Just-In-Time GPU Compilation of Interpreted Programs with Profile-Driven Specialization

Abstract

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

On the application side, interpreted dynamic languages are increasingly becoming popular in many emerging domains for their simplicity, expressiveness and flexibility. However, this creates a wide gap between the nice high-level abstractions offered to non-expert programmers and the low-level hardware-specific interface. Currently, programmers have to rely on specialized high performance libraries or are forced to write parts of their application in a low-level language like OpenCL. Ideally, 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 analytics. Our experimental results show execution on a GPU yields speedups of over 150x when compared to the sequential FastR implementation and performance is competitive with manually written GPU code. We also show that when taking into account startup time, large speedups are achievable, even when the applications runs for as little as a few seconds.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Code generation, Compilers, Interpreters, Optimization

General Terms Language, Performance

Keywords Code Generation, GPU, JIT, Virtual Machine

1. Introduction

Nowadays, most computer systems are equiped with powerful parallel devices such as Graphics Processing Units (GPUs). Many applications domains can benefit from these devices, often achieving up to three orders of magnitude speedup over parallel CPU code. However, exploiting this hardware requires the knowledge of new computer architectures and low-level languages such as OpenCL, CUDA, C and Fortran. This is a very challenging task for non-expert programmers.

Many non-computer scientists prefer using interpreted dynamic languages such as Ruby, Python or R which are hugely popular despite their reputation for poor performance. They offer high-level functionality, simplicity of use and the interpreter allows for fast software development. However, exploiting a GPU from these languages is far from trivial for programmers since they 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 parallelize the execution on 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 well suited for GPUs given that some parts of the interpreter are hard to port such as method dispatch and object representation. As a consequence, we can expect to pay a large overhead for interpreting code on the GPU.

Recent developments in the area of improving interpreter’s performance on traditional CPUs have seen the emergence of profile-driven specialization [20, 21]. Such techniques specialize the interpreter to the application being run by coupling partial evaluation with a JIT code generator. The code produced, therefore, becomes specialized to the actual observed types or to the hot path in the control-flow as in a trace-based compiler [7]. As a result, the interpreter is almost completely compiled away, leaving only the actual application logic and computation.

In this work, we propose to extend these techniques to GPUs. Juan I think we should change this, according to the current title□ We present an OpenCL JIT compiler framework for dynamic programming languages based on profile-driven specialization to accelerate interpreted programs with minimal effort for the language implementer. The main idea is that if the code performing the interpreter can be completely removed using specialization and partial evaluation, it is possible to produce efficient GPU code. As we will see, this is almost possible with little effort and minor modification to the existing interpreters.

Using R as a use case, we extend the recently developed R interpreter FastR [17] and the Graal JVM JIT compiler to support the generation of OpenCL code from R source. We achieve this using profile-driven specialization with only minor modifications to the existing infrastructure. The R interpreter is slightly modified 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 transforms the AST of the program into the Graal Intermediate Representation (Graal IR) that is used for the compiler to machine code Juan something is missing here, generate?□. We then perform a series of passes that simplify the IR as much as possible and the code generator attempts to produce an OpenCL kernel. If successful, the kernel is executed on the GPU. We currently support a subset of the R language, therefore, if the JIT compilation process fails due to unsupported features, controls return to the interpreter automatically and safely using the standard back guard (deoptimisation [9]) mechanism.

Our experimental results show that this approach is feasible and that it is possible to accelerate R programs on the GPU automatically. 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 48x speed-up in a more realistic scenario where we include the OpenCL compilation time and OpenCL device initialization costs.

To summarize, the contributions of this paper are as follow:

- We present an OpenCL JIT compiler framework for AST interpreters using Truffle and Graal.
- We present a technique for simplifying the Graal intermediate representation for OpenCL code generation.
- We present a performance evaluation for a set of R applications running on GPUs. We compare our implementation with the standard de-facto GNU-R, FastR and OpenCL C++ implementations.

2. Background

Truffle [21] 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).

Interpreter engine 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` methods 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 incoming 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 behaviour. 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.

Truffle DSL 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.

Compilation 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, Truffe employs Graal [21] for generating optimized machine code. Graal is a bytecode to native code JIT compiler implemented in Java, which can replace the client [10] and server [13] compilers in the HotSpot JVM. It transforms bytecode to the high-level GraalIR [4] 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 [6], 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 [8] and control will be transfered back to the interpreter which modifies the AST to accommodate the new data types. The AST is then recompiled with the additional functionality.

3. Design and System Overview

This section presents an overview of our OpenCL JIT compiler framework for dynamic languages. We describe the middle-ware software stack and we present our modifications to the existing Truffle and Graal projects.

3.1 Motivation Example in R

Modern programming languages normally have built-in functions in their specifications for common operations. Some of these intrinsics can be computed in parallel, such as map, reduce, or filter. Although the compiler and the runtimes are free to implement those operations in parallel, they do not usually provide a parallel implementation. Our idea is to take those built-in functions as skeletons and generate the corresponding parallel code for OpenCL.

We take the apply operations in R, like *mapply*, and represent them as a new node in the AST with parallel semantic. The implementation of the parallel operations contain the default sequential implementation with a few modifications that enables the OpenCL compilation and execution.

Listing 1 shows an example in R with the `mapply` function to compute `Daxpy` (a multiplication of a double for a vector and an addition in double precision).

```
> x <- runif(size)
> y <- runif(size)
> mapply(function(x, y) 0.12 * x + y, a, b)
```

Listing 1: R example to compute Daxpy

The `mapply` method showed in Listing 1 takes three arguments, the function and the input data. The `mapply` executes the function passed as argument for every element in the input data set.

If the user code is executed multiple times, we compile the R code to OpenCL automatically. Listing 2 shows the auto-generated OpenCL C code. Lines 1-4 show the OpenCL code for the R function in the `mapply` built-in. Lines 5-12 correspond to the parallel skeleton for the `mapply` function (map parallel skeleton [3]).

```
 1  double f(double p0, double p1) {
 2     double r1 = x0 * 0.12;
 3     double x2 = x1 + x1; return x2;
 4  }
 5  kernel void oclKernel(global double * p0,
 6                        global double * p1,
 7                        global double  *p2) {
 8     int idx = get_global_id(0);
 9     double  x0 = p0[idx];
10     double  x1 = p1[idx];
11     double result = f(x0, x1);
12     p2[idx] = result; }
```

Listing 2: OpenCL C code for the R code shown in Listing 1

3.2 Compiler framework overview

Figure 2 shows the middle-ware software stack of our system. The light gray blocks represent the existing projects and the dark gray blocks show our contributions.

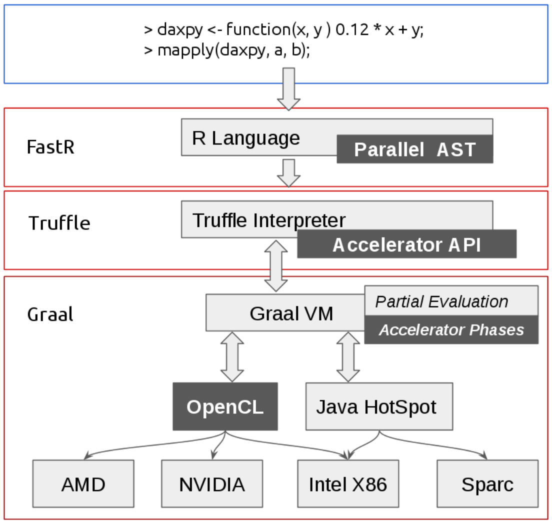
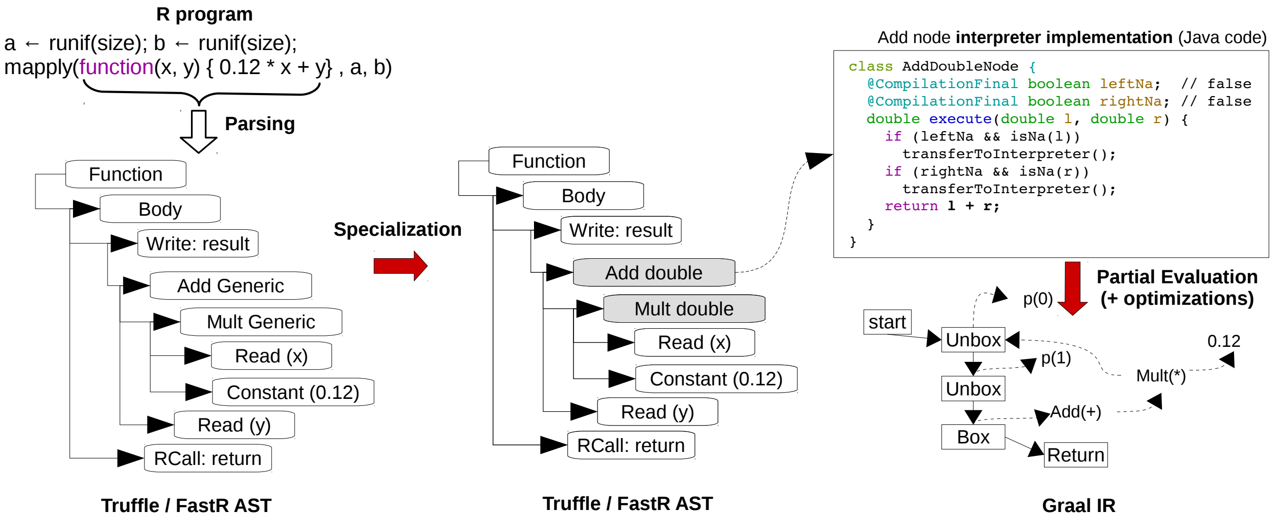


Figure 2: Compilation process from R programming language to OpenCL C code.

The top of the Listing shows an R application (daxpy) representing the input to our system. The R program is executed within the Truffle R interpreter (FastR). First, FastR builds an AST that represents the input program. If there are calls to any parallel built-in like `mapply`, we represent the operation in an AST node with parallel semantic.

Then we execute the R user function in the AST interpreter. The interpreter collects profiling information and rewrites the AST with the inferred types. If the R function is executed multiple times and it exceeds a threshold, Truffle marks the AST as candidate for compilation. The compilation is first performed via partial evaluation in Graal.

The partial evaluator produces a CFG in the Graal-IR form [4]. This graph is used by Graal to apply the optimizations and generate efficient machine code. We intercept the CFG in one of the final steps in the high-level phases of Graal and we optimize it for OpenCL. One of the biggest challenges is to filter the compiler overhead to the real computation. As Stefan et. al. [12] explained, partial evaluation does not solve to remove the overhead of the interpreter, but the lowering phases. However, we do not wait until Graal compiler applies the full lowering. Our code generator takes the high-level Graal IR and generates OpenCL code. This means that we can only generate OpenCL code from the



```
class AddDoubleNode {
  @CompilationFinal boolean leftNa;  // false
  @CompilationFinal boolean rightNa; // false
  double execute(double l, double r) {
    if (leftNa && isNa(l))
      transferToInterpreter();
    if (rightNa && isNa(r))
      transferToInterpreter();
    return l + r;
  }
}
```

Figure 1: AST specialization in Truffle.

Graal high-level IR. The following sections explain how we handle and clean the CFG for OpenCL code generation.

4. JIT Compiler for AST Interpreters

This section explains in detail all the extensions we implement on top of the FastR interpreter, Truffle and Graal that allows OpenCL JIT compilation for R.

FastR OpenCL components Figure 3 shows how all the components in the compiler and the runtime are connected. The light gray squares show our extensions to the FastR and Truffle/Graal compilers. FastR parses and builds an AST for the input R code. Once the AST is built, we infer the input and output data types for OpenCL. We also analyze the lexical scope of the R function and obtain their types and capture their values.

When the R code is running in the interpreter, we check if the R function is marked as candidate for compilation. In that case, we create a mapping between the AST and the compiled graph. Right after the partial evaluation we simplify the CFG for OpenCL and perform the Graal lowering. This allows us to reduce the compiler graph before the OpenCL code generation.

Then we compile to OpenCL C code. We rely on the existing code generator back-end for OpenCL [5] with Graal. When the OpenCL compilation finishes, we transform the data from the R data types to OpenCL types (marshal), execute the OpenCL program and un-marshal the data. Marshalling the data, as we will see, is a very slow process. Fumero et. al. [5] presents a strategy to avoid the marshal for Java and OpenCL programming by using a data structure called `PArray`. We extend the concept of `PArrays` for Truffle languages to avoid the data transformation completely. In the rest of this section we explain the extensions in the AST interpreter.

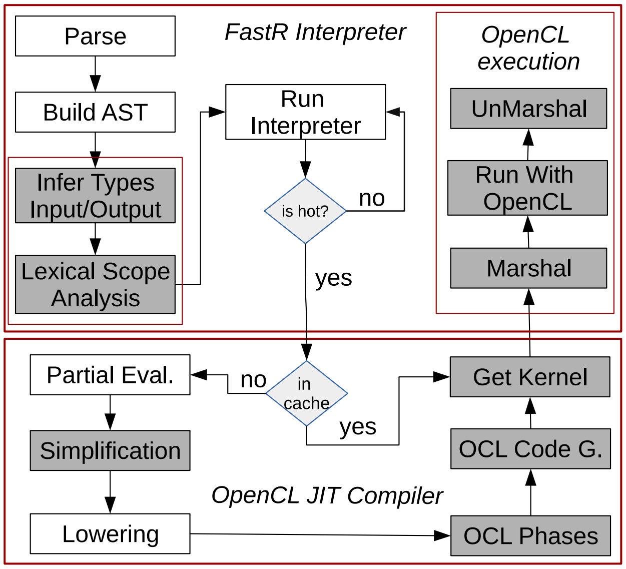


Figure 3: JIT Compiler from R interpreter to OpenCL C code. The white squares represent the existing components in FastR and Graal compilers. The gray squares represent our contributions.

4.1 Built-in redefinition for OpenCL

The parallelization to OpenCL is automatic. To do so, we extend the FastR interpreter with a few parallel built-ins based on the existing ones in the R language. We load a new version of the built-in during the VM start-up. Listing 3 shows the modification for the `mapply` function. Lines 3-5 check if there is any `OpenCL` driver available in the local machine. If it is true, we call the new version and we create a new node in the AST (*OpenCLMApply*) with parallel semantic.

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

Listing 3: Modification of the R MApply function for OpenCL computation

The following sections describe how we translate from the input R code to the output OpenCL C code at runtime.

4.2 Enabling OpenCL acceleration within FastR

With the built-in redefinition in the VM start-up, if the `mapply` function is called, it will execute our updated version for OpenCL. Listing 4 shows a sketch of our interpreter for the OpenCL `apply` in the AST. Firstly we enable the OpenCL execution in lines 2-3. Then we check if the R function has been already compiled to OpenCL (line 6). If it is that the case, we obtain the OpenCL binary from an internal cache, and we execute it (line 7). If not we execute the R function in the AST interpreter (lines 8-17).

The compilation normally happens in a background thread in the VM. Therefore, the R code can be compiled at any point after the VM detects the threshold. We add a condition to know if the CFG after partial evaluation was already inserted into an OpenCL compilation queue (lines 12-15). If it is that the case, we just compile the CFG to OpenCL code and swap the execution to the GPU. This is similar to the on stack replacement (OSR) technique currently implemented in the Oracle Hotspot.

```
 1  Object run(RFunction function, RVector input) {
 2    if (cache.lookup(function))
 3      function.getRootNode()).enableOCL();
 4    graph = cache.lookup(callTarget.getID());
 5    oclUnit = cache.lookup(graph);
 6    if (wasCompiled(graph))
 7      return runWithOCL(input, oclUnit);
 8    for (int i = 1; i < size; i++) {
 9      Object value = function.call(args);
10      output.add(value);
11      graph = cache.get(callTarget);
12      if (graph != null && oclUnit == null) {
13          oclUnit = compile(callTarget, graph);
14          cache.insert(graph, oclUnit);
15          return runWithOpenCL(input, oclUnit); }
16    }
17    return output; }
```

Listing 4: Run method in the AST interpreter for `OpenCLMApply` node.

4.3 Own AST root node

Truffle is a self-optimizing AST interpreter. This means that the AST is re-written itself and when it reaches a stable point, it compiles to native code. The AST interpreter is implemented in Java, therefore, for the JVM, the interpreter is just another Java application. When the partial evaluator compiles the AST, it compiles the interpreter (execution engine) itself with the actual values and constants.

```
class FunctionDefinitionNode {
  @Child private RNode body;
  @CompilationFinal boolean openCLExecution;
  private Object oclExecution(VirtualFrame vf) {
    return body.execute(vf);
  }
  @Override
  public Object execute(VirtualFrame frame) {
    if (openCLExecution) {
      return oclExecution(frame);
    } else {
      return fastRExecution(frame);
    }
  }
}
```

Listing 5: Sketch of the execution engine for the AST root node in our compiler

After partial evaluation there is still information about the AST interpreter that we have to get rid of. Handling an AST interpreter on GPUs will generate very irregular and complex kernels that can kill the parallelism. We create a kernel with the computation that was expressed in the guest language. In FastR we can easily influence in the Graal-IR generated after partial evaluation by simplifying the AST interpreter.

We have our own representation of the AST root node. Listing 5 shows a sketch of the logic for the `execute` method in the AST root node for FastR.

The root node is called `FunctionDefinitionNode` in FastR. We extend it with a simplified execution path for OpenCL. If the function to be executed is not candidate for OpenCL, it follows the default path for interpretation and compilation. But, it is candidate for running with OpenCL, it will execute a simplify version in the AST interpreter. This simplifies the CFG after partial evaluation, and, therefore, the OpenCL compilation.

4.4 Data types and type inference

We currently support the most common R data types: vectors of primitive data types for integer, double and `Boolean`. We also support sequences and R lists and NA (Not Available) values.

Type Inference: input, output and scope OpenCL is a static typed language which requires to infer the input/output types and the size before compilation. Moreover, OpenCL does not allow null references. If we detect null references, unknown types or sizes, we fall back (deoptimize) to the normal FastR execution and we do not compile to OpenCL. Therefore, the user application is always executed, but with different strategy.

We infer the input data types via Truffle specializations in the AST interpreter. We obtain the types and the size for all the input variables. For OpenCL execution, we also have to check that the input values are the same type and they do not contain `null` references.

To infer the output data types, we first execute the R function in the interpreter with only the first element from the input data set. Based on this execution we build all the meta-data needed for compiling to OpenCL.

We also support lexical scope variables automatically. We explore the AST and we build an array of the lexical scope variables for the function. Then, for each variable we infer its type and size and we also pass this meta-data to the OpenCL code generator.

R lists Lists in R can handle a dynamic list of different data types. The OpenCL programming model does not contain lists or any dynamic data structure. We decide to map them to a C `struct` of fixed size. We handle the R lists as a collection of *Tuples* in our compiler framework. We can safely do this because, at the partial evaluation time, we do know the length of any array and lists. If the R function returns a list, we build an internal Tuple with the number of elements. For instance, if the function returns a list with two elements $list(a, b)$, we build a $Tuple2(a, b)$.

NA values R is a languages designed for statistics. Therefore, it is very common to have missing or not available values in a data set. NA values are internally implemented as an integer in the interpreter. For instance, if the input vector is an integer array, an NA value is represented as the minimum value for the integers. This becomes an NA just as another integer value for OpenCL.

This simple strategy allows us to work with NA values coming from large input data sets. However, we still operate with the NA values inside the OpenCL kernel. Lukas not sure what that means□ Juan I want to say that, we manage NAs values but, as they are just numbers for OpenCL, we operate with them. I think a better strategy is to avoid the NA computation inside the OpenCL kernel (if value != NA then compute;), but we do not support this at the moment.□

5. OpenCL Code Generation

This section shows the internal process of our code generator and how we transform the AST to a valid OpenCL C code at run-time.

5.1 Compilation example

Figure 4 shows in detail how our compiler translate from the R to OpenCL C code and how the intermediate representation looks like. We use the same example showed in Listing 1. The top of the Figure represents the input R code, which is transformed to an AST in step 1. In this step, the AST is rewritten itself with specialized type information.

If the R function is executed multiple times, Truffle compiles the AST with Graal via partial evaluation. The output for the partial evaluation is a CFG in the Graal-IR form. The CFG showed in step 2 of the figure represents the Graal-IR for the `daxpy` function. The left side of the step 2 shows the corresponding Java code in the AST interpreter that produces the Graal-IR shown in the right part.

Then we mark the CFG as candidate graph for OpenCL compilation. We introduce annotations in our AST interpreter to deal with CFG filtering and clean up. This allows us to remove unnecessary interpreter overhead in the compilation step. We extend the partial evaluator with a few phases for processing our custom OpenCL annotations. If the AST interpreter contains our annotations, we safely remove the guards nodes associated to the CFG. In this example, all the inputs have the annotation `@KnownType`, which is required for OpenCL compilation. Therefore, if an input field has this annotation, we safely remove the condition and the guard associated to it. Lukas it's not clear here under what circumstances it is safe to use KnownType, and why the checks are there in the first place.□

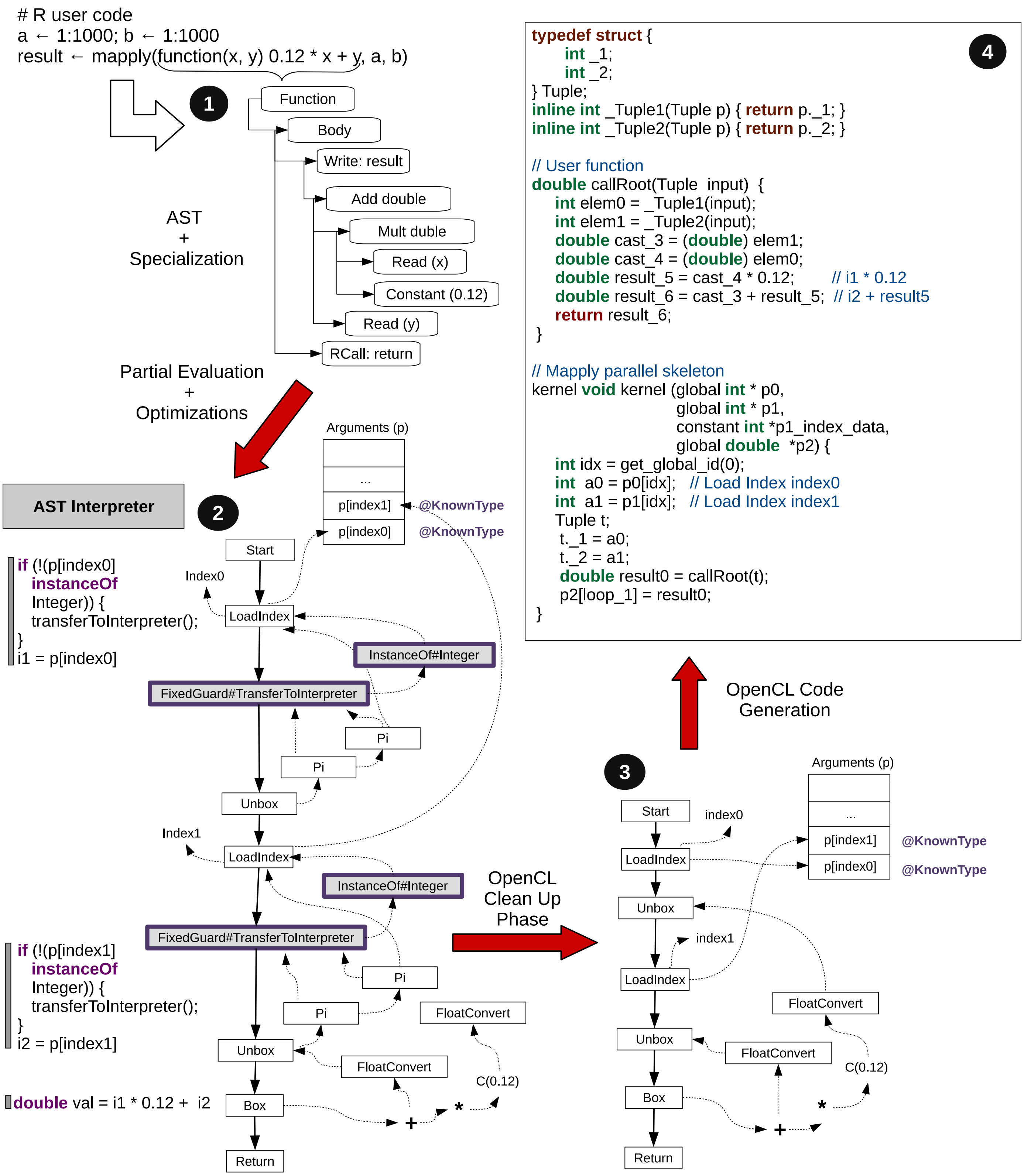
However, we keep the rest of the checks in order to preserve the semantic of the input language and deoptimise if it is required. As a result, we produce the CFG showed in step 3. Then we take this optimized CFG and generate OpenCL code using the existing code generator for OpenCL. The output of the code generator is showed in the step 4. As a result, we have a R expression running on GPUs with OpenCL.

5.2 Compiler information in the AST level

To identify the R function with the corresponding CFG to be compiled, we extend Truffle and Graal to pass information from the partial evaluator the to AST interpreter. The compilation unit in Graal is called *call target*. The call target includes the functions and methods to be compiled by Graal. The Graal IR has a unique number to identify the CFG, and we also extend the call target to include the same information. Our OpenCL JIT compiler keeps track of the call target and its CFG.

5.3 Simplifying overhead of the AST interpreter

Partial evaluation and self-optimizing interpreters help to reduce the compilation overhead of the interpreter itself. But the partial evaluation keeps the control flow of the interpreter in the compilation unit [12], which means that the partial evaluator not only compiles the user code written in the guest language, but also the interpreter itself. For instance, if we want to add two variables, the partial evaluator does not just compile the addition but also the logic in the interpreter to perform the addition. Lukas this needs an example - what's the "logic to perform the addition"?□



```
# R user code
a ← 1:1000; b ← 1:1000
result ← mapply(function(x, y) 0.12 * x + y, a, b)
```

```
typedef struct {
      int _1;
      int _2;
} Tuple;
inline int _Tuple1(Tuple p) { return p._1; }
inline int _Tuple2(Tuple p) { return p._2; }

// User function
double callRoot(Tuple  input)  {
    int elem0 = _Tuple1(input);
    int elem1 = _Tuple2(input);
    double cast_3 = (double) elem1;
    double cast_4 = (double) elem0;
    double result_5 = cast_4 * 0.12;        // i1 * 0.12
    double result_6 = cast_3 + result_5;  // i2 + result5
    return result_6;
 }

// Mapply parallel skeleton
kernel void kernel (global int * p0,
                              global int * p1,
                              constant int *p1_index_data,
                              global double  *p2) {
    int idx = get_global_id(0);
    int  a0 = p0[idx];   // Load Index index0
    int  a1 = p1[idx];   // Load Index index1
    Tuple t;
     t._1 = a0;
     t._2 = a1;
     double result0 = callRoot(t);
     p2[loop_1] = result0;
 }
```

```
if (!(p[index0]
   instanceOf
   Integer)) {
   transferToInterpreter();
}
i1 = p[index0]
```

```
if (!(p[index1]
   instanceOf
   Integer)) {
   transferToInterpreter();
}
i2 = p[index1]
```

```
double val = i1 * 0.12 +  i2
```

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

This overhead can be partially removed by using node specialization and lowering. Lukas lowering doesn't remove anything, it may expose opportunities to remove things□ Graal has different compilation levels for lowering. If we wait too long to compile, we find architecture dependent optimizations in the CFG. If we optimize too early, we still find many Graal nodes coming from the AST interpreter. Lukas this needs to start at a higher level: Graal's IR starts out in a form that is close to bytecode. It is successively lowered toward more low-level operations (e.g., from field load to memory read), and at each stage various optimizations are performed. the OpenCL compiler needs high-level operations, but still wants to take advantage of as many optimizations as possible. □

We analyze the different phases to generate code for OpenCL. We intercept every compilation phase in the high-tier to obtain a better approach to the real function computation in R. Based on an empirical evaluation, we see that after the last phase in the high-tier (non architecture optimizations) we obtain a good approach to the lambda expression we need to translate to OpenCL. Lukas this basically means: inlining, escape analysis and high-level loop optimizations have been applied, but no lowering of bytecode operations□

The resulting CFG contains the computation and a bunch of checks based on speculations and assumptions. As we will see in the next subsection, some of the speculations can be easily removed. R is a dynamic and interpreted language which, to generate machine code, uses speculations and assumptions. Lukas ... and in order to generate machine code, FastR uses spec and ass.□ However, for some parts of the R code we always know that the input data does not contain null references and its type is resolved.

In order to simplify more the CFG, we introduce a set of **annotations** to the methods and fields of the AST interpreter that allows the partial evaluator to remove redundant information. This technique is, in fact, generic for all the Truffle languages.

Truffle annotations for OpenCL Truffle DSL provides annotations for the language implementer to give more information to the compiler and produce better optimizations. We extend the DSL with a few annotations for OpenCL.

- `@NotNull`: it tells the compiler not to check for null references for a specific input value.
- `@KnownType`: it tells the compiler there is no need to check the input data type and deoptimise if the type was not the expected one.
- `@ArrayComplete`: it tells the compiler that the values are present and no NA values are found. This annotation is specific to the R semantic, however, other Truffle languages can benefit from it.

By annotating the AST interpreter with these annotations, we can safely simplify the CFG after partial evaluation. For instance, if a field is annotated with `@NotNull` as showed in Listing 6, the partial evaluator will remove the if condition (line 3) and the deoptimization (line 4) in the CFG.

```
1  @NotNull Object[] field;
2  ...
3  if (field == null) {
4          deoptimize();
5  }
```

Listing 6: Example of field annotation in the AST interpreter

We process these new annotations during the partial evaluation. If those annotations are present and the OpenCL mode is enabled, we clean the graph according to the semantics of the annotations.

6. Optimisations

This section presents the optimizations we implement in our compiler for an efficient management of vectors. We also present how our compiler framework handle the deoptimizations on GPUs.

6.1 R sequences

A sequence is an ordered collections of elements. Sequences are, in fact, an optimized data structure in FastR for vectors. Sequences do not store any value. Instead, they are computed by the following computation: $start + stride * index$. We take the same approach for OpenCL. If any input to the R function is a sequence, we in-line the formula in the OpenCL C code.

Listing 7 shows the effect of the optimization for the `Daxpy` application showed in Listing 1. This OpenCL code corresponds to the R computation $1 : size$. We know type of sequence and how to compute it at compiled, therefore we can in-line the access to the array by computing the same formula in OpenCL.

```
int x1 = 1 + (1 * idx);
int x2 = 1 + (1 * idx);
```

Listing 7: Sketch of OpenCL C code for R sequences

This technique has two clear advantages: a) OpenCL buffers and data transfers between the OpenCL host and device are completely avoided. We just create the buffer for the two elements we need (start, stride) independently of the input array size; b) accesses to the OpenCL device global memory for these arrays are totally removed. This has a clear positive effect in performance as we will see in the next section.

6.2 PArray for R vectors

`PArray` (Portable Array) is a data structure presented in [5] for avoiding marshaling and unmarshalling of arrays between Java and OpenCL. We extend the `Parrays` data structure for the R programming language. The `PArray` logic is very straightforward. When a PArray is created, there is an internal buffer in the target architecture format (OpenCL in this case).

We extend the PArray for the Truffle languages. The arrays in R are stored in primitive type arrays. We extend the PArray API to pass those primitive arrays to the OpenCL device. Therefore, we map the arrays to a reference that the OpenCL Java wrapper can read and write directly. This strategy completely avoids the data type transformation between R and OpenCL C. We will see in the evaluation section the impact of this optimization in the total performance of the R application.

6.3 Deoptimizations

Truffle speculates over the types and branches based on runtime profiler information. The partial evaluator introduces guards or check points for the speculations. Those guards are useful because the profiler information might not be complete at compilation time, therefore a deoptimization will be performed.

Deoptimization process in OpenCL We keep those guards in the OpenCL code. Listing 8 shows an example in R of an application with a branch that depends on an input value. If the input value is less than 1, it returns 0, otherwise it returns 1.

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

Listing 8: Simple R function with a branch for speculation

Let's assume an input array of 1000 values, with all zeros except the last position, with 1. Let's also assume that the code is going to be compiled after 999 iterations in the AST and, during the computation in the last iteration, the AST is compiled to OpenCL. This means that the profiler never registered an execution of the false condition at compilation time during the AST interpretation. Consequently, the partial evaluator produces a CFG with the true condition plus a condition to check if the false condition was reached.

```
1  double callRoot(double p0,__global int*
       ↪ deoptFlag) {
2    bool cond1 = p0 < 1.0;
3    if(!cond1) deoptFlag[0] = get_global_id(0);
4    return 0.0;
5  }
```

Listing 9: OpenCL C code generated for the R program in Listing 8.

The generated OpenCL code for the R function in Listing 8 is showed in Listing 9. Note that the entire block in the false condition is removed and it is substituted by a check and a flag. Line 4 sets the deoptFlag.

GPUs are parallel hardware with no control for exceptions or hardware traps. Therefore, we have to wait until the kernel execution finishes. When it is finished, we check the deopt flag. If there is no deoptimizations (not equal to -1), the speculations are correct. Otherwise, we invalidate the OpenCL binary and we execute the application in the AST interpreter again. If the application keeps running in the interpreter, it follows the normal procedure for compilation but this time with better profiling information.

We register the last identifier (thread ID in OpenCL) that provokes the deoptimization. Note that these buffers are shared among the all threads, therefore is not thread safe. However, we are interested in knowing one of the threads that provokes the deoptimization. Then we re-execute the R program in the AST interpreter with the ID that provokes the deoptimization in order to register profiler information in all the branches affected. Then, the code is recompiled to OpenCL with the updated profiling information. The kernel we finally generate is showed in Listing 10.

```
double callRoot(double p0,__global int*
    ↪ deoptFlag) {
  bool cond1 = p0 < 1.0;
  if (cond1) return (0.0);
  else return (1.0);
}
```

Listing 10: OpenCL C code generated for the R program in Listing 8 after re-profiling

Our technique is a very simple strategy to deoptimise with minimal overhead in the OpenCL kernel.

Deoptimization Evaluation We execute the R program described in Listing 8. We set-up the compile threshold to 1000 to compile to OpenCL. We create an array of 1000 elements, all set to zeros with the exception of the last element (to one). Note that when the R code is partially evaluated, there is no profiler information associated to the false condition.

Figure 5 shows the time in milliseconds to execute the described R program. The total time is 320 milliseconds. During the first 90 milliseconds the R program runs on the AST interpreter. Then, the R function starts to compile to OpenCL, taking 70 milliseconds. We then execute the kernel and check if deoptimization flag is enabled. In this case, this flag is enabled because it should have taken the branch that was not generated. We run the R program again in the AST interpreter to re-profile and rewrite the AST (another 80 milliseconds), and the we see that the new version is recompile and executed with OpenCL.

7. Evaluation

This Section presents the performance evaluation of our OpenCL JIT compiler framework the R language on the AMD and NVIDIA GPUs. We first describe our set-up and the benchmarks we use. Then we show our performance results and analysis.

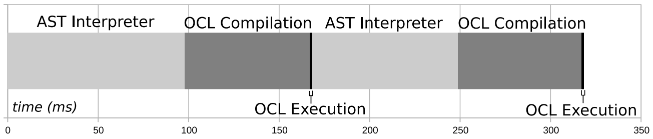


Figure 5: Total time breakdown in seconds of an R application when performing a deoptimization.

7.1 Evaluation set-up

We evaluate our compiler infrastructure on two different platforms. Both have a four core Intel i7 4770K @ 3.50GHz with 16GB of DDR3 RAM. The first platform contains a GPU AMD Radeon R9 295X2 with 8GB of GDDR5 memory. The second platform has a GPU GeForce GTX TITAN Black with 6GB of GDDR5 memory. We use the GPU drivers AMD 1598.5 and Nvidia 367.35.

Virtual Machines We compile FastR with Java 1.8. We compile our R interpreter with Graal 0.9. We provide a comparison between FastR, GNU R and OpenCL C++ and our VM infrastructure. It is hard to make a fair comparison between GNU R because and FastR because it is a different VM and JIT compiler. We compare each version with GNU R 3.3.1 with $enableJIT$ level 3.

Measurements We execute each benchmark 10 times for all the benchmarks and we report the peak performance with the median time. We execute FastR as well as our R extension for OpenCL with 12GB of Java heap memory. We measure our R benchmarks using a custom built-in, `nanotime`, which internally calls to the `System.nanotime()` for FastR and our compiler (we extended FastR with this call). For the GNU R, we measure by calling the `proc.time()` function. All the timers reported are end-to-end.

7.2 Benchmarks

To evaluate our JIT compiler and execution enviroment we ported a set of benchmarks from Rodinia [1], AMD OpenCL SDK[1] and programming language benchmarks[2] to R. We choose the benchmarks that represent most data and computing intensive. We take 8 benchmarks for different domains: `Daxpy`, `Black-scholes`, `NBody`, `DFT`, `Mandelbrot`, `Kmeans`, `Hilbert Matrix` and `Spectral Norm`.

We execute each benchmark with the data sizes showed in Table 1. We choose a representative input size for FastR, big enough for OpenCL GPU computation. However, these data sizes are still very small for the GPUs. As we will see in the analysis section, even with this moderate data size on GPUs, we obtain large speedups.

7.3 Performance Results

We now analyse the performance results we obtain for all the benchmarks in R and OpenCL. We first compare the effect of our optimisations. Then we compare our compiler and runtime with the related work and native implementations. We conclude by a showing a comparison that imitates a real scenario for the R programmers, where the application is executed just once.

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	128	128
Spectral Norm	0.256	0.256

Table 1: Benchmarks and sizes we use to evaluate our OpenCL JIT compiler framework for R.

Optimisations impact In this section we analyse the effect of our optimisations presented in section 6. Figure 6 shows our three configuration on the two GPUs. The top of the Figure shows the speedup over our basic GPU version on AMD. The bottom shows the speedups for the NVIDIA GPU. The leftmost shows the full version where we marshal and unmarshal the data. In this case we pay the cost of transforming the data between R and OpenCL. The second bar shows the speedup we obtain when we represent the `RVector` in the GPU representation as described in the previous section. The last bar shows the speedup when the input data can be also represented as a sequence. It the case of `DFT`, `mandelbrot`, `hilbert` and `spectral norm`.

Using the PArray we obtain 3x speedup over the baseline and, by adding the sequences, we obtain 3.8x in avarage. Sequences are specially beneficial for 2D kernels. That is the case of `mandelbrot` and `hilbert`, where we obtain higher speedups.

Comparison with FastR, GNU-R and native OpenCL C++ Figure 7 shows the performance results for each benchmark. The leftmost bar of each benchmark represents the speedup of GNU R over FastR. The second bar represents our baseline, FastR. The third bar shows the speedup with our best version for OpenCL executed on the AMD and NVIDIA GPUs. The last bar corresponds to the speedup of the native implementation implemented in OpenCL C++.

FastR is 10 to 100 times faster than GNU R for these data computing intensive benchmarks, being in average 10x faster than GNU-R in peak performance.

On the GPUs, our approach is 150x times faster than FastR, and it is 1000x faster than GNU R on AMD GPU and 1300x faster on NVIDIA GPU. Note that the R input code is exactly the same compared to FastR. When comparing to the native version, our compiler and execution environment

[1] `goo.gl/28AAUY`

[2] `http://benchmarksgame.alioth.debian.org/`

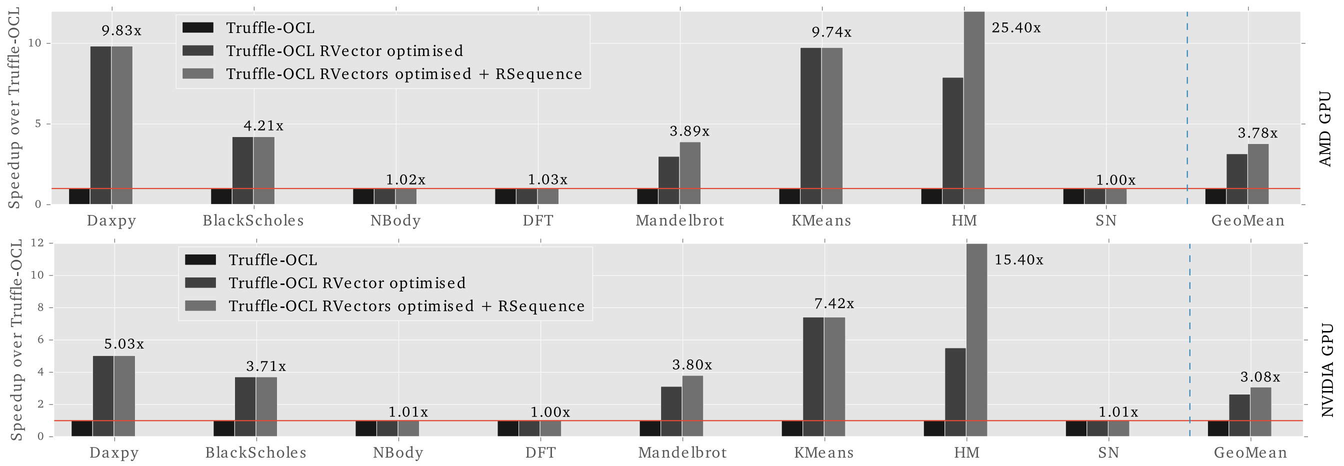


Figure 6: Performance of our optimisations for the R programming language on GPUs over our basic OpenCL version with no optimizations. The higher the better.

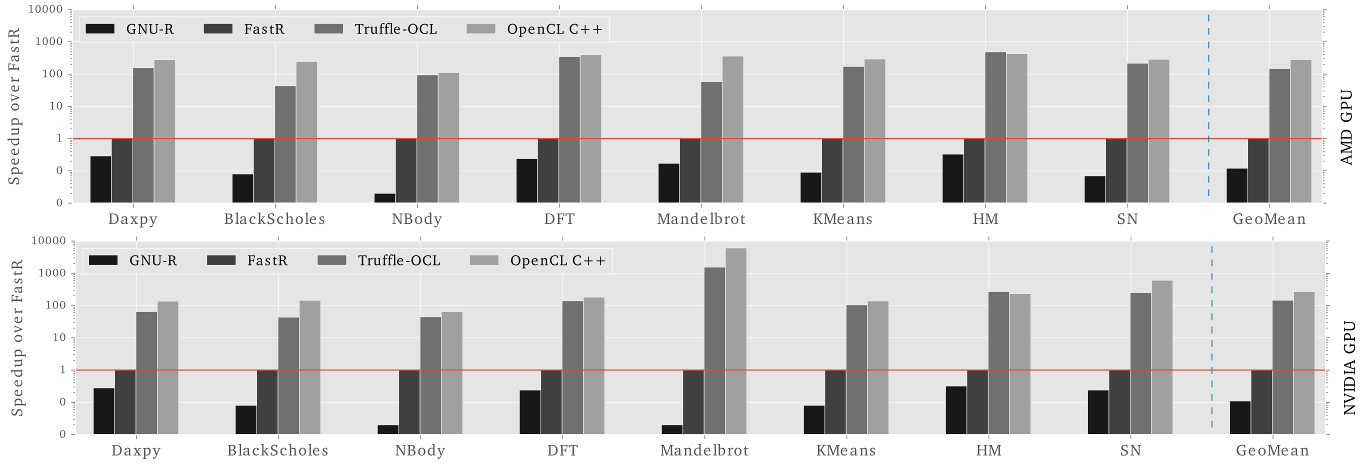


Figure 7: Speedup over Oracle FastR of our R compiler and VM framework for OpenCL compared against GNU-R, FastR, Java and OpenCL C++. The higher, the better.

is 1.8x slower than OpenCL C++ in peak performance on AMD GPU and NVIDIA GPU.

For the `nbody`, `dft` and `spectralNorm` benchmarks we obtain similar performance. For `hilbert`, our approach is faster than OpenCL C++ because we use the optimisation sequence to index the input data, saving, therefore, the data transfers.

Although we can easily execute run applications on the GPU, the OpenCL kernels can be optimized. The kernel we generate is a generic OpenCL kernel directly from the Graal-IR and we do not currently explore OpenCL optimisations. For managed languages like R, the bottleneck is in the data management rather than the kernel itself. Moreover, in most of the benchmarks, the generic OpenCL kernel is close enough to the OpenCL C++. In fact, in comparison with FastR, the speedup of our approach is in the same order of magnitude than OpenCL C++.

Cold-run The typical scenario for the R users is to write the R program and execute it only once. Figure 8 shows the speedup of our OpenCL-R execution framework over FastR for ten different sizes representing this execution mode. This Figure does not show peak performance. We report the time, end-to-end, that takes to execute the full R application. We named it *cold-run*, because we start with a fresh VM. The Figure shows two lines: the blue line shows the speedup on AMD GPU and the red line on NVIDIA GPU. The x-axis shows the time, in seconds, that takes the R application for the FastR version (no GPU). The y-axis represents the speedup for each GPU architecture over the FastR version. As we see, after 3 seconds of execution within the R code, it starts to obtain speedup with our approach. Both GPUs show similar performance in cold-run mode with an average speedup of 57x on AMD GPU and 47x on NVIDIA GPU, ranging from 17x to 120x.

Compilation time During the cold-run, most of the time is spending in the interpreter and OpenCL JIT compilation. The execution time in the AST interpreter, partial evaluation and OpenCL code generation ranges from 100 milliseconds up to 2 seconds. The OpenCL code generation takes, in average, 600ms. If the execution is, in fact, less than 0.6 seconds, there is no benefit to execute on GPU.

8. Related Work

This section reviews the most relevant R projects as well as related work in other programming languages.

8.1 Parallel built-ins

Haichuan Wang et. al. [19] have a similar idea of vectorizing the R `apply` functions. Wang shows an algorithm to vectorize the R and transform the looping-over data functions for the apply methods in the R interpreter. Wang argues that his framework focuses in the apply operations for similarity and compatibility to map-reduce frameworks and big data. However, we show that the JIT compilation to OpenCL is worth, and has more impact in performance when the R application runs in the interpreter for more than 3 seconds.

8.2 R JIT Compilers Implementations

Good performance is crucial for data analytic where a lot of data can be processed in parallel. The way of achieving good performance is through specialization and JIT compilation.

As there is no formal specification of the R language, the reference implementation (GNU R) is the de-facto specification. However, GNU R interpreter is slow.

There are a few projects that either modified the existing GNU R compiler. Orbit VM [20] optimizes the byte-code interpreter. Revolution R[3], pqR[4] and TIBCO R[5] improve the vector type operations and native libraries. Rho[6] is a reimplementation of the R interpreter on top of LLVM. RLLVM[7] is an R interface for LLVM, allowing to optimize and compile R code thought the LLVM IR.

There are other projects that build a new R VM with the purpose of taking advantage of the JIT compilation. Renjin[8], Riposte [18] and FastR [17] compile R user code to an efficient native machine code. As far as we know, none of these implementations provides an OpenCL or CUDA JIT compiler as we present in this paper.

8.3 R libraries for GPUs

There are many R projects that offer CUDA and OpenCL execution. However, there is no OpenCL or CUDA JIT compilers for R yet. If R programmers want to execute heterogeneous applications they have to call to native and statically compiled libraries or provide the kernels as a wrapper for OpenCL or CUDA.

GPUR[9] is a library for GPU programming. It provides native functions for vector and matrix operations in CUDA and OpenCL. Internally it uses ViennaCL [16], a library for GPU programming based on parallel skeletons. There is no code generation and there are direct calls to the native C libraries for CUDA and OpenCL. Programmers require to know concepts such as OpenCL *platform* and OpenCL *device*. We believe that, although R users can benefit from the GPU speedups, this approach still remains very low-level for most of the R programmers.

GMatrix[10] and *GPU Tools*[11] are libraries for accelerating R matrix operations with CUDA. It implements common vector and matrix operations such as matrix multiplication, addition, subtraction, sorting and trigonometric functions. With GMatrix, the programmer explicitly creates and manipulates GPU objects in R.

Rth[12] is an interesting project which provides an R interface for CUDA Thrust. The package executes parallel code for three different back-ends for the same interface: Cuda, OpenMP and Intel TBB.

OpenCL for R[13] is a wrapper that exposes the OpenCL API to R. However, it changed the standard OpenCL API compared to the OpenCL standard. Thus programmers need to learn a new non-standard API. The user also provides the OpenCL kernel as a string in R that will be compiled by the OpenCL driver. This is, in fact, a very low-level approach where the parallelism is fully exposed to the R programmers. Moreover, because of the OpenCL kernel is expressed in C as an R string, it increases the complexity and manageability of the code.

There are other projects specialized for statistic computation on GPUs. *RPUD* and *RPUDPLUS*[14] are open source R packages for performing statistical computation using CUDA. RPUD implements vector operations and Bayesian classification algorithms. In this case, the R programmer imports the library and uses the GPU with predefined functions. Those functions call directly to the GPU.

8.4 Other languages

There are a few compilers for OpenCL available for languages such as LINQ, Python, Java and Javascript.

[3] `https://www.microsoft.com/en-us/cloud-platform/r-server`

[4] `http://www.pqr-project.org/`

[5] `https://tap.tibco.com`

[6] `https://github.com/rho-devel/rho`

[7] `https://github.com/duncantl/Rllvm`

[8] `http://www.renjin.org/`

[9] `https://cran.r-project.org/web/packages/gpuR/index.html`

[10] https://cran. r-project.org/web/packages/gmatrix/index.html

[11] `https://github.com/nullsatz/gputools`

[12] `http://heather.cs.ucdavis.edu/~matloff/rth.html`

[13] `https://cran.r-project.org/web/packages/OpenCL/`

[14] `https://cran.r-project.org/src/contrib/Archive/rpud/`

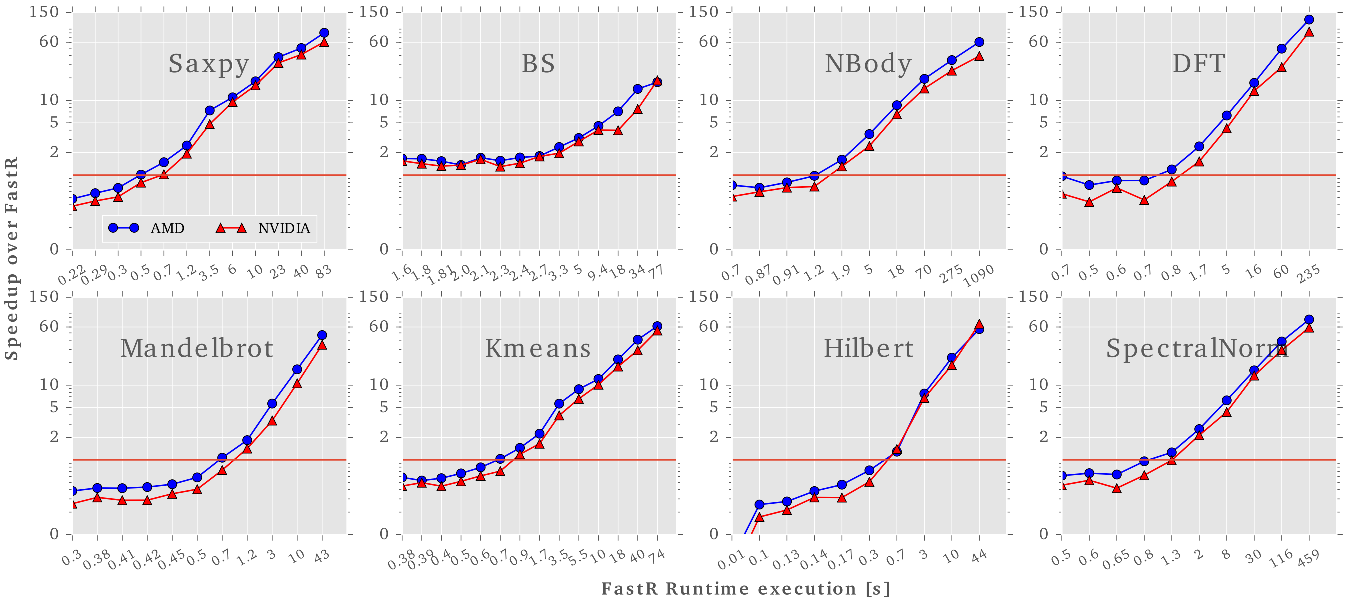


Figure 8: Speedup of the first run over FastR for all the benchmarks running on AMD and NVIDIA GPUs with our compiler infrastructure. The higher the better.

Bytespresso [2] is a DSL on top of Java language for heterogeneous computing. It uses an AST to as an intermediate representation. Bytespresso uses annotations for extracting the parallelism for CUDA applications. Our approach is language agnostic, and our annotations are generic to other truffle languages independently of OpenCL.

Numba [11] is a CUDA JIT compiler for python. Numba GPU compilation is based on annotations. The programmer annotates the functions to tell the compiler which code can be parallelized. With our approach we do not use any annotation, just the default built-ins in the language.

LINQ Cuda JIT from Quant Alea[15] is a very promising approach that allows data base applications to use heterogeneous hardware. However, programmers still have to adapt the code by using explicit GPU data structures and control in which device to execute the application. Although it offers to the programmer a fine control about the architecture, we believe programmers does not have to know necessary hardware details.

There are a few OpenCL/CUDA JIT compilers for Java such as *AMD Aparapi*[16], *Rootbeer* [15], and *JaBBE* [22]. However, in all of these cases, the programmers need to modify the base code by implementing a GPU class and creating an `execute` method. This approach still remains low-level even though the user applications are implemented in high-level programming languages. *Sumatra* is a very promising project in OpenJDK to automatically generate HSAIL code at runtime for Java 8 Streams. As far as we know, Sumatra project is no longer maintainable. *Harlan-J* [14] 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 always need to change the code base and adapt it to the new technologies. Our approach is totally agnostic about the technology, and all the code transformations happen automatically at runtime without the programmer intervention.

9. Conclusions

We have presented an OpenCL JIT compiler for optimizing AST interpreters in Truffle. We implemented our technique for the R language. To the best of our knowledge, this is the first OpenCL JIT compiler for R. We have shown the issues for generating high performance OpenCL code from managed languages. We show that, the combination of specialization, compiler lowering and annotations in the language, help to reduce the compiler overhead. We also present some opportunities for optimizations such as `RSequences` with no array copy and how to handle speculation failures on GPUs through deoptimizations. We showed that our OpenCL JIT compiler is, in average, 150x times faster than FastR and 1.8 slowdown compared to best OpenCL C++.

We currently support a subset of the R language into the GPU. We plan to support more features for the R programming language such as dealing with data frames and formulas. We plan to introduce other Truffle languages such as Ruby or JavaScript for OpenCL by using the same compiler techniques presented in this paper.

[15] `http://quantalea.azurewebsites.net/`

[16] `http://aparapi.github.io/`

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] S. Chiba, Y. Zhuang, and M. Scherr. Deeply Reifying Running Code for Constructing a Domain-Specific Language. PPPJ 2016.

[3] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1991.

[4] 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.

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

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

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

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

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

[10] 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.*

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

[12] S. Marr and S. Ducasse. Tracing vs. Partial Evaluation: Comparing Meta-compilation Approaches for Self-optimizing Interpreters. OOPSLA 2015.

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

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

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

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

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

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

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

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

[21] 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.

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