approach does.

7. Conclusion and future work

In this paper we presented our work in progress on the `ArrayFunction` API which aims to be a simple and elegant array programming interface in Java. As seen, computations can be transparently executed on GPUs for the map function by generating OpenCL code from Java code using Graal. Our API design combines high-level functional aspects with object oriented design principles and, thus, integrates well into the new Java 8. By introducing a single interface representing all functions we enable reusability and composability in our API in form of chaining and nesting of functions. Preliminary experiments show that we can correctly generate and execute code for the Blacks-Scholes application.

Our API currently lacks several important features that we plan to implement in the future. To start, we will add more array patterns such as partitioning, flattening, stencil computation, ordering and filtering. We will also add supports for Collections to allow a richer set of applications to be expressed. Currently, only the map pattern can be compiled to OpenCL. We intend to add kernel generation for the other patterns and optimisations such as fusing patterns or overlapping of computation and communication. At the moment the decision of where to run each array function is made by setting a global parameter which is far from ideal for a dynamic system. To alleviate this problem, we will develop a runtime model that decides where each array function should be run. Our implementations also lacks proper handling of exception in OpenCL at runtime. Currently, we only support exception raised by our backend (e.g. unsupported features) or by the OpenCL runtime (e.g. device unavailable). In such cases, we fall back automatically to the pure Java implementations. We plan to introduce proper Java exception handling at runtime by adding a mechanism to stop the computation and fall-back to the sequential Java implementation for accurate exception handling.

Acknowledgments

We want to thanks Ranjeet Singh for the initial implementation of the OpenCL backend in Graal. This work was supported by a grant from Oracle Labs' External Research Office.

References

[1] Aparapi. `https://code.google.com/p/aparapi/`.

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

[3] 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*, PPoPP '11, pages 47–56, New York, NY, USA, 2011. ACM.

[4] M. M. T. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the POPL 2011 Workshop on Declarative Aspects of Multicore Programming*, pages 3–14, 2011.

[5] A. Collins, D. Grewe, V. Grover, S. Lee, and A. Susnea. NOVA: A functional language for data parallelism. Nvidia, 2013. Technical Report.

[6] CUDA. Nvidia CUDA. `http://developer.nvidia.com/`.

[7] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink. Compiling a high-level language for gpus: (via language support for architectures and compilers). In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 1–12, New York, NY, USA, 2012. ACM.

[8] G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck. 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*, VMIL '13, pages 1–10, New York, NY, USA, 2013. ACM.

[9] J. Guo, W. Rodrigues, J. Thiyagalingam, F. Guyomarc'h, P. Boulet, and S.-B. Scholz. 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*, pages 1183–1190, 2011.

[10] 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 &#38; Applications*, OOPSLA '13, pages 729–744, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2374-1. . URL `http://doi.acm.org/10.1145/2509136.2509516`.

[11] J. Hoberock and N. Bell. Thrust: A parallel template library. `http://developer.nvidia.com/thrust`.

[12] JOCL. Java bindings for OpenCL. `http://www.jocl.org/`.

[13] R. Lämmel. Google's MapReduce programming model – revisited. *Science of Computer Programming*, 70(1):1 – 30, 2008.

[14] D. Lea. The java.util.concurrent synchronizer framework. *Sci. Comput. Program.*, 58(3):293–309, Dec. 2005. ISSN 0167-6423. . URL `http://dx.doi.org/10.1016/j.scico.2005.03.007`.

[15] M. McCool, A. D. Robison, and J. Reinders. *Structured Parallel Programming*. Morgan Kaufmann, 2012.

[16] F. Mueller and Y. Zhang. Hidp: A hierarchical data parallel language. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, Washington, DC, USA, 2013. IEEE Computer Society.

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

[18] OpenCL. Opencl. `http://www.khronos.org/opencl/`.

[19] P. C. Pratt-Szeliga, J. W. Fawcett, and R. D. Welch. Rootbeer: Seamlessly using GPUs from java. In G. Min, J. Hu, L. C. Liu, L. T. Yang, S. Seelam, and L. Lefevre, editors, *HPCC-ICESS*, 2012.

[20] M. Steuwer and S. Gorlatch. SkelCL: Enhancing OpenCL for high-level programming of multi-GPU systems. In *Parallel Computing Technologies*, volume 7979 of *Lecture Notes in Computer Science*, pages 258–272. Springer, 2013.

[21] W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 365–376, New York, NY, USA, 2010. ACM.