VRIR specifications

Rahul Garg
McGill University

1 Introduction

Array-based languages, such as MATLAB and Python's NumPy library, usually share some common features. For example, both MATLAB and Python have matrix multiplication built-in to the language and offer support for array slicing operators. Details of the exact semantics of some array operators may vary slightly, but the basic ideas are substantially similar. Languages like MATLAB and especially Python also offer many general-purpose language constructs (such as classes), but we are not interested in those constructs in this document. We focus only on the numerical portions of the program, which typically use arrays, array-operators and common control structures such as for-loops, if/else statements and functions.

Recently, there has been some interest and activity in building compilers for array-based languages to GPUs and multi-cores. Such compilers have typically focused on compiling numerical sections of the program because numerical sections are often the most performance critical section of scientific programs and also the most amenable to acceleration using a GPU. The challenges of efficiently compiling the array-based parts of a program to CPUs and GPUs are similar across languages. However, compiler development for these languages currently occurs in isolation. Work done on a Python compiler does not benefit the MATLAB community and vice-versa.

We want to build a common compiler toolkit, called Velociraptor, that can be reused by different compiler writers for compiling numerical parts of a program written in array-based languages. Velociraptor is dynamic code-generation toolkit that generates CPU and GPU code from VRIR. The first step of building such a compiler infrastructure is the specification of a common intermediate representation (IR). We propose an IR called VRIR (abbreviated for Velociraptor IR) for representing the numerical computation parts of the program. The idea is as follows. A dynamic compiler for a array-based language will perform analysis such as type inference, identify numerical sections of the program and then generate VRIR for numerical parts of the program. Non-numerical parts can be handled in any suitable way. Numerical sections compiled to VRIR will be passed to Velociraptor, which will perform code-generation for CPUs and GPUs. With this background, we now present the design and specifications of VRIR.

2 Design overview

We present some design choices we made for VRIR.

1. VRIR is an abstract syntax tree (AST) based notation. AST based notations are flexible yet simple to understand and relativley simple to generate, making them potentially accessible to many compiler writers.

2. VRIR is a typed IR. Every symbol and every expression in a VRIR program must have an assigned type. The idea is that VRIR will be generated by a compiler after it has done type checking and/or type inference, depending on the source language. If all the types cannot be successfully inferred in the relevant sections of the program, then it is unlikely that such parts will be efficiently compilable.

3. VRIR programs do not have a "main" or entry function. VRIR is not meant to represent the entire application, only some numerical functions. VRIR functions are meant to be called as computation routines by the application.

4. VRIR is designed to support mixed CPU/GPU systems. By default, all the statements in a VRIR function execute on the CPU. However, any statement list in VRIR can be annotated as GPU-accelerated. Velociraptor uses such annotations as hints and tries to compile the annotated statements to a GPU. However, such annotations are only hints and Velociraptor is free to ignore them. Velociraptor is free to generate CPU code instead of GPU code for the annotated statements, provided the result of the computation is not changed.

5. VRIR does not expose the architectural details of any CPU or GPU. We do not expose the number of cores, cache size, GPU memory size or other such details in the IR. VRIR is architecture-neutral. All architecture-specific optimizations are solely Velociraptor's responsibility.

6. VRIR does not expose the details of CPU/GPU data transfer. VRIR only allows the marking of certain computations as GPU-friendly, but identifying the data transfers required to execute a computation on a CPU/GPU system is solely Velociraptor's responsibility. Exposing the data transfer details in VRIR would go against the previously stated principle of not exposing the architectural details in the IR, as the data transfers required depend on many details of the system. For example, data transfers may not be required on a system that implements shared memory between CPU and GPU, but will be required on other systems. Similarly, the size of the on-board memory on a discrete GPU may affect the lifetime of big arrays on the GPU. Thus, we chose to not expose the details of the data transfers in VRIR.

7. VRIR has rich support for arrays. VRIR supports multidimensional arrays. Arrays can have row-major, column major or strided layouts. Arrays

can be indexed in flexible ways, including creation of slices. Many array operators, such as matrix multiplication, are built-in. Element-wise array addition, subtraction, multiplication etc. is also supported. Reduction operators like sum and product, that reduce multidimensional arrays about a given axis dimension, are also supported.

8. VRIR supports common statements such as assignment statements, if/else statements, for-loops and while-loops. In addition, we also support a parallel-for loop.

9. VRIR balances safety and performance. Array accesses are optionally bound checked, but bound-checks can be disabled for performance if required.

10. VRIR typesystem can be classified into four categories: scalars (integers, floats and complex types), arrays (with specified element-type, number of dimensions and layout type), functions (with specified input and output types) and domains (a datatype for specifying ranges of iteration domains). We focus on only numerical programs and do not support user-defined datatypes nor datatypes like strings or dictionaries.

11. In the current release of the specification, functions are not first class. Variables (including function parameters) cannot be assigned to a function. We are considering allowing functions to be passed as function parameters in the next release of the specification because some important application domains (like numerical optimization) require such functionality.

3 VRIR AST specifications

We describe the AST structure using an EBNF-inspired notation. This is not a grammar for a parser but rather it is meant to concisely describe the attributed AST structure of VRIR constructs. Some nodes have attributes, such as a string specifying a name. Attributes are indicated in round brackets '()' in non-italicisized font and are indicated using pseudo-Java types String, bool and int for string, integer and boolean respectively.

Velociraptor [1] implementation maps the AST nodes to C++ classes and provides suitable constructors to construct the AST objects. Velociraptor has also implemented a reader for an XML-based textual representation of VRIR. Users (i.e. compiler writers) can choose either method (direct C++ constructors, or XML input) to construct VRIR.

$\langle Module \rangle$ ::= (String name) $\langle Function \rangle$*

$\langle Function \rangle$ ::= $\langle FuncType \rangle$ (String name) $\langle FuncType \rangle$ $\langle SymTable \rangle$ $\langle ArgList \rangle$ $\langle StmtList \rangle$

$\langle \mathit{StmtList} \rangle$ ::= (bool onGpu) $\langle \mathit{Statement} \rangle$*

$\langle \mathit{Statement} \rangle$ = $\langle \mathit{StmtList} \rangle$
| $\langle \mathit{AssignStmt} \rangle$
| $\langle \mathit{IfElseStmt} \rangle$
| $\langle \mathit{ForStmt} \rangle$
| $\langle \mathit{WhileStmt} \rangle$
| $\langle \mathit{ParforStmt} \rangle$
| $\langle \mathit{ReturnStmt} \rangle$
| $\langle \mathit{ExprStmt} \rangle$
| $\langle \mathit{BreakStmt} \rangle$
| $\langle \mathit{ContStmt} \rangle$
| $\langle \mathit{RefIncrStmt} \rangle$
| $\langle \mathit{RefDecrStmt} \rangle$

$\langle \mathit{ForStmt} \rangle$ ::= $\langle \mathit{NameExpr} \rangle$+ $\langle \mathit{DomainExpr} \rangle$ $\langle \mathit{StmtList} \rangle$

$\langle \mathit{ParforStmt} \rangle$ ::= $\langle \mathit{NameExpr} \rangle$+ $\langle \mathit{DomainExpr} \rangle$ $\langle \mathit{StmtList} \rangle$

$\langle \mathit{WhileStmt} \rangle$ ::= $\langle \mathit{Expr} \rangle$ $\langle \mathit{StmtList} \rangle$

$\langle \mathit{AssignStmt} \rangle$ ::= ($\langle \mathit{NameExpr} \rangle$ | $\langle \mathit{ArrayWriteExpr} \rangle$)+ ($\langle \mathit{Expr} \rangle$|$\langle \mathit{FuncallExpr} \rangle$)

$\langle \mathit{IfElseStmt} \rangle$ ::= $\langle \mathit{Expr} \rangle$ $\langle \mathit{StmtList} \rangle$ $\langle \mathit{StmtList} \rangle$?

$\langle \mathit{ReturnStmt} \rangle$ ::= $\langle \mathit{NameExpr} \rangle$+

$\langle \mathit{ExprStmt} \rangle$ ::= $\langle \mathit{Expr} \rangle$|$\langle \mathit{FuncallExpr} \rangle$

$\langle \mathit{RefIncrStmt} \rangle$ ::= $\langle \mathit{NameExpr} \rangle$

$\langle \mathit{RefDecrStmt} \rangle$ ::= $\langle \mathit{NameExpr} \rangle$

$\langle \mathit{Expr} \rangle$ ::= $\langle \mathit{NameExpr} \rangle$
| $\langle \mathit{BinaryExpr} \rangle$
| $\langle \mathit{ArrayReadExpr} \rangle$
| $\langle \mathit{ArrayWriteExpr} \rangle$
| $\langle \mathit{ConstExpr} \rangle$
| $\langle \mathit{NotExpr} \rangle$
| $\langle \mathit{NegateExpr} \rangle$

$\langle \mathit{FuncallExpr} \rangle$ = $\langle \mathit{NameExpr} \rangle$ $\langle \mathit{Expr} \rangle$+

$\langle \mathit{RangeExpr} \rangle$ ::= $\langle \mathit{Expr} \rangle$ $\langle \mathit{Expr} \rangle$? $\langle \mathit{Expr} \rangle$

$\langle \mathit{DomainExpr} \rangle$ ::= $\langle \mathit{RangeExpr} \rangle$+

⟨*NameExpr*⟩ ::= (int id)

⟨*BinaryExpr*⟩ ::= ⟨*PlusExpr*⟩
  | ⟨*MinusExpr*⟩
  | ⟨*MulExpr*⟩
  | ⟨*DivExpr*⟩
  | ⟨*MatMulExpr*⟩
  | ⟨*LeqExpr*⟩
  | ⟨*GeqExpr*⟩
  | ⟨*LtExpr*⟩
  | ⟨*GtExpr*⟩
  | ⟨*AndExpr*⟩
  | ⟨*OrExpr*⟩

⟨*NegateExpr*⟩ ::= ⟨*NType*⟩ ⟨*Expr*⟩

⟨*NotExpr*⟩ ::= ⟨*NType*⟩ ⟨*Expr*⟩

⟨*PlusExpr*⟩ ::= ⟨*NType*⟩ ⟨*Expr*⟩ ⟨*Expr*⟩

⟨*MinusExpr*⟩ ::= ⟨*NType*⟩ ⟨*Expr*⟩ ⟨*Expr*⟩

⟨*MulExpr*⟩ ::= ⟨*NType*⟩ ⟨*Expr*⟩ ⟨*Expr*⟩

⟨*DivExpr*⟩ ::= ⟨*NType*⟩ ⟨*Expr*⟩ ⟨*Expr*⟩

⟨*MatmultExpr*⟩ ::= ⟨*NType*⟩ ⟨*Expr*⟩ ⟨*Expr*⟩

⟨*LeqExpr*⟩ ::= ⟨*NType*⟩ ⟨*Expr*⟩ ⟨*Expr*⟩

⟨*GeqExpr*⟩ ::= ⟨*NType*⟩ ⟨*Expr*⟩ ⟨*Expr*⟩

⟨*LtExpr*⟩ ::= ⟨*NType*⟩ ⟨*Expr*⟩ ⟨*Expr*⟩

⟨*GtExpr*⟩ ::= ⟨*NType*⟩ ⟨*Expr*⟩ ⟨*Expr*⟩

⟨*AndExpr*⟩ ::= ⟨*NType*⟩ ⟨*Expr*⟩ ⟨*Expr*⟩

⟨*OrExpr*⟩ ::= ⟨*NType*⟩ ⟨*Expr*⟩ ⟨*Expr*⟩

⟨*ArrayReadExpr*⟩ ::= ⟨*NType*⟩ (bool negativeMode) (bool flattenIndex) (bool enableCheck) ⟨*NameExpr*⟩ (⟨*Expr*⟩ | ⟨*RangeExpr*⟩)+

⟨*ArrayWriteExpr*⟩ ::= ⟨*NType*⟩ (bool negativeMode) (bool flattenIndex) (bool enableCheck) ⟨*NameExpr*⟩ (⟨*Expr*⟩ | ⟨*RangeExpr*⟩)+

⟨*ConstExpr*⟩ ::= (int ival | float fval | double dval)

$\langle \mathit{SymTable} \rangle ::= \langle \mathit{Sym} \rangle *$

$\langle \mathit{Sym} \rangle ::= \text{(int id) (String name)} \ \langle \mathit{VType} \rangle$

$\langle \mathit{ArgList} \rangle ::= \langle \mathit{Arg} \rangle *$

$\langle \mathit{Arg} \rangle ::= \text{(bool restrict) (int id)}$

$\langle \mathit{VType} \rangle ::= \langle \mathit{NType} \rangle$
$\quad | \quad \langle \mathit{VoidType} \rangle$
$\quad | \quad \langle \mathit{FuncType} \rangle$
$\quad | \quad \langle \mathit{DomainType} \rangle$

$\langle \mathit{NType} \rangle ::= \langle \mathit{ScalarType} \rangle$
$\quad | \quad \langle \mathit{ArrayType} \rangle$

$\langle \mathit{ArrayType} \rangle ::= \text{(int ndims)} \ \langle \mathit{Layout} \rangle \ \langle \mathit{ScalarType} \rangle$

$\langle \mathit{Layout} \rangle ::= \langle \mathit{RowMajorLayout} \rangle$
$\quad | \quad \langle \mathit{ColMajorLayout} \rangle$
$\quad | \quad \langle \mathit{StrideLayout} \rangle$

$\langle \mathit{ScalarType} \rangle ::= \langle \mathit{Int32} \rangle$
$\quad | \quad \langle \mathit{Int64} \rangle$
$\quad | \quad \langle \mathit{Float} \rangle$
$\quad | \quad \langle \mathit{Double} \rangle$
$\quad | \quad \langle \mathit{Complex} \rangle$
$\quad | \quad \langle \mathit{Boolean} \rangle$

$\langle \mathit{FuncType} \rangle ::= \langle \mathit{FuncInType} \rangle \ \langle \mathit{FuncOutType} \rangle$

$\langle \mathit{FuncInType} \rangle ::= \langle \mathit{NType} \rangle *$

$\langle \mathit{FuncOutType} \rangle ::= \langle \mathit{VoidType} \rangle$
$\quad | \quad \langle \mathit{NType} \rangle +$

The VRIR specification is a work-in-progress and I am quite open to suggestions.

References

[1] Laurie Hendren Rahul Garg. A compiler toolkit for array-based languages targeting CPU/GPU hybrid systems. Technical Report SABLE-TR-2012-3, Sable Research Group, School of Computer Science, McGill University, Montréal, Québec, Canada, November 2012.