Just-In-Time GPU Compilation for Interpreted Languages with Partial Evaluation

Juan Fumero[†] Michel Steuwer[†] Lukas Stadler[*] Christophe Dubach[†]

[†]The University of Edinburgh, [*]Oracle Labs, AT

juan.fumero@ed.ac.uk michel.steuwer@ed.ac.uk lukas.stadler@oracle.com christophe.dubach@ed.ac.uk

Abstract

Computer systems are increasingly featuring powerful parallel devices with the advent of many-core CPUs and GPUs. This offers the opportunity to solve computationally-intensive problems at a fraction of the time traditional CPUs need. However, exploiting heterogeneous hardware requires the use of low-level programming language approaches such as OpenCL, which is incredibly challenging, even for advanced programmers.

On the application side, interpreted dynamic languages are increasingly becoming popular in many domains due to their simplicity, expressiveness and flexibility. However, this creates a wide gap between the high-level abstractions offered to programmers and the low-level hardware-specific interface. Currently, programmers must rely on high performance libraries or they are forced to write parts of their application in a low-level language like OpenCL. Ideally, non-expert programmers should be able to exploit heterogeneous hardware directly from their interpreted dynamic languages.

In this paper, we present a technique to transparently and automatically offload computations from interpreted dynamic languages to heterogeneous devices. Using just-in-time compilation, we automatically generate OpenCL code at runtime which is specialized to the actual observed data types using profiling information. We demonstrate our technique using *R*, which is a popular interpreted dynamic language predominately used in big data analytic. Our experimental results show the execution on a GPU yields speedups of over 150x compared to the sequential FastR implementation and the obtained performance is competitive with manually written GPU code. We also show that when taking into account start-up time, large speedups are achievable, even when the applications run for as little as a few seconds.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, contact the Owner/Author(s). Request permissions from permissions@acm.org or Publications Dept., ACM, Inc., fax +1 (212) 869-0481.

VEE '17, April 08 - 09, 2017, Xi'an, China
Copyright © 2017 held by owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4948-2/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3050748.3050761

1. Introduction

Nowadays, most computer systems are equipped with powerful parallel devices such as Graphics Processing Units (GPUs). Many application domains benefit by achieving orders of magnitude speedup over parallel CPU code. However, exploiting this hardware requires a deep knowledge of the architectures and low-level languages such as OpenCL, which is very challenging for non-expert programmers.

Many non-computer scientists prefer using interpreted languages such as Ruby, Python or R which are hugely popular despite their poor performance. They offer high-level functionality and simplicity of use, and the interpreter enables fast iterative software development. However, exploiting a GPU from these languages is far from trivial since programmers either have to write the GPU kernels themselves or rely on third-party GPU accelerated libraries.

Ideally, an interpreter for a dynamic programming language would be able to exploit the GPU automatically and transparently. A possible solution is to port the interpreter to the GPU and directly interpret the input program on the GPU. Unfortunately, this naïve solution is not practical since many parts of the interpreter are hard to port to a GPU such as method dispatch and object representation.

Partial evaluation has emerged as an important technique to improve interpreters' performance on CPUs [22, 23]. Such techniques specialize the interpreter to the application using Just-In-Time (JIT) compilation and profiling information. Using partial evaluation, the code produced becomes specialized to the actual observed data and to the hot path in the control-flow as in a trace-based compiler [6]. As a result, most of the interpreter code is compiled away, leaving only the actual application logic and computation.

In this work, we propose to extend these techniques for GPU code generation. We show how to produce OpenCL code at runtime for dynamic interpreted languages to accelerate programs with minimal effort for the language implementer. This is achieved using partial evaluation which specialized the program using profiling information. The specialized program becomes much easier to compile to the GPU since most high-level language constructs and interpreter code are removed away.

Using R as a use case, we extend the existing FastR [18] interpreter which is built upon the Truffle framework [23] and we use the Graal [2] JIT compiler to produce OpenCL code. The R interpreter is modified slightly to detect parallel operations and represent them as parallel nodes in the Abstract Syntax Tree (AST) interpreter. When a piece of R code is executed multiple times, the Truffle interpreter specializes the AST using profiling information and transforms it into the Graal Intermediate Representation (IR) using partial evaluation. The Graal IR is then intercepted and a series of passes is applied to simplify the IR as much as possible and the code generator attempts to produce an OpenCL GPU kernel. If the JIT compilation process fails due to unsupported features, controls return to the interpreter automatically and safely using the existing de-optimization process [7, 11, 23].

Our experimental results show that it is possible to accelerate R programs on the GPU automatically with large gains in performance. We achieve an average of 150x speed-up at peak performance when using the GPU compared to the runtime of the FastR interpreter on the CPU. Impressively, we still achieve an average of 57x speed-up in a realistic scenario where we include all startup time, OpenCL compilation time and OpenCL device initialization costs.

To summarize, the contributions of this paper are:

- We present an OpenCL JIT compiler for the FastR interpreter using Truffle and Graal.
- We present a technique for simplifying the Graal intermediate representation for OpenCL code generation.
- We demonstrate that this approach delivers high speedup on a set of R applications running on GPUs and it is compared with the standard de-facto GNU-R, FastR and OpenCL C++ implementations.

2. Background

Truffle [23] is a framework for implementing programming languages on top of a Java Virtual Machine (JVM). The Truffle API contains various building blocks for a language's runtime environment and provides infrastructure for managing executable code, mainly in the form of abstract syntax trees (ASTs).

2.1 AST Interpreters with Truffle

AST interpreters are a simple and straightforward technique to build an execution engine for a programming language. The source code is transformed into a tree of nodes, and the `execute` method of each node defines its behavior.

For dynamic programming languages, even seemingly simple operations such as adding two values can perform a multitude of different tasks depending on the input value types and the overall state of the runtime system. The Truffle AST nodes start out in an uninitialized state and replace themselves with specialized versions geared towards the specific input data that were encountered during execution. As new situations are encountered, the AST nodes are progressively made more generic to incorporate the handling of more inputs in their specialized behavior. At any point in time, the AST encodes the minimal amount of functionality needed to run the program with the inputs encountered so far. Most applications quickly reach a stable state where no new input types are discovered.

Writing nodes that specialize themselves involves a large amount of boilerplate code that is tedious to write and hard to get right under all circumstances. The Truffle DSL provides a small but powerful set of declarative annotations used to generate this code automatically.

2.2 Efficient JIT Compilation with Graal

The specialization of AST nodes together with the Truffle DSL allow the interpreter to run efficiently, e.g., to avoid boxing primitive values in certain situations. However, the inherent overhead of an interpreter which dispatches `execute` calls to the AST nodes cannot be removed.

To address this, Truffle employs Graal [23] for generating optimized machine code. Graal is a byte-code to native code JIT compiler implemented in Java, which can replace the client [12] and server [14] compilers in the HotSpot JVM. It transforms byte-code to the high-level GraalIR [2] intermediate representation, optimizes the IR, and transforms it to a low-level intermediate representation, before finally generating executable machine code for various platforms.

When Truffle detects that the number of times an AST was executed exceeds a certain threshold, it will submit the AST to Graal for compilation. Graal compiles the AST using partial evaluation [5], which essentially inlines all execute methods into one compilation unit and incorporates the current state of the AST to generate a piece of native code that works for all data types encountered so far. If new data types are encountered, the compiled code will deoptimize [7] and control will be transferred back to the interpreter which modifies the AST to accommodate the new data types. The AST is then recompiled with the additional functionality.

2.3 Example in *FastR*

Figure 1 shows a program executed using *FastR*, an implementation of the R language built using Truffle and Graal.

The program shown in the upper left corner executes a `function` for each pair of elements from vectors `a` and `b` using the `mapply` function. On execution, FastR creates the AST of the function passed to `mapply`, as shown on the left side of the figure. This AST is generic and not yet specialized to the input data used in the execution.

As it is executed, the AST rewrites itself to the state that is shown in the middle of Figure 1. The addition and multiplication nodes are specialized for the `double` data used in the example. The `AddDoubleNode` shown in the top right corner is the variant of the addition node specialized for `double` values. FastR calls the `execute` methods of the AST nodes inside the interpreter.

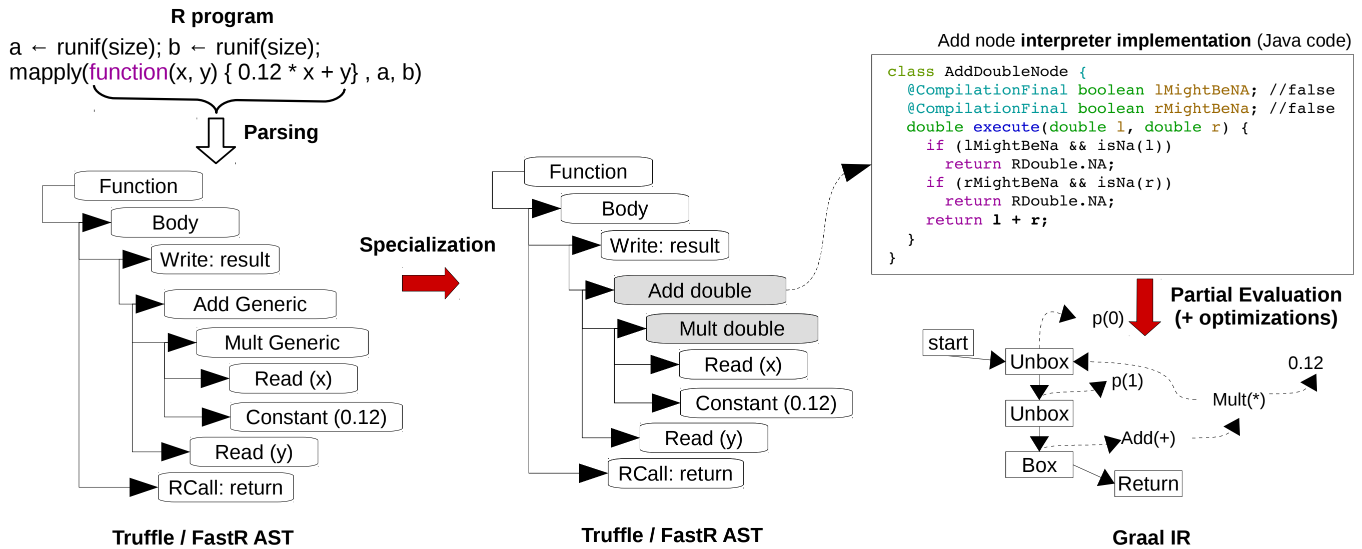

Figure 1: Execution of an R program in FastR via Truffle and Graal.

In order to respect the semantics of the R language, the `AddDoubleNode` needs to handle `NA` (not available) values. `NA` is a special marker that represents the absence of a value, e.g., a missing value inside a collection. The `execute` method of `AddDoubleNode` handles `NA`s by returning `RDdouble.NA` when one of the two operands is `NA` itself.

As an optimization, the infrastructure that performs arithmetic operations on vectors sets the `lMightBeNA` and `rMightBeNA` fields to `true` only if a vector that may contain `NA`s was encountered as an operand. As long as these flags remain `false`, no checks for `NA` values in the operands are necessary. Truffle uses *compiler directives* to convey this information to an optimizing runtime. The `CompilationFinal` directive is used in the example to indicate that the Boolean values can be assumed to be constant during compilation.

The partial evaluation performed by Graal transforms the FastR AST into the Graal IR shown on the lower right of Figure 1. Information specified in the compiler directives is used by the partial evaluation to perform optimizations. In the example, both branches visible in the node's `execute` method are removed based on the knowledge that the operands can never be `NA`. Therefore, partial evaluation optimizes the generated code by removing logic from the interpreter which is only required for exceptional cases.

3. OpenCL JIT Compiler for AST Interpreters

Figure 2 shows a system overview with dark gray boxes highlighting our contributions. Starting from the R application at the top, the FastR implementation parses the program and creates an AST where for parallel operations such as `mapply` special AST nodes are created.

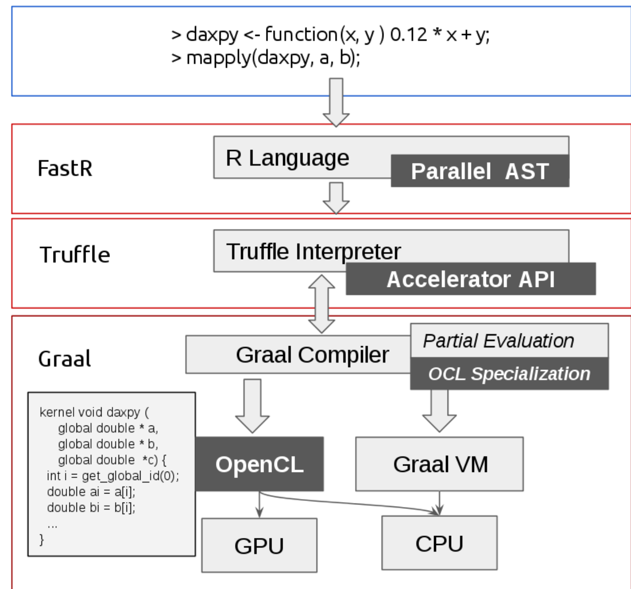

Figure 2: System Overview: Starting from R code our system transparently generates and executes OpenCL code via FastR which is built on top of Truffle and Graal.

The Truffle specialization process transforms the AST and eventually hands to code to Graal for compilation via partial evaluation. Our custom OpenCL-specialization passes remove unnecessary safety checks on a GPU. Finally, an OpenCL kernel is generated – shown in the bottom left corner – from the specialized Graal IR and is executed on a GPU via our OpenCL-enabled execution back-end integrated into Truffle.

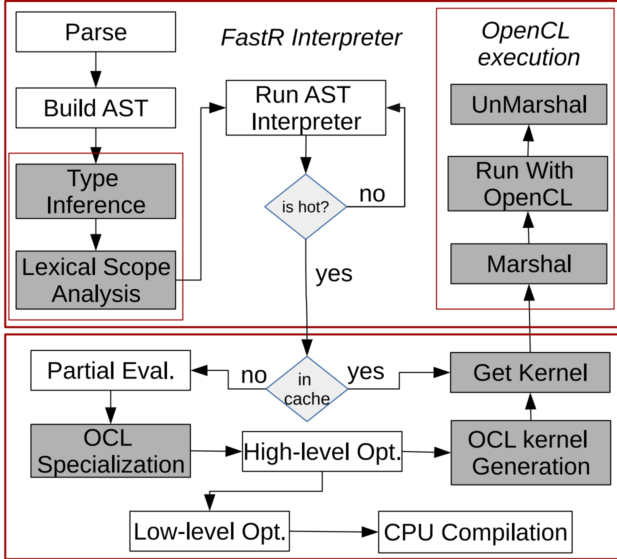

Figure 3: JIT Compiler from R interpreter to OpenCL C code. The white squares represent the existing components in FastR and Graal compilers. The gray squares represent our additions.

3.1 OpenCL JIT Compiler

In this section we discuss the changes we have made to the FastR interpreter for supporting the generation and execution of OpenCL code.

Compilation Flow Overview Figure 3 shows a work-flow of the FastR execution in combination with our extensions for running on GPUs. The unmodified FastR implementation performs the steps in the white boxes of: 1) parsing the R input program; 2) building an AST, similar to the one we have seen in Figure 1; 3) interpreting the nodes in the AST by running their `execute` methods; 4) when the code becomes hot due to running long enough in the interpreter it is marked for compilation; 5) the partial evaluator produces a control flow graph (CFG); 6) the CFG is optimized and then compiled by Graal to machine code before being executed.

Additions for OpenCL code generation To support the generation and execution of OpenCL kernels we provide a special implementation of the `mapply` R function and an extended FastR implementation with the additional steps indicated by the gray boxes in Figure 3. We will discuss these steps in details in the next sections.

3.2 Parallelizing `mapply` in R

We automatically parallelize R code which uses the `mapply` function. This function is widely used in data intensive R programs to apply a function to every element of an input data set. We exploit the fact that the order of execution is not specified to parallelize using our own OpenCL enabled AST node. Related work has looked into parallelizing this function using vectorization [21].

```
1  mapply <-function (FUN, ...) {
2    FUN <- match.fun(FUN)
3    if (fastR.oclEnabled())
4      return(.FastR(.NAME="mapplyOCL", FUN, ...))
5    #Default sequential implementation ...
6  }
```

Listing 1: Modification of the FastR `mapply` function to support OpenCL execution

Listing 1 shows the modification for the `mapply` implementation in FastR. Lines 3-4 check if an `OpenCL` driver is available and create a specialized built-in AST node that is able to generate OpenCL. If there is no OpenCL driver, we follow the default implementation.

3.3 Type Inference

We extend the FastR interpreter to infer the data types of the variables used in the function passed to our OpenCL enabled `mapply`. As OpenCL is a statically typed programming language we need to know the data types to generate an OpenCL kernel.

We currently support the most commonly used R data types: vectors of primitive data types for integer, double and logical values. R lists and sequences are handled specially, as we will discuss below. Collections of data might contain the special value `NA` (Not Available) for missing data.

Type inference for input and output values Truffle automatically infers the data types of the input to `mapply` using the same mechanism used for specialization. For OpenCL execution, we also have to check that the input values of a collection are homogeneous, i.e., all have the same type, and they do not contain `null` references.

To infer the output data types, we execute the R function in the interpreter by applying it only to the first element of the input. Based on this execution we obtain the type of the output data later used for generating the OpenCL kernel.

If Truffle fails to infer a unique type or if we detect `null` references, we fall back to the FastR execution on the CPU and do not generate an OpenCL kernel. Using this strategy, we ensure that the original R program is always executed.

Handling of R lists Lists in R are heterogeneously typed, i.e., different elements of a list can have different data types. We represent lists as `structs` in OpenCL to handle different types. This restricts lists to a fixed length, which is fine in our case, because at the time of partial evaluation we do know the length of arrays and lists.

Handling of NA values R is a language designed for statistics where it is very common to have missing values in a data set. FastR represents `NA` values by special bit patterns which depend on the data type. For instance, for integer values `NA` is represented as the minimum value for the integers. We preserve the semantics of FastR on the GPU by using the same bit patterns to represent `NA` values in OpenCL.

```
class MApplyOCLNode {
 Object execute(RFunction function, RVector input) {
  function.compileToOCL();

  oclCode = cache.getOCLCode(function)
  if (oclCode != null)
    return runWithOCL(input, oclCode);

  for (int i = 0; i < input.size; i++) {
    output.add( function.call( input.at(i) ) );
    graph = cache.getGraph(function);
    if (graph != null) {
      oclCode = compileForOCL(graph);
      return runWithOCL(input, oclCode); } }
  return output; }
}
```

Listing 2: Run method in the AST interpreter for `OpenCLMApply` node.

3.4 Lexical Scope Analysis

The function passed to `mapply` is free to access variables which are defined outside of the function in its lexical scope. We traverse the AST of the function to determine which variables in the lexical scope are accessed. Then, for each found variable we infer its type. We use this information during the OpenCL code generation, where these variables become additional arguments to the OpenCL kernel.

3.5 AST Interpreter for `mapply`

Listing 2 shows a sketch of implementation of the execution method of the OpenCL enabled `mapply` node. This method is executed by the FastR interpreter.

We start by setting a flag indicating that the function passed as argument to `mapply` will be compiled to the GPU in line 3. We will discuss the OpenCL compilation process of this function in detail in Section 4. Next, we check in a cache if the R function has already been compiled to OpenCL code and execute the OpenCL kernel if that is the case (lines 5–7). If the function has not been compiled to OpenCL code yet, we continue by applying the function to the elements of the input vector in line 10. This executes the function in the Fast R AST interpreter on the CPU. After each iteration of the loop in line 9 we check if Truffle has performed the partial evaluation in a background thread. If this is the case the cache contains the control flow `graph` (CFG) (line 11). Once we obtain the CFG, we generate an OpenCL kernel in line 13 which we execute on the input vector in the next line.

The shown implementation continues execution on the CPU in the FastR AST interpreter until Truffle exceeds a compilation threshold and partial evaluation is performed to produces a CFG. This has the advantage that for small input sizes where the compilation to OpenCL has a significant overhead we compute the result directly in the interpreter.

3.6 OpenCL Code Generation Overview

The lower half of Figure 3 shows an overview of the OpenCL code generation process. The partial evaluation in Graal produces a CFG on which we perform additional specializations for removing unnecessary checks in OpenCL. After performing multiple lowering passes for constant folding, inlining, and other common optimizations we generate an OpenCL kernel using a simple visitor. The generated OpenCL kernel is stored in a cache to avoid the overhead of generating the same kernel twice for the same input CFG. We will discuss the OpenCL code generation in more details in the next section.

3.7 OpenCL Execution

To execute the generated OpenCL kernel we use an existing back-end for OpenCL [3]. To copy the input data to the GPU, we marshal the data from the R data types to OpenCL types, execute the OpenCL program and un-marshal the data. We will discuss in Section 5 how we can often avoid the slow marshaling of data.

4. OpenCL Code Generation

This section discusses the OpenCL code generation process starting from the AST created by FastR which is transformed into a control flow graph (CFG) by Graal via partial evaluation. The CFG is then specialized to OpenCL by removing unnecessary checks. Finally, an OpenCL kernel is generated from this specialized CFG. We discuss these steps in detail by following the example shown in Figure 4.

4.1 Partial Evaluation and Optimizations

We start from the R code shown in the top left corner of Figure 4. On interpretation FastR creates an AST and specializes it for the input data it observes during execution. The shown AST is already specialized based on the data types encountered while executing the function as part of the implementation of `mapply` (see Listing 2).

When the R function is executed often enough by `mapply`, the R function AST is partially evaluated by Graal into a control flow graph (CFG). To optimize the CFG, we apply a set of common Graal passes such as function inlining, constant folding, and partial escape analysis [19]. We do not apply any Graal passes which specialize the CFG for the target architecture (e.g. replacing a node with corresponding machine code instructions). The architecture independent but optimized CFG is shown in the lower left corner of Figure 4.

The CFG produced by partial evaluation combines the control flow of the FastR AST interpreter with the original R program. The addition and multiplication nodes at the bottom of the CFG are from the R program. The nodes checking for data type (`InstanceOf#Integer`) come from the FastR AST interpreter, whose Java source code we show on the left side.

In the example, the FastR AST interpreter has been specialized for the AST for `Integer` inputs. The Java code on the left side shows the guarding code which checks if the optimization assumption is broken and deoptimizes if that is the case. Deoptimization means that the CFG is invalidated and FastR returns to interpreting the code without the optimization assumption which was broken.

4.2 OpenCL Specialization

When executing the function on a GPU we can be sure that the input data must have the proper data type, `Integer` in this case. That is true, because we have marshaled the data prior to sending it to the GPU where we can ensure that the data types of all elements must match. Therefore, we can eliminate these checks from the AST interpreter.

We implemented a generic mechanism for eliminating such AST interpreter overheads and, thus, specialize the CFG for OpenCL execution. Truffle defines a set of *compiler directives* which are annotations conveying information useful for specializing and optimizing the CFG. One example is the `@CompilationFinal` directive we saw in Figure 1 which specifies that a value will not change anymore and, therefore, can be assumed constant by the Graal JIT compiler.

We added compiler directives for OpenCL specific use:

- `@NotNull`: specifies that the annotated variable is guaranteed to be not null.
- `@KnownType`: specifies that the type of the annotated variable is known.
- `@ArrayComplete`: specifies that the annotated array is guaranteed to not contain `NA` values.

We annotate the input arguments with all three directives as we can ensure these properties in the marshaling step. In the CFG, the arguments are part of the `FunctionFrame` shown as a table in Figure 4.

For functions marked as being compiled to OpenCL, as we have seen in Listing 2, an OpenCL-specific compiler pass is performed. This pass traverses the CFG and removes nodes by exploiting the additional information given by the OpenCL specific compiler directives. The highlighted `FixedGuard#TransferToInterpreter`, `Pi` and `InstanceOf#Integer` nodes, which dynamically check that the given parameter is of type `Integer` and convey this additional type information to the compiler using the `Pi` node, will be removed based on the `@KnownType` directive.

Using this strategy, we only remove checks where we are sure that this is legal. We will see in the section 6 an example where a deoptimization check is still required.

4.3 OpenCL Kernel Generation

After specializing the CFG for OpenCL most of the interpreter logic for handling exceptional cases on the CPU are gone. The remaining CFG, shown in the bottom right corner of Figure 4, has still the nodes corresponding to the original R program. The function `f` shown in the top right is generated by traversing the CFG and emitting an OpenCL snippet for each node. The remaining OpenCL kernel is generic for `mapply`. The two input vectors (`a` and `b`) are passed as arguments to the `mapply` kernel where the function `f` is called with two elements of the corresponding OpenCL thread-id.

5. Data Management Optimizations

This section discusses two important optimizations we implement in our compiler for efficient data management between R and the GPU.

Avoiding of marshaling for R vectors Vectors are the essential data structure in R to store collections of values which have the same data type. FastR specializes its implementation of R vectors based on the data type of the elements. Instead of storing an array of objects, FastR uses directly a primitive array for storing the values of a specialized R vector. For example, a `double` array for a R vector of double values. We take advantage of this specialization by FastR, as fortunately, primitive arrays are already in the byte format required for the GPU and no extra marshaling step to translate the data managed by FastR into a format accessible by OpenCL is required.

We modified the `PArray` (portable array) data structure presented in [3] to expose the primitive array in the R vector implementation to it. The `PArray` implementation then passes the primitive array to the OpenCL implementation which copies it to the GPU. We will see in the evaluation section the impact of avoiding the data marshaling step.

Optimization of R sequences A sequence in R represents all values between a *start* and *stop* value which can be reached with a given *step* size. In FastR sequences are handled specially: instead of storing the elements in memory, they are generated on-demand by evaluating the formula: *start + stride * index*. This is especially beneficial in the context of OpenCL, as we inline the formula in the OpenCL C code whenever an R function takes a sequence as its input argument. This saves the costly data transfer to the GPU.

In Listing 3 `x` and `y` are sequences ranging from 1 to a given `size`.

```
1 x <- 1:size; y <- 1:size   # R sequences
2 mapply(function(x, y) 0.12 * x + y, a, b)
```

Listing 3: R example to compute Daxpy with input sequences of integers

Listing 4 shows the OpenCL code corresponding to the R sequence specification `1:size`. The OpenCL thread id is used as the *index* in the computation of the formula.

```
1 int idx = get_global_id(0); // OpenCL thread id
2 int x = 1 + (1 * idx);
3 int y = 1 + (1 * idx);
```

Listing 4: OpenCL code snippet for R sequences

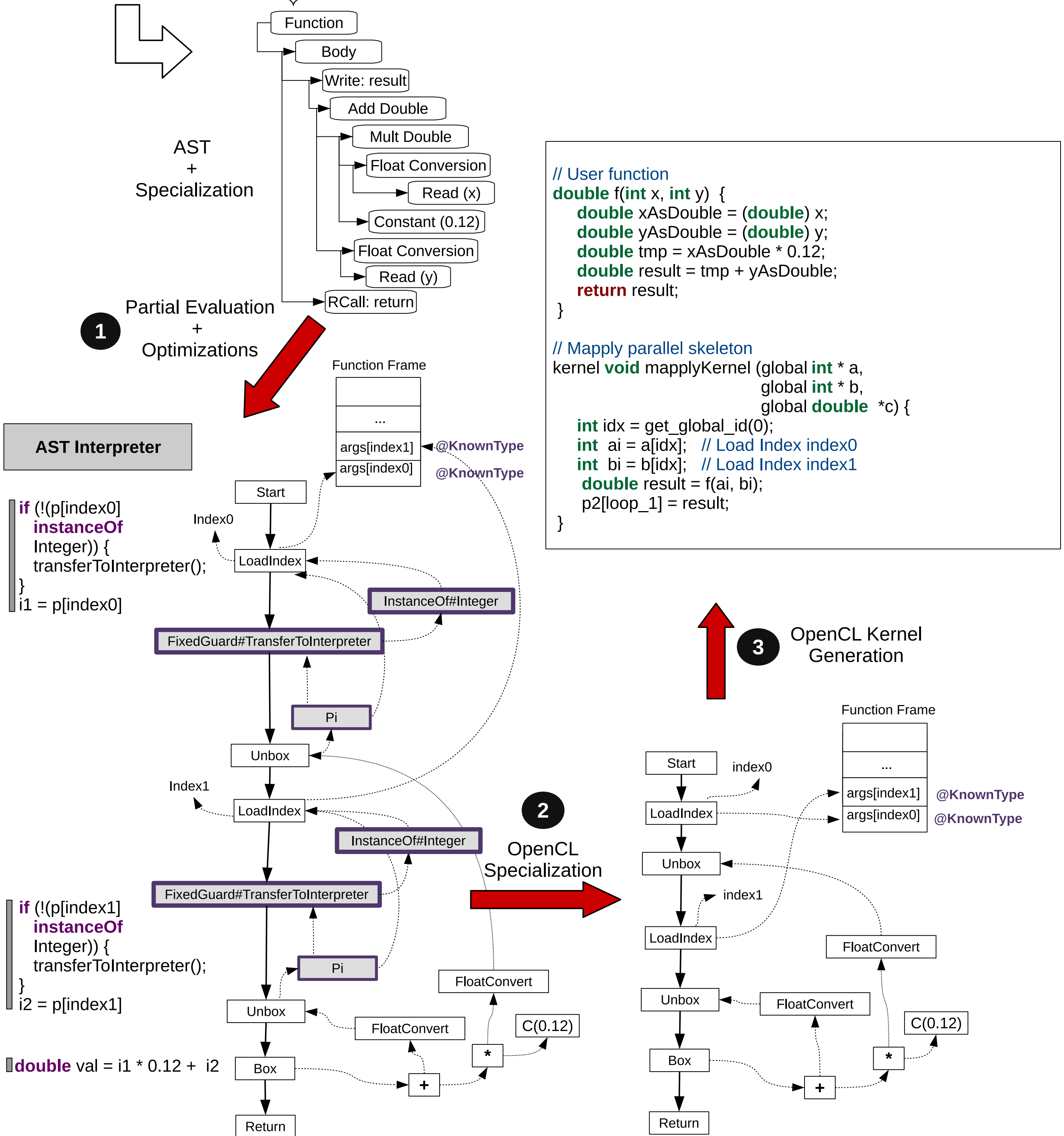

Figure 4: JIT Compiler from R interpreter to OpenCL C code.

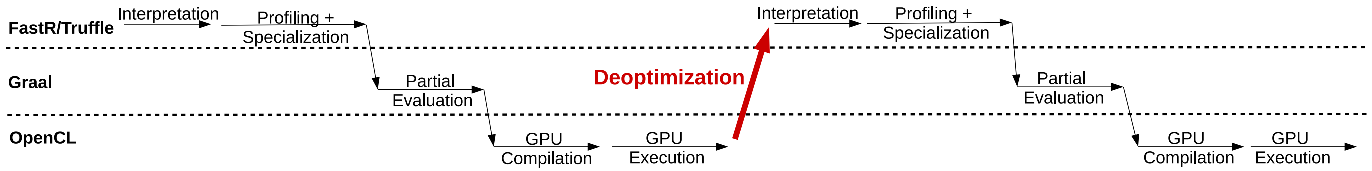

Figure 5: Execution of an R program which gets specialized, compiled to the GPU, before a deoptimization is performed, the program is re-profiled and generalized, compiled to the GPU again, and, finally, successfully executed on the GPU.

This technique has two clear advantages: a) OpenCL buffers and data transfer between the CPU and GPU are completely avoided. We only pass the start and stride information which are independent of the input array size; b) accesses to the slow OpenCL global memory into the input vector are avoided. This has a clear positive effect on the overall performance as we will see in the evaluation section.

6. Handling of Changes in Program Behavior

As we have seen earlier, Truffle uses specialization to optimize the AST during execution. The specialization is based on profiling information gathered during program execution. Truffle optimistically speculates based on the assumption that the profiling information of past executions is representative for future executions as well. Guards are introduced to handle changes in the program behavior when this assumption is broken and a deoptimization is performed.

Deoptimizations in OpenCL We discuss a simple example to illustrate how we handle deoptimization in our OpenCL code generator. The execution of the example is visualized in Figure 5. Listing 5 shows an R program which applies a function to an `input` vector. The function has a branch in line 2 that depends on the input `x`. If `x` is less than `1`, the function returns `0`, otherwise `1`.

```
1  mapply(input, function(x) {
2    if (x < 1) return(0)
3    else return(1) })
```

Listing 5: R function with a input depending branch

Let us assume that this function is executed on a large input array where the first 1000 elements have the value `0`. As shown at the top left corner of Figure 5, FastR will start interpreting the code and profile the execution by collecting statistics on which branches are taken in the execution. For the execution of the first elements the profiler will not registered an execution of the `else` branch. Based on this profiling information, the AST is specialized by FastR and after sufficient iterations the AST is handed to Graal for compilation via partial evaluation, as shown in Figure 5. The partial evaluation speculatively removes the computation in the `else` branch based on the profiling information. A guard checks that in case the else branch is hit the execution is returned to the interpreted code.

As a consequence of this behavior our OpenCL kernel generator operates on a CFG where the `else` branch is no longer present and generates the code shown in Listing 6.

```
1  double f(double x,global int* deoptFlag) {
2    bool cond = x < 1.0;
3    if(!cond) deoptFlag[0] = get_global_id(0);
4    return 0.0;
5  }
```

Listing 6: OpenCL C code generated for the R program in Listing 5.

The code unconditionally returns `0` which corresponds to the `then` branch of the original program. A guard checking for the optimization assumption is generated in line 3 where a global flag is set to indicate the deoptimization.

GPUs are a parallel hardware with no support for raising exceptions. Therefore, we wait until the kernel finishes execution. When it is finished, we check if the `deopt` flag was raised by any thread. If this is not the case and the flag is still set to its initial value of -1, the speculation was correct and the computed result is returned. Otherwise, we perform a deoptimization as indicated with the red arrow in Figure 5. This invalidates the generated OpenCL kernel and the control is transferred back to the FastR AST interpreter which continues interpreting the program and collect more profiling information.

Writing to the `deoptFlag` in global memory is not thread safe, but as we are only interested to see if a single thread has taken this branch this handling is sufficient. Writing the thread identifier into the flag has the advantage, that we can force the FastR AST interpreter to interpret the program with the input data of of least one particular thread that provoked the deoptimization. When the FastR AST interpreter reaches its compilation threshold again, (as indicated in Figure 5) the code is recompiled to OpenCL via partial evaluation with the updated profiling information. The kernel shown in Listing 7 is generated.

```
1  double f(double x,global int* deoptFlag) {
2    bool cond = x < 1.0;
3    if (cond) return (0.0);
4    else return (1.0);
5  }
```

Listing 7: OpenCL C code generated for the R program in Listing 5 after re-profiling

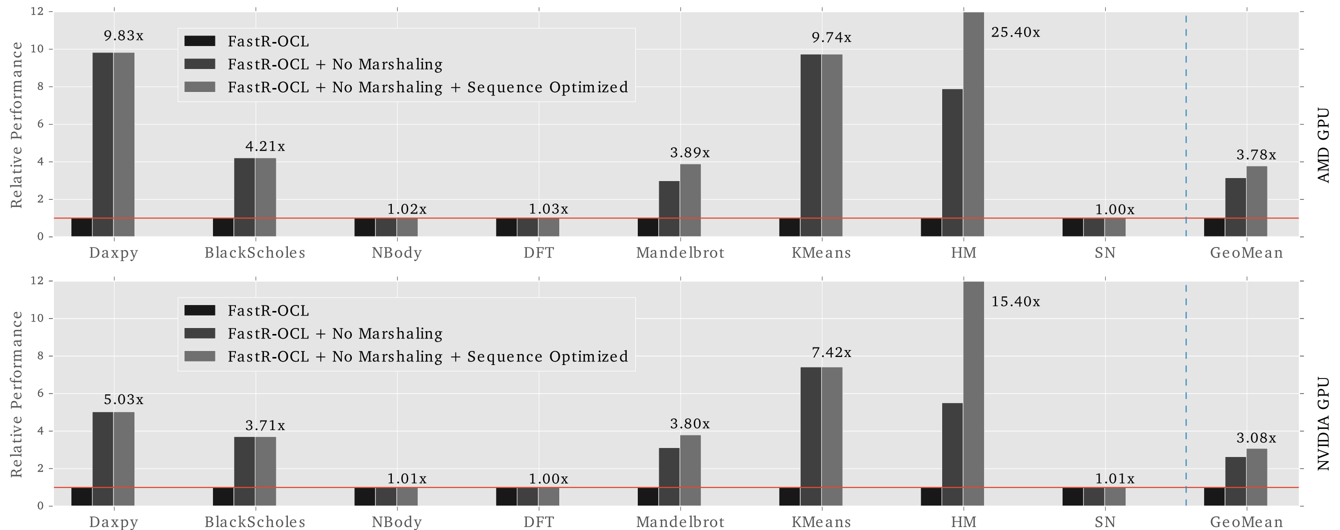

Figure 6: Performance impact of data management optimizations in FastR-OCL.

The additional profiling information have helped generalizing the AST and both branches are now visible to our OpenCL code generator. Our technique is a very simple strategy to handle deoptimizations with minimal overhead in the OpenCL kernel.

7. Evaluation

This section presents the performance evaluation of our JIT GPU compilation approach for the R language. We describe the experimental setup and the benchmarks used, discuss the performance results, and provide some analysis.

7.1 Experimental setup

We evaluate our compiler approach on two different GPUs; an AMD Radeon R9 295X2 with 8GB memory and a GeForce GTX TITAN Black with 6GB memory. We use the AMD 1598.5 and Nvidia 367.35 GPU drivers. The CPU used in the experiments is a Intel Core i7 4770K @ 3.50GHz with 16GB of DDR3 RAM.

We provide a comparison between FastR, GNU R, OpenCL C++ and our GPU enabled version of FastR which we call *FastR-OCL*. We use GNU R version 3.3.1 with the $enableJIT$ option set to level 3 which enables the JIT compiler. Both FastR and our FastR-OCL extension are running on top of the Graal 0.9 Java VM (Virtual Machine). Our OpenCL code generator is built on top of Graal VM JIT compiler which is exposed via a Java API.

We execute each benchmark 10 times and report the median execution time. We execute FastR as well as our FastR-OCL with 12GB of Java heap memory. For FastR and FastR-OCL, we performed time measurements using a custom built-in, which internally calls `System.nanotime()`. For GNU R, we measure the time by calling the `proc.time()` function. All times presented in this section are measured from R and include GPU related overheads such as the data management.

Benchmark	Input (MB)	Output (MB)
Daxpy	128	64
Black-Scholes	8	16
NBody	5	3
DFT	0.156	0.156
Mandelbrot	16	8
Kmeans	64	16
Hilbert Matrix (HM)	128	128
Spectral Norm (SN)	0.256	0.256

Table 1: Benchmarks and default data sizes use to evaluate our FastR-OCL implementation.

7.2 Benchmarks

We implemented a set of benchmarks in R following data parallel benchmark implementations from Rodinia [1], the AMD OpenCL SDK and the programming language benchmarks game. We selected the following eight benchmarks representing data and compute intensive applications from different domains: `Daxpy`, `Black-scholes`, `Mandelbrot`, `DFT`, `NBody`, `Kmeans`, `Hilbert Matrix` and `Spectral Norm`.

Table 1 shows the data sizes used for the execution of the benchmarks in our first two experiments. The sizes are chosen to be large enough to lead to sufficiently long runtimes in FastR but are fairly small for our massively data parallel GPUs. As we will see, even with this moderate data size for GPUs large speedups are obtained compared to FastR.

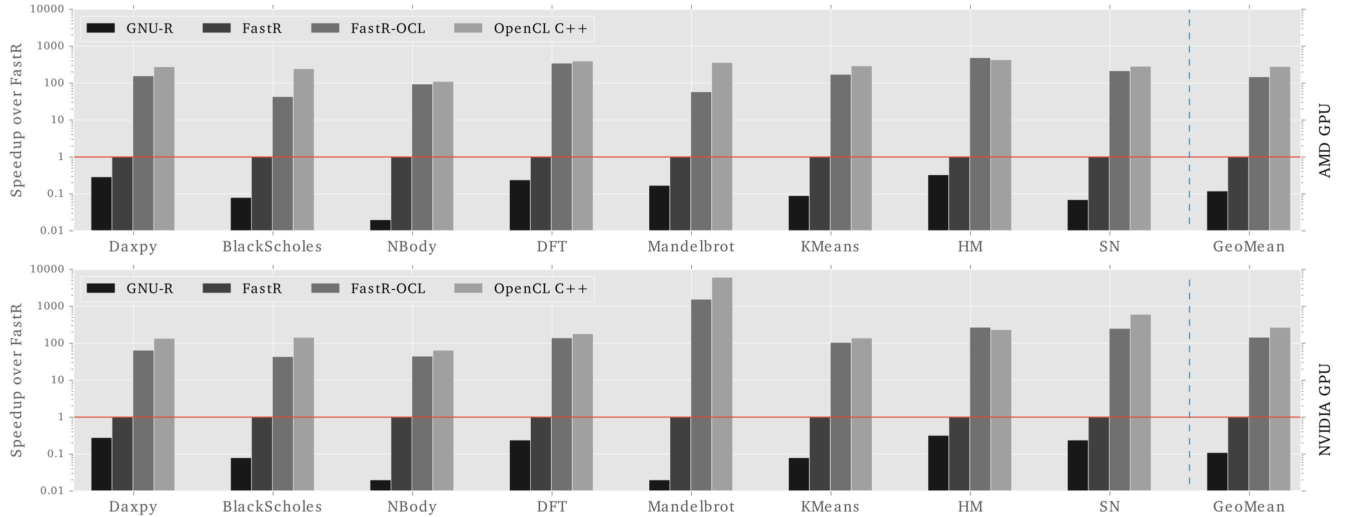

Figure 7: Speedup of FastR-OCL executing on AMD and Nvidia GPUs compared to GNU-R, FastR, and OpenCL C++.

7.3 Impact of Data Management Optimizations

We start by investigating the impact of the data management optimizations presented in Section 5. Figure 6 shows relative performance of three versions of our FastR-OCL implementation: 1) the leftmost bar shows the naive version which performs marshaling for R vectors and sequences; 2) the middle bar shows the optimized version avoiding marshaling for R vectors and sequences but performs data transfers for R sequences; 3) the rightmost bar shows the most optimized version avoiding marshaling for R vectors and also avoiding data transfers for R sequences. The graph shows the performance for AMD at the top and Nvidia at the bottom.

The benefits of the data optimization are dependent on the compute intensity of the benchmark. For compute intensive benchmarks, such as `Nbody`, `DFT`, and `SN` the optimizations have a minor effect, as the vast majority of execution time is spend in the computation on the GPU. For all other benchmarks, the benefits are more significant with runtime improvements of up to $25.4\times$ for the `HM` benchmark. Since the `HM` program takes two R sequences as the input, it is possible, through our optimization, to remove most of the data transfer, since an RSequence can be simply represented by three numbers: start, stride and end. Overall, the geometric mean improvement of our optimizations is $3.78\times$ on AMD and $3.08\times$ on Nvidia.

7.4 Performance Comparison

Figure 7 shows a performance comparison for each benchmark with GNU R, FastR, our FastR-OCL implementation, and a native hand-optimized OpenCL implementation of each benchmark in C++. The bars show speedup over the sequential execution with FastR. Due to the large speedups achieved by the GPUs, the y-axis is in logarithmic scale. The input R programs used in our evaluation are exactly the same for GNU R, FastR, and our FastR-OCL implementation.

We can see that, FastR is 10 to $100\times$ times faster than GNU R for these data intensive benchmarks, confirming prior results [18]. By using the AMD GPU, our approach is $150\times$ times faster than FastR on average and $1000\times$ faster than GNU R. Using the Nvidia GPU our FastR-OCL is on average $130\times$ faster than FastR. For some benchmarks, such as `Mandelbrot` FastR-OCL achieves a speedup of more than $1000\times$ compared to FastR by using the Nvidia GPU.

The last set of bars shows the performance achieve by a highly tuned manually written native C++ OpenCL implementation. For the `nbody`, `dft`, `HM` and `SN` benchmarks, FastR-OCL achieves very similar performance to the native OpenCL implementation. For benchmarks such as `Mandelbrot`, the C++ OpenCL implementation is clearly faster due to the fact it exploits parallelism in multiple dimension where our code generator only exploits a single dimension of the OpenCL thread iteration space. On average our implementation is about $1.8\times$ slower than the native OpenCL implementation. Although there is room for improvement, this is achieved fully automatically, starting from a program written in the R dynamic interpreted language.

7.5 Performance of Cold Runs

The experiments we did so far where measured as a median of multiple runs inside the same R session running on top of the Java VM. Therefore, the VM had time to warm up and perform the JIT compilation before reporting a measurement. A more typical end user scenario is to execute the R program only once with a fresh JIT compiler state. To this end, we start a new R session, execute and measure the whole R program execution time once and produce a single data point — which corresponds to a single execution of a R

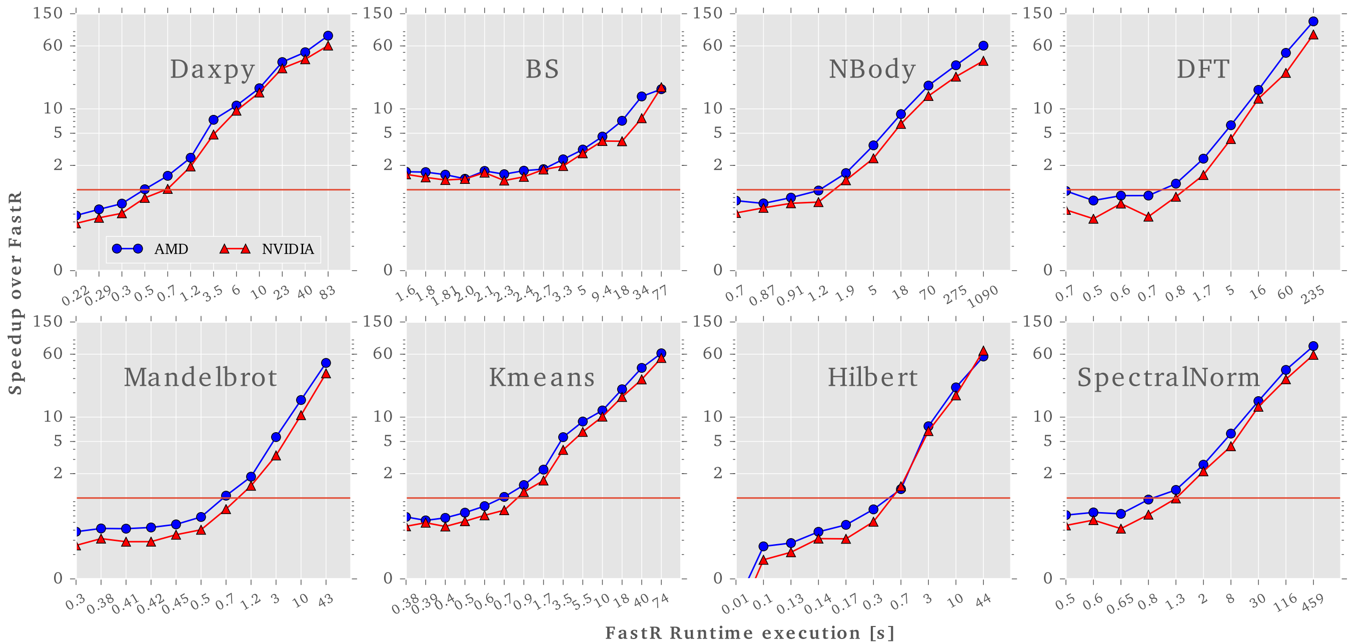

Figure 8: Speedup of FastR-OCL over FastR executed with a cold JIT compiler with increasing input sizes from left to right. The x-axis labels show the FastR runtime while the y-axis shows the speedup of our GPU-enabled FastR-OCL implementation. Even R programs running only for a few seconds achieve large speedups.

program on a cold JIT compiler — after which the R session is terminated. We repeat this process 10 times and report the median value to account for noise.

Figure 8 shows a performance comparison of FastR and FastR-OCL measured using a cold JIT compiler. For each benchmark, we increase the input size from left to right and the x-axis labels show the FastR execution time. The y-axis shows the speedup achieved of FastR-OCL over FastR for the AMD (blue circle) and Nvidia (red triangle) GPU.

Across all benchmarks, we see that even on a cold JIT compiler, a data size which executes for only a few seconds in FastR is sufficient for achieving a speedup using the GPU via our FastR-OCL. For example, using the GPU becomes beneficial for the simple `daxpy` benchmark for input sizes where the computing process takes longer than 0.7 seconds in FastR. For an input size where FastR runs for 83 seconds, FastR-OCL obtains speedups of more than $60\times$ for both GPUs. Overall we observe that despite the AST specialization, partial evaluation, OpenCL kernel generation and compilation, large speedups are obtained even for R programs which execute only for a few seconds.

7.6 Breakdown of OpenCL Execution Times

The selected benchmarks reflect different types of GPU applications. Figures 9 and 10 show a breakdown of the peak performance OpenCL run-times for all benchmarks on the AMD and NVIDIA GPUs. The total runtime is broken down into four parts: copying the data to the GPU ($H \rightarrow D$), the kernel execution, copying the data from the GPU ($D \rightarrow H$) and remaining time (other), which includes the runtime of the interpreter. The data sizes used for these experiments are shown in Table 1.

Kernel Execution The benchmarks that achieve the highest speedups with our OpenCL JIT compiler for GPUs, are those where the predominant part is the OpenCL kernel execution. This is the case for NBody, DFT and Spectral Norm, where the OpenCL computation takes up to 90-99% of the overall runtime. For the rest of the benchmarks, the kernel takes between 3-15%. In these cases, an efficient data management is crucial to achieve good performance.

Data Transfers The figures show clearly the benefits of our data transfer optimizations for the R sequences in OpenCL. This is the case of DFT, Mandelbrot, Hilbert and Spectral Norm, where copying the data to the GPU takes less than one per cent. For some benchmarks the data transfer time takes up to 75% of the overall execution time. However, as shown in Figure 8, even with the data transfers to and from the GPU, R programs can benefit with large speedups.

7.7 Compilation time

Table 2 shows the compilation time for each benchmark. Partial evaluation and optimizations performed by Graal take significantly longer than the OpenCL kernel generation which takes only about 11ms. The compilation of the OpenCL kernel by the GPU driver can take up to 250ms. Overall we see that the overhead for OpenCL kernel generation is small and less than partial evaluation on the AMD

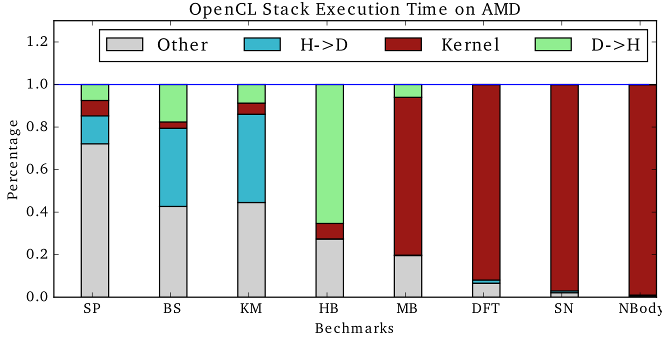

Figure 9: Breakdown of the OpenCL execution run-times for 8 R benchmarks executed on a AMD ATI GPU.

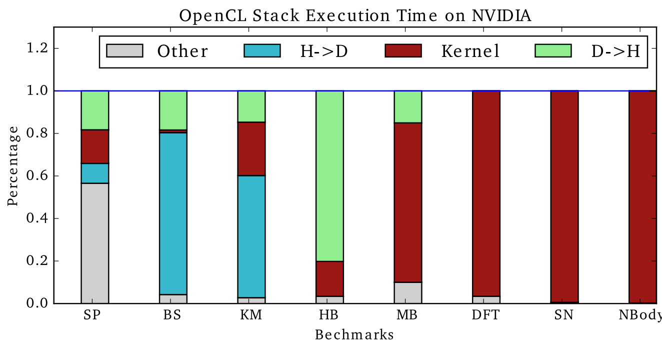

Figure 10: Breakdown of the OpenCL execution run-times for 8 R benchmarks executed on a Nvidia Gforce GPU.

platform and similar to partial evaluation for the Nvidia platform. This low overhead also explains why using the GPU is already beneficial for small input sizes, as seen before.

The actual OpenCL GPU compilation time shows that further improvement are possible by directly targeting the GPU instruction set and bypass the OpenCL compiler. In the future, we intend to investigate the use of OpenCL SPIR (Standard Portable Intermediate Representation) instead of plain OpenCL source code to lower even more the compilation overhead. However, as our results have shown, large speedups are already achievable, even when relying on the OpenCL compiler.

7.8 Summary

This evaluation has shown that accelerating R programs with FastR-OCL is not just feasible but highly beneficial for data parallel applications. Exploiting the parallel power of the GPU leads to large speedups. Our data management optimizations to avoid marshaling for R vectors and data transfers altogether for R sequences increase performance by up to $25\times$. We obtain speedups ranging from $43\times$ up to more than $1000\times$ depending on the benchmark when compared to the sequential execution in FastR. Even when executing the R program only a single time on a cold JIT compiler we still obtain large speedups and have shown that using the GPU is beneficial even for short running R programs.

Time in ms				
Benchmark	**PE + Opt.**	**Kernel Gen.**	**Comp. AMD**	**Comp. Nvidia**
Daxpy	106.27	7.27	29.19	81.32
Black-scholes	94.54	12.70	55.79	180.20
NBbody	93.59	11.35	47.04	109.82
DFT	95.09	20.12	61.56	250.86
Mandelbrot	114.31	9.37	34.70	102.78
K-Means	113.54	9.62	41.33	93.43
Hilbert	105.44	7.74	27.82	95.67
Spectral N.	146.57	11.86	87.03	219.53
Mean	107	11	48	142

Table 2: Time (in milliseconds) for different phase of the GPU code generation process: Partial evaluation and optimizations (PE + Opt.), OpenCL kernel generation (Gen.) and compilation by the GPU driver (Comp.).

8. Related Work

This section reviews the relevant literature on exploiting parallelism from R and other interpreted languages.

Library-based R GPU support The majority of approaches for parallelizing programs written in dynamically interpreted languages such as R relies on the use of libraries. There are numerous R libraries that support CUDA and OpenCL execution. These libraries typically implement well-known building blocks in specific application domains, such as the *GPUR* library which supports matrix operations. Internally, the library is written in CUDA or OpenCL, or it leverages an existing implementation such as ViennaCL [17] in the case of *GPU*. Other approaches rely on the use of wrapper for low-level interface such as OpenCL and they require the programmer to write OpenCL code inside R programs. The technique presented in this paper is fully automatic and it does not rely on any library implementation. Moreover, it is more generally applicable than all these existing library-based approaches.

Parallelism Exploitation in R Riposte [20] uses trace-driven compilation to dynamically discover vector operations from arbitrary R code and it produces specialized parallel code for CPUs. Similar to us, other researchers have targeted the `apply` function to extract parallelism by automatically vectorizing [21] its implementation for the CPU. This paper is the first work to show how parallelism can be exploited to automatically accelerate R programs on GPUs.

Specialization in R There has been work on using specialization techniques with R in order to speed up the execu-

tion of the R interpreter. FastR [9, 18], on which our work is based, using Graal as a JIT compiler to produce efficient code for the CPU. Orbit VM [22] uses profile-directed specialization techniques for accelerating the execution of R code. Renjin [10] uses a delay computation mechanism similar to lazy evaluation that produces a computation graph. Once the results is needed, the computation is then optimize using type information available for instance to produce a more efficient execution of the computation. As far as we are aware, this paper is the first one to investigate the use of these techniques for JIT GPU code generation.

GPU acceleration for other interpreted languages Very few GPU code generators exist for other interpreted languages. Among the existing works, *Numba* [13] is a CUDA JIT compiler for python which is based on annotations to identify parallel section of code and data types. Our approach does not require any user annotation and instead it exploits the parallel semantic of existing R operations such as the `apply` functions. *Harlan-J* [15] is an OpenCL JIT compiler for JavaScript. It is based on Harlan-J language, a JavaScript extension for data parallelism. None of these approaches provide a fully automatic heterogeneous JIT compiler for high-level languages. Programmers need to change the code and adapt it for GPU execution. Our approach is totally transparent with all code transformations happening automatically at runtime without programmer intervention.

GPU JIT Compilers for Java There are some prior works on JIT compilation to CUDA or OpenCL for Java such as *AMD Aparapi*, *Rootbeer* [16], and *JaBBE* [24]. In all the projects, programmers need to extend a provided GPU class and implement an `execute` method that wraps the GPU code. These approaches still remain low-level as the parallel execution is explicit even if the applications are implemented in a high-level programming language.

Sumatra is a JIT compiler that automatically generates HSAIL code at runtime using the Graal compiler for the Java 8 Stream API. Similar to Sumatra, *IBM J9* [8] generates OpenCL C code for the `forEach` construct of the Stream API. In our prior work we implemented an OpenCL JIT compiler using Graal for the JVM [3]. Parallel and heterogeneous Java programs are implemented using a new Java API for heterogeneous computing, JPAI, where, in contrast to Java Stream API, the computation is composable and reusable multiple times once they are defined [4].

This paper targets interpreted and dynamic programming languages on GPUs, which is significantly more challenging because of the OpenCL compiler has to reduce the interpreter overhead, specialize the input program and handle with speculations on GPUs. Once our OpenCL JIT compiler has a specialized AST and has obtained the type information, the code generation process that translates the Graal IR to OpenCL is very similar to these existing Java works.

9. Conclusions

In this paper, we have presented a technique to transparently and automatically offload computations from interpreted dynamic languages to GPUs. We implemented this technique for the R programming language as a modification of the FastR interpreter which is built on top of Truffle and Graal. To the best of our knowledge, this paper presents the first OpenCL JIT compiler for R.

We have discussed the challenges when generating high performance OpenCL code from managed languages and we have shown that the combination of AST specialization and partial evaluation helps to reduce overheads. We use compiler directives to convey optimization information for avoiding unnecessary checks on the GPU.

We presented data management optimizations for avoiding marshaling of R Vectors and an optimized handling of R Sequences which even avoids costly data transfers. Our approach is able to handle cases where a specialized AST has to be deoptimized and generalized during execution. On average our approach is $150\times$ faster than FastR on a range of data intensive benchmarks and only $1.8\times$ slower compared to manually written OpenCL code. We achieve significant speedups of $60\times$ and more even for short running R programs when using a cold JIT compiler.

In future work, we would like to extend the coverage of R language features, such as the handling of data frames. Our technique is not special to R but it could easily be applied to other Truffle languages such as Ruby or JavaScript. We plan to explore this path in the future.

Acknowledgments

The authors would also like to thank the anonymous reviewers as well as Roland Schatz, Stefan Marr and Gilles Duboscq for fruitful discussions.

References

[1] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. IISWC 2009.

[2] 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. VMIL 2013.

[3] J. J. Fumero, T. Remmelg, M. Steuwer, and C. Dubach. Runtime Code Generation and Data Management for Heterogeneous Computing in Java. PPPJ 2015.

[4] J. J. Fumero, M. Steuwer, and C. Dubach. A Composable Array Function Interface for Heterogeneous Computing in Java. ARRAY, 2014.

[5] Y. Futamura. Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation*, 1999.

[6] A. Gal, C. W. Probst, and M. Franz. HotpathVM: An Effective JIT Compiler for Resource-constrained Devices. VEE 2006.

[7] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. PLDI 1992.

[8] K. Ishizaki, A. Hayashi, G. Koblents, and V. Sarkar. Compiling and optimizing java 8 programs for gpu execution. In *PACT*, 2015.

[9] T. Kalibera, P. Maj, F. Morandat, and J. Vitek. A Fast Abstract Syntax Tree Interpreter for R. VEE 2014.

[10] M.-J. Kallen and H. Mühleisen. Latest developments around renjin. Talk at R Summit & Workshop, Copenhagen, 2015.

[11] M. N. Kedlaya, B. Robatmili, C. Caşcaval, and B. Hardekopf. Deoptimization for Dynamic Language JITs on Typed, Stack-based Virtual Machines. VEE 2014.

[12] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot&Trade; Client Compiler for Java 6. *ACM Trans. Archit. Code Optim.*

[13] S. K. Lam, A. Pitrou, and S. Seibert. Numba: A LLVM-based Python JIT Compiler. LLVM 2015.

[14] M. Paleczny, C. Vick, and C. Click. The java hotspottm server compiler. JVM' 2001.

[15] U. Pitambare, A. Chauhan, and S. Malviya. Just-in-time Acceleration of JavaScript. In *Technical Report, School of Informatics and Computing, Indiana University*, 2013.

[16] P. C. Pratt-Szeliga, J. W. Fawcett, and R. D. Welch. Rootbeer: Seamlessly Using GPUs from Java. HPCC-ICESS, 2012.

[17] K. Rupp. GPU-Accelerated Non-negative Matrix Factorization for Text Mining. page 77, 2012.

[18] L. Stadler, A. Welc, C. Humer, and M. Jordan. Optimizing R Language Execution via Aggressive Speculation. DLS 2016.

[19] L. Stadler, T. Würthinger, and H. Mössenböck. Partial escape analysis and scalar replacement for Java. In *CGO*, 2014.

[20] J. Talbot, Z. DeVito, and P. Hanrahan. Riposte: A Trace-driven Compiler and Parallel VM for Vector Code in R. PACT '12, 2012.

[21] H. Wang, D. Padua, and P. Wu. Vectorization of Apply to Reduce Interpretation Overhead of R. OOPSLA 2015, .

[22] H. Wang, P. Wu, and D. Padua. Optimizing R VM: Allocation Removal and Path Length Reduction via Interpreter-level Specialization. CGO 2014, .

[23] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to Rule Them All. Onward! 2013.

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