Design of VRIR

Rahul Garg and Laurie Hendren
School of Computer Science
McGill University

A key design decision in our approach in designing Velociraptor compiler toolkit is that the numeric computations should be separated from from the rest of the program, and that our IR should concentrate on those numerical computations. This decision was motivated by an informal study of thousands of MATLAB programs using an in-house static analysis tool. We have found that typically the core numeric computations in a program use a procedural style and mostly use scalar and array datatypes. We have based the design of VRIR on this study and on our own experiences in dealing with scientific programmers.

VRIR is a high-level, procedural, typed, abstract syntax tree (AST) based program representation. Each node in the AST may have certain attributes and may have children. Attributes of a node can be of four types: integers (such as integer id of a symbol), boolean, floating point (such as value of a floating-point constant occuring in the code) or a string (such as a name). We have defined C++ classes corresponding to each tree node type. We have also defined an S-expression based textual representation of VRIR. Containing compilers can generate the textual representation of VRIR and pass that to Velociraptor, or alternatively can use the C++ API directly to build the VRIR trees. The syntax for the textual representation for a VRIR node consists of the node name, followed by attributes and then children which are themselves VRIR nodes.

VRIR is not meant for representing all parts of the program, such as functions dealing with complex data structures or various I/O devices. The parts of the program that cannot be compiled to VRIR are compiled by the containing compiler in the usual fashion, using the containing compiler's CPU-based code generator, and do not go through Velociraptor.

In the remainder of this chapter we present the important details of VRIR. The basic structure and constructs of VRIR are introduced in Section 1 and supported datatypes are introduced in Section 2. VRIR supports a rich array datatype which is flexible enough to model most common uses of arrays in multiple languages such as MATLAB or Python. Array indexing is introduced in Section 3 while standard array operators such as matrix multiplication are discussed in Section 4. In addition to serial constructs, VRIR supports parallel and GPU constructs in the same IR and these are discussed in Section 5. Finally, error handling and memory management are discussed in Section 6 and

```
def myfunc(a: PyInt32, b: PyInt32): PyInt32
        c = a+b #c is also 32-bit integer according to our type rules
        return c
```

Figure 1: Example of a Python function with type declaration

in Section 7 respectively.

In the next sections, we provide high-level grammar rules as an aid to understand the structure of some constructs. These grammar rules follow an EBNF based syntax, but we add names to each child to make it more readable. For example, a child named foo that invokes a grammar rule bar is written as "foo:bar".

1 Structure of VRIR programs and basic constructs in VRIR

1.1 Modules

```
module -> name:String exclude:Bool zeroIndex:Bool fns:
    function*
```

The top-level construct in VRIR is a *module*. Each module has zero or more functions as children. Functions in a module can call the standard library functions provided in VRIR and other functions within the same module. Each module has three attributes: a string attribute which is it's name, a binary attribute indicating if the indexing is 0-based or 1-based and a binary attribute indicating whether ranges (expressions such as 1 : 5) in the module include or exclude the stop value. In languages like Python, ranges exclude the stop value, while some languages like MATLAB include the stop value.

1.2 Functions and scoping rules

```
function -> name:String functype symtable body:statement+
symtable -> sym*
sym -> id:Int name:String type
```

Functions have a name, type signature, a list of arguments and a function body. The function body is a statement list. Similar to MATLAB or Python, VRIR supports multiple return values for functions. An example Python function with type declaration is shown in *Figure 1* and corresponding VRIR in S-expression form is shown in *Figure 2*.

Each function also has a symbol table where each symbol has a string name and a type. Inside the body of the function, only the integer IDs of symbols are referenced. VRIR has only three scopes: function scope, module scope and standard library scope. However, given that modules only consist of either functions defined within the module or declarations of external functions, and that

```
(function "myfunc"
  (fntype (int32vtype int32vtype)  (int32vtype))
  (symtable
    (0 "a" int32vtype)
    (1 "b" int32vtype)
    (2 "c" int32vtype))
  (args 0 1)
  (body
    (assignstmt
      (lhs
        (nameexpr 2 int32vtype))
      (rhs
        (plusexpr int32vtype
          (nameexpr 0 int32vtype)
          (nameexpr 1 int32vtype))))
    (returnstmt
      (exprs
        (nameexpr 2 int32vtype)))))
```

Figure 2: VRIR generated from example Python code

the standard library also consists solely of functions and no data variables, the name lookup semantics is considerably simplified. Generally, all VRIR expressions reference variables using symbol IDs defined in the function's symbol table. However, a few constructs involve calling functions, such as function calls and construction of function handles. In these cases the function name is stored as a string attribute in the AST. The function name is looked up first in the function scope (either a local function handle or a recursive call), then at module scope and finally at the standard library scope.

Function call semantics is similar to Java and Python, i.e. all variables are passed by value, where the value of non-scalar variables is the reference to the corresponding object.

1.3 Statements

VRIR supports many familiar statement types. Assignment statements have an RHS, which must be an expression, and one or more LHS targets. Assignment targets can be names, array subscripts or tuple indexing expressions. Statement lists are compound statements which contain other statements as children.

VRIR has a number of structured control-flow constructs. If-else statements have a condition expression, an if-body and an optional else-body. For-loops are modelled after for-loops over range expressions in Python where a loop index takes on values in a specified range. The range expression is evaluated before the execution of the for-loop and thus the number of iterations is fixed before the loop begins execution except if a break, return or exception is encountered. In

VRIR, we have defined a for-loop that iterate over multidimensional domains where the domain expression is evaluated before the loop begins execution. While-loops are more general loops and similar to while-loops in languages such as C or Python. Each while-loop has a test expression and a loop-body. If the condition is true, then the body is executed and the process is repeated until the condition becomes false. Both for-loops and while-loops may exit early if a break, continue or return statement is encountered or if an exception is thrown.

One of the main goals of VRIR is to provide constructs for parallel and GPU computing through high-level constructs and we describe the supported constructs in Section 5.

1.4 Expressions

Expression constructs provided include constants, name expressions, scalar and array arithmetic operators, comparison and logical operators, function calls (including standard math library calls), and array indexing operators. All expressions are typed. We have been especially careful in the design of VRIR for arrays and array indexing, in order to ensure that VRIR can faithfully represent arrays from a wide variety of source languages.

2 Supported datatypes

Knowing the type of variables and expressions is important for efficient code-generation, particularly for GPUs. Thus, we have made the design decision that all variables and expressions in VRIR are typed. It is the job of the containing compiler to generate the VRIR, and the appropriate types. Velociraptor is aimed at providing code generation and backend optimizations, and is typically called after a type checking or inference pass has already occurred. We have carefully chosen the datatypes in VRIR to be able to represent useful and interesting numerical computations.

2.1 Scalar types

VRIR has real and complex datatypes. Basic types include integer (32-bit and 64-bit), floating-point (32-bit and 64-bit) and boolean. For every basic scalar type, we also provide a corresponding complex scalar type. While most languages only provide floating-point complex variables, MATLAB has the concept of integer complex variables as well, and thus we provided complex types for each corresponding basic scalar type.

2.2 Array types

Array types consist of three parts: the scalar element-type, the number of dimensions and a layout. The layout must be one of the following three: row-major, column-major or strided-view. For example, a two-dimensional array of doubles in row-major format is written as *(arraytype float64vtype 2 row)* in VRIR s-expression syntax. The effect of layouts on address computations for array subscripts is described in Section 3.

We have made a design decision that the value of array sizes and strides are not part of the type system. Thus, while the number of dimensions and element type of a variable are fixed throughout the lifetime of the variable, it may be assigned to arrays of different sizes and shapes at different points in the program. This allows cases such as assigning a variable to an array of different size in different iterations of a loop, such as in some successive reduction algorithms. Further, requiring fixed constant or symbolic array sizes in the type system can also generate complexity. For example, determining the type of operations such as array slicing or array allocation, where the value of the operands determines the size of the resulting array, would be difficult if the array sizes are included in the type system. Thus, we chose to not include array sizes and strides in the type system for flexibility and simplicity.

The number of dimensions of an array variable is fixed in the type system because it allows efficient code-generation and optimizations. Our design decision implies that some functions, such as *sum*, are not fully expressible in the type system because the function can take an array of any dimensions as an argument and return a result that is an array with one less dimension. This weakness is somewhat mitigated by the fact that the compiler has special knowledge about standard library functions such as sum. Thus, many programs are still expressible in the language. An alternative may be to extend VRIR with support for symbolic number of dimensions in the future.

2.3 Domain types

Domain types represent multidimensional strided rectangular domains respectively. An n-dimensional domain contains n integer triplets specifying the start, stop and stride in each dimension. This is primarily useful for specifying iteration domains. A one-dimensional specialization of domains is called a range type, and is provided for convenience.

2.4 Tuple type

The tuple type is inspired from Python's tuple type and can also be used for one-dimensional cell arrays in MATLAB with some restrictions. A tuple is a composite type, with a fixed number of components of known types. Given that a tuple can contain elements of different types, allowing dynamic indexing of the tuple can lead to unpredictable types which goes against the design principles of VRIR. Thus, we enforce the restriction that a tuple can only be indexed using constant integers.

2.5 Void type

The Void type is similar to void type in C. Functions that do not return a value have the void type as return type.

2.6 Function types

Function types specify the type signature of a function. VRIR functions have a fixed number of arguments and outputs. The input arguments can be any type (except void). The return type can either be void, or one or more values each of any type excluding void or function types.

Some computations, such as function optimization routines, require passing functions as input. In languages such as C, this is done via function pointers while the same concept is called function handles in languages such as MATLAB. VRIR has support for function handles. Function handles are values of function type. Function handles can be stored in local variables, passed to functions or returned from functions.

2.7 Dynamically-typed languages and VRIR

Languages such as MATLAB and Python/NumPy are dynamically typed while we require that the types of all variables be known in VRIR. A good JIT compiler, such as McVM [?], will often perform type inference before code generation. For example, McVM dynamically specializes the code of a function at run-time, based on types of the actual parameters [?]. The McVM project shows that a type-specializing JIT compiler can infer the types of a large number of variables at runtime. Part of the reason of the success of McVM is that while MATLAB is dynamically typed, scientific programmers do not often use very dynamic techniques in the core numeric functions compared to applications written in more general purpose languages like JavaScript where dynamic techniques are more common. If a compiler is unable to infer types, users are often willing to add a few type hints for performance critical functions to their program such as in the Julia [?] language or the widely used Python compilation tool Cython [?]. Finally, if the containing compiler is unable to infer the types of variables, it can use its regular code generation backend for CPUs as a fallback, instead of using Velociraptor.

3 Array indexing

Array indexing semantics vary amongst different programming languages, thus VRIR provides a rich set of array indexing modes to support these various semantics. The indexing behavior depends upon three things: the layout (row-major, column-major or strided) of the array, the type of each index (integer, range or array) and the attributes specified for the expressions.

Consider a n-dimensional array A indexed using d indices. VRIR has the following indexing modes:

3.1 Array layouts

Row-major and column-major layouts are contiguous layouts, and we follow rules similar to C and $Fortran$ respectively. The strided-view layout is inspired from the `ndarray` datatype in Python's NumPy library. To explain the strided-view layout, consider an n-dimensional array A with base address $base(A)$. Then, the strided-view layout specifies that the n-dimensional index vector $(i_1, i_2, ..., i_n)$ is converted into the following address: $addr(A[i_1, i_2, ..., i_d]) = base(A) + \sum_{k=1}^{d} s_k * i_k$ (in a 0-based indexing scheme). The values s_k are called strides of the array.

3.2 Types of indices supported

3.2.1 Integer indices

If n integer indices are specified, then the address selected is calculated using rules of the layout of the array (row-major, column-major or stride-based).

3.2.2 Slicing arrays - Data sharing or copying

An array can be sliced by supplying a range instead of an integer as an index. Consider an array slicing operation such as $A[m : n]$. Array slicing operations return an array. Some languages like Python have array slicing operators that return a new view over the same data while other languages like MATLAB return a copy of the data. We support both possibilities in VRIR through a boolean attribute of array index nodes.

3.2.3 Using arrays as indices

Arrays can also be indexed using a single integer array (let it be named B) as index. B is interpreted as a set of indices into A corresponding to the integer values contained in B and the operator returns a new array with indexed elements copied into the new array.

3.3 Array indexing attributes

3.3.1 Negative indexing

This attribute is inspired from Python's negative indexing and affects both of the above cases. In languages like Java, if the size of the k-th dimension of the array is u_k, then the index in the dimension must be in the set $[0, u_k - 1]$. However, in NumPy, if an index i_k is less than zero, then the index is converted into the actual index $u_k + i_k$. For example, let the index i_k be equal to -1. Then, the index is converted to $u_k + i_k$, or $u_k - 1$, which is the last element in the dimension. We have a boolean attribute in the array index node to distinguish between these two cases.

```
%creates a 2x2 array
A = zeros(2);
% Results in 3x2 array [0 0; 0 0; 0 1]
A(3,2) = 1;
```

Figure 3: Example of array growth in MATLAB

3.3.2 Enabling/disabling index bounds checks

Array bounds-checks can be enabled/disabled for each individual indexing operation. This allows the containing compiler to pass information about array bounds-checks, obtained through compiler analysis, language semantics or programmer annotations, to Velociraptor.

3.3.3 Flattened indexing

If $d < n$, and all indices are integers, then different languages have different rules and therefore we have provided an attribute called *flattened indexing.* When flattened indexing is true, the behavior is similar to MATLAB where the last $n-d+1$ dimensions are treated as a single flattened dimension of size $\prod_{k=m}^{d} u_k$ where u_k is the size of the array in the k-th dimension. When flattened indexing is false, the behavior is similar to NumPy and the remaining $d-m$ indices are implicitly filled-in as ranges spanning the size of the corresponding dimension.

3.3.4 Zero or one-based indexing

A global variable, set by the containing compiler, controls whether one-based or zero-based indexing is used for the language.

3.3.5 Array growth

Consider an array-write at a particular index that is outside the current bounds of the array. In some languages, such as NumPy, an out-of-bounds exception is thrown. However, in MATLAB, the array is grown such that array-write is to a valid index. An example of MATLAB behavior is shown in *Figure 3.* We have defined an optional binary attribute for array growth that can be specified for each array subscript dimension separately. If not specified, then arrays are assumed to not grow.

4 Array operators

Array-based languages often support high-level operators that work on entire matrices. Thus, VRIR provides built-in element-wise operation on arrays (such as addition, multiplication, subtraction and division), matrix multiplication and matrix-vector multiplication operators. Unary functions provided include operators for sum and product reduction as well as transcendental and trigonometric functions operating element-wise on arrays.

5 Support for parallel and GPU programming

Programmers using languages such as MATLAB and Python are usually domain experts rather than experts in modern hardware architectures and would prefer high-level constructs to expose the hardware. VRIR is meant to be close to the source language. Therefore, we have focused on supporting high-level constructs to take advantage of parallel and hybrid hardware systems. Low-level, machine-specific details such as the number of CPU cores or the GPU memory hierarchy are not exposed in VRIR. The task of mapping VRIR to machine specific hardware is done by Velociraptor.

We support high-level parallel programming constructs that can be easily understood and used by scientific programmers. VRIR provides a lock-free parallelism model that is geared towards data parallelism but also allows some task parallelism. This is provided through a combination of parallel-for loops, atomic operations and parallel library functions. Heterogeneous programming is fully supported through accelerated sections. The constructs are as follows:

5.1 Parallel-for loops

Parallel-for loops are loops defined over a multi-dimensional domain where each iteration can be executed in parallel. A number of restrictions are placed on parallel loops in order to execute them safely in parallel. Parallel for-loops cannot contain break, continue or return statements because these do not make sense in a parallel context. Some restrictions are placed on errors such as out-of-bounds errors and these are detailed in Section 6. Parallel for-loops can contain other parallel-for loops, but the implementation converts any inner parallel-for loops into serial loops.

In parallel-for loops, variables are classified into thread-local variables and shared variables. Each iteration of the parallel-for loop has a private copy of thread-local variables while shared variables are shared across all the iterations. The list of shared variables of a parallel-for loop needs to be explicitly provided.

Shared variables cannot be assigned-to inside parallel-for loops but indexed writes to shared arrays is still permitted. An example showing legal use of shared variables in VRIR is shown in *Figure 4*. An example with illegal use of shared variables is shown in *Figure 5*.

In order to understand the reasoning behind preventing reassignment to shared array objects, we need to distinguish between *array objects* and *data arrays*. The arrays in languages such as C/C++ are just pieces of memory, which we can call as the *data array*, whereas arrays in languages such as MATLAB and Python are objects. The array object may have many member fields such as the pointer to the data array, an array of integers for sizes and potentially

```
a = zeros(10)
b = ones(10)
c = zeros(10)
#a,b,c are shared variables
for i in PAR(range(10),shared=[a,b,c]):
        #legal use of a,b,c. no reassignment, indexed-write only
        a[i] = b[i] + c[i]
```

Figure 4: An example of legal use of shared variables in parallel-for loop in pseudo-Python syntax

memory management related metadata. We want that the shared array objects should be immutable inside a parallel-for loop. However, the data array pointed to by the object can be mutable and can be read/written by multiple threads. An indexed write only modifies the data array, while a reassignment may change the array object itself.

Disallowing reassignment to shared variables prevents modification of the shared array object and simplifies reasoning for both the programmer and Velociraptor. If the shared array object is immutable, then that implies that the size and strides of the array remain the same throughout the execution of the parallel loop. This simplifies reasoning for the programmer. For example, if *Figure 5* was legal, then the size and contents of the array *b* will be completely unknown after the loop has finished because the iterations may execute in any order. This is generally not desirable for the programmer.

Allowing reassignment to shared arrays would have also caused issues for the compiler. Let us assume we allowed assignment to shared variables and one thread attempts to reassign a shared array. Reassignment operation may require that the thread modifies one or more shared array objects. While one thread is performing the reassignment, the other threads need to be prevented from reading or writing the shared array objects to ensure that the threads are not reading/write partially formed array objects. In such a situation, we may need to insert locks around each use or definition of shared array variables which will completely kill performance. Thus, disallowing reassignment or reallocation of shared variables simplifies the programming model for the programmer as well as the implementation.

5.2 Atomic operations

We also support atomic compare and swap inside parallel-for loops. The atomic compare-and-swap can be performed for 32-bit and 64-bit values. The target of the swap is a given memory location specified by an array and array index. Atomic compare-and-swap is a building block operation that allows the construction of higher level atomic operations.

```
a = zeros(10)
b = zeros(10)
#a,b are shared variables
for i in PAR(range(10),shared=[a,d]):
        #assignment to shared variable b is illegal
        b  = a[0:i]
```

Figure 5: An example of illegal use of shared variables in parallel-for loop in pseudo-Python syntax

5.3 Parallel library operators

VRIR has a number of implicitly parallel built-in operators. Velociraptor takes advantage of parallel libraries for these operators where possible. The built-in library operators that can be parallelized include matrix operators such as matrix multiply and transpose, element-wise binary operations such as matrix addition and element-wise unary operations such as trigonometric functions, Parallel library functions also provide support for reduce operations on arrays. We support four reduce operators: sum, product, min and max of an array either along a given axis or for all elements of the array.

5.4 Accelerated sections

Any statement list can be marked as an *accelerated section*, and it is a hint for Velociraptor that the code inside the section should be executed on a GPU, if present. Velociraptor and its GPU runtime (VRruntime) infer and manage all the required data transfers between the CPU and GPU automatically. At the end of the section, the values of all the variables are synchronized back to the CPU, though some optimizations are performed by Velociraptor and VRruntime to eliminate unneeded transfers.

In VRIR, accelerated sections can contain multiple statements. Thus, multiple loop-nests or array operators can be inside a single accelerated section. We made this decision because a larger accelerated section can enable the compiler or runtime to perform more optimizations compared to accelerated sections with a single loop-nest.

Parallel-for loops in accelerated sections are compiled to OpenCL kernels. In order to meet OpenCL restrictions, parallel-for loops in accelerated sections cannot have memory allocation or recursion.

We also disallow assignment of function handle variables inside sections. This restriction allows us to examine all possible function handles that may be called inside an accelerated region before the region begins executing in order to generate OpenCL code for the region.

6 Error reporting

We support array out-of-bounds errors in serial CPU code, parallel CPU loops and also in GPU kernels. Previous compilers have only usually supported out-of-bounds errors on CPUs and thus error handling is a distinguishing feature of our system.

In serial CPU code, errors act similar to exceptions and unwind the call stack. However, there is no ability to catch or handle such errors within VRIR code, and all exception handling (if any) must happen outside of VRIR code.

In parallel loops or in GPU sections, multiple errors may be thrown in parallel. Further, GPU computation APIs such as OpenCL have very limited support for exception handling, thus the design of VRIR had to find some way of handling errors in a reasonable fashion. We provide the guarantee that if there are array out-of-bounds errors in a parallel loop or GPU section, then one of the errors will be reported but we do not specify which one. Further, if one iteration of a parallel-loop, or one statement in a GPU section raises an error, then it may prevent the execution of other iterations of the parallel-for loop or other statements in the GPU section. These guarantees are not as strong as in the serial case, but these help the programmer in debugging while lowering the execution overhead and simplifying the compiler.

7 Memory management

Different containing compiler implementations have different memory management schemes. VRIR and Velociraptor provide support for two memory management schemes: reference counting and Boehm GC. The memory management scheme used in a given VRIR module is specified as an attribute of the VRIR module.

Dealing with reference counts is particularly tricky. We had earlier considered a scheme where we defined explicit reference increment and reference decrement statements in VRIR and containing compiler was required to explicitly insert these statements. This scheme has several drawbacks. Consider a statement such as $D = A * B + C$ where A, B, C and D are arrays. Here, the semantics dictates that a temporary array $t1$ will be created as follows: $t1 = A * B; D = t1 + C$. The reference count of $t1$ needs to be decremented once it is no longer needed. If we impose the requirement that the containing compiler should explicitly insert reference increment and decrement instructions in VRIR, then we also impose the burden of simplification of the complex array expressions before generating VRIR so that all temporary arrays are explicit. This increased burden goes against our goals of requiring minimal work from the containing compiler.

We have moved the burden of dealing with reference counting to Velociraptor. Velociraptor will automatically generate calls to the reference increment and decrement functions using rules inspired from CPython's reference counting mechanism. As discussed above, correct implementation of reference counting needs to carefully consider any temporary arrays in the computation. We have defined a simplification pass in Velociraptor that simplifies the IR. The simplification pass is done before code generation. During code generation, Velociraptor uses the following rules on the simplified VRIR to insert code for reference increment and decrement:

Assignment statements: First, the RHS is evaluated and if the result is an array, then the reference count of the result of the RHS is incremented. If the target of an assignment is an array, then the reference count of the object previously referenced by the target is decremented.

Function calls: In simplified VRIR, all parameters being passed in a function call are either name expressions or constants. If the function takes arrays as input arguments, then the reference count of the arrays being passed as parameters is incremented.

Return: In VRIR, a return statement may have zero or more return expressions. In simplified VRIR, the return expressions are all name expressions. The reference count of all array variables, except those occurring in return expressions, is decremented.

Implementation of memory management requires some glue code to be supplied by the containing compiler.