Velociraptor: A Compiler Toolkit for Array-Based Languages Targeting CPUs and GPUs

Rahul Garg
McGill University, Canada
rahul.garg@mail.mcgill.ca

Sameer Jagdale
McGill University, Canada
sameer.jagdale@mail.mcgill.ca

Laurie Hendren
McGill University, Canada
hendren@cs.mcgill.ca

Abstract

We present a toolkit called Velociraptor that can be used by compiler writers to quickly build compilers and other tools for array-based languages. Velociraptor operates on its own unique intermediate representation (IR) designed to support a variety of array-based languages. The toolkit also provides some novel analysis and transformations such as region detection and specialization, as well as a dynamic backend with CPU and GPU code generation. We discuss the components of the toolkit and also present case-studies illustrating the use of the toolkit.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Compilers

General Terms Languages, Performance

Keywords Compiler framework for Array-Based Language, MATLAB, Python

1. Introduction

Dynamic array-based languages have become extremely popular for scientific computing. While these languages are very productive, many implementations of these languages don't offer good performance. There has been a lot of interest in building compiler infrastructure for these languages.

In this paper, we describe a compiler toolkit called Velociraptor that can be used by compiler writers to build compilers and other tools for array-based languages. Velociraptor is an optimizing dynamic compilation framework that generates LLVM [10] for CPUs and OpenCL for GPUs respectively. We have described Velociraptor in previous publications [7, 8]. In this paper, our goal is to describe the software architecture and discuss how it may be used by the compiler community to build compiler infrastructure for array-based languages.

Velociraptor is not a standalone compiler. Instead it is a library that is designed to be integrated or embedded into an existing language implementation. We describe the system design of how a compiler may integrate Velociraptor in Section 2. Velociraptor takes as input it's own IR called VRIR. This IR is described in Section 3. Velociraptor is a modular framework and divided into separate components. We describe these components in Section 4. More details of how Velociraptor may be embedded into a compiler are discussed in Section 5. We discuss some case-studies of using the compiler toolkit for dynamic compilers for MATLAB [11] and Python in Section 6. We discuss some alternate use-cases, including a static backend for Velociraptor, and also offer some ideas of how the toolkit may be used or extended by the community in Section 7. We refer the reader to our webpage[1] for instructions on obtaining our toolkit, the case-studies, code samples and other documentation.

ARRAY'15, June 13, 2015, Portland, OR, USA
© 2015 ACM. 978-1-4503-3584-3/15/06...$15.00
http://dx.doi.org/10.1145/2774959.2774967

2. Overall Design

Consider a typical just-in-time (JIT) compiler for a dynamic language targeting CPUs shown in Figure 1. Such a compiler takes program source code as input, converts into its intermediate representation, does analysis (such as type inference), possibly does code transformations and finally generates CPU code.

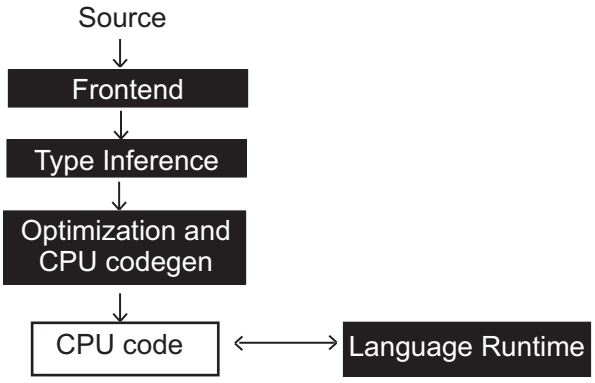

Figure 1: Possible design of a conventional compiler targeting CPUs.

Let us assume that the existing compiler is targeting serial CPU execution, and may not implement many optimizations. Now consider that a JIT compiler developer wants to extend this existing compiler to introduce optimizations such as bounds-check elimination, introduce parallelization for multi-cores and also efficiently target GPUs.

We have built the Velociraptor toolkit to simplify the building of compilers that want to efficiently compile numerical computations to CPUs and GPUs. The key idea is that Velociraptor is embedded inside the original compiler and takes over the duties of generating code for numerical sections of the program. Numerical sections of

[1] http://www.sable.mcgill.ca/mclab/projects/velociraptor

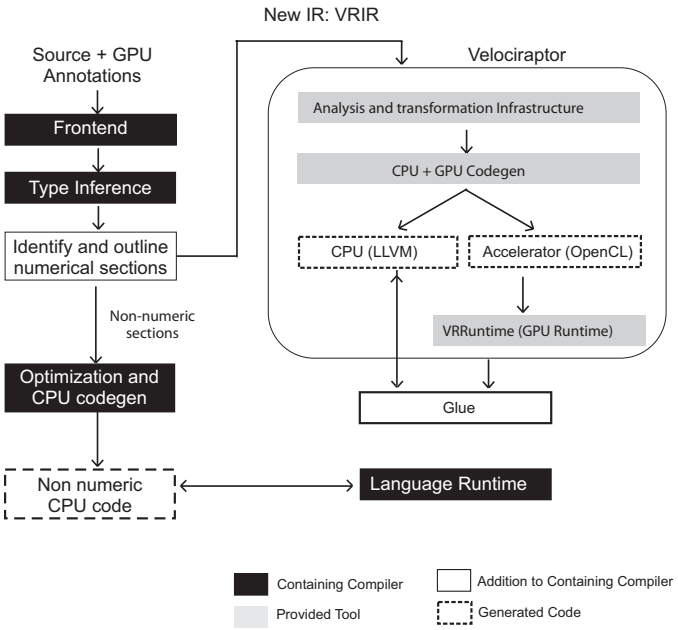

Figure 2: Extending the containing compiler through provided tools

the program may include numerical computations that execute on the CPU, as well as numerical computations that are annotated to be executed on GPUs.

We call the compiler that embeds our tools as the *containing compiler*. *Figure 2* demonstrates how our approach cleanly integrates with a containing compiler. The figure shows the original parts of the containing compiler in dark. The toolkit described in this paper is shown in shaded box on the right. The components of the toolkit are discussed in Section 4.

The compiler writer who is integrating our tools into the containing compiler needs to add two new components to the containing compiler, and these are shown in white boxes. These two components can be understood in terms of our design goals.

The first design goal was to ensure that Velociraptor is not tied to a single source language. To achieve this goal, we introduced a new flexible IR called VRIR. Velociraptor takes VRIR as input which ensures that it is not tied to a single source language. The compiler writer integrating Velociraptor needs to add new passes to the containing compiler to generate VRIR from the numerical sections of the program.

Our second design goal was to think of Velociraptor as a component used to extend or enhance a compiler toolchain and not on rewriting from scratch or providing an alternate virtual machine environment. We wanted to ensure that the internal data structures of the existing implementation typically exposed in the C/C++ API are preserved as far as possible. For example, many libraries depend upon CPython's C API and therefore a dynamic compiler integrating with CPython should not require a change in the data-structures exposed through this API. We have achieved this design goal by introducing glue code which abstracts away the internals of the language runtime from Velociraptor.

The components that need to be added to the containing compiler are described in more detail in Section 5. We illustrate how to use Velociraptor by describing two-case studies (McVM [3] and PyRaptor) about integrating Velociraptor into a containing compiler in Section 6.

3. VRIR

A key design decision in our approach is that the numeric computations should be separated from the rest of the program, and that our IR should concentrate on those numerical computations. We have designed VRIR to meet these goals. VRIR is a high-level, procedural, typed, abstract syntax tree (AST) based program representation. The distinguishing feature of VRIR is that it contains a rich array datatype that makes it easy to generate VRIR from array-based languages. VRIR also has support for high-level parallel and GPU programming constructs.

Structurally, each AST node in VRIR may have certain named attributes and may have children. Named attributes are essentially name/value pairs where the value can be one of four types: integers (such as integer id of a symbol), boolean, floating point (such as value of a floating-point constant occurring 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. An example is shown in *Figure 4*.

```
def myfunc(a: PyInt32, b: PyInt32): PyInt32
    c = a+b
    return c
```

Figure 3: Example of a Python function with type declaration

```
(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 4: VRIR generated from example Python code

In this paper, we focus on a high-level description due to space constraints. A detailed grammar of textual representation of VRIR, written using notation used by ANTLRv3 parser generator, is available on our website. More samples of VRIR generated by our tools from some sample programs are also available. VRIR has also been described in a previous publication [8].

3.1 Overview

VRIR is a procedural IR. The top-level component in VRIR is a module, which consists of one or more functions. Similar to familiar programming languages such as C or Python, functions contain a sequence of statements. We support usual structured programming constructs such as assignment statements, if/else condition-

als, while-loops, for-loops, break and continue statements and return statements. VRIR type system consists of scalar types such as floating-point datatypes, array datatypes, function types and a unit type similar to void type in C. Array datatypes consist of an element-type, number of dimensions and the layout (row-major, column-major or strided).

3.2 Array Operators

VRIR has a rich and flexible array datatype. The array data-type supports multiple types of layouts (row-major, column-major and strided). Array-based languages often have diverse array indexing and slicing schemes. Therefore, we support various indexing and slicing modes such as zero and one-based indexing and also whether or not to use negative indexing similar to Python's negative indexing. Such properties are usually specified as node attributes of suitable indexing nodes. In addition, we also allow specifying whether or not to enable bounds-checks on a particular array subscript. This allows the containing compiler to convey any knowledge it may have about the array subscripts to Velociraptor. Finally, we also support many array operators such as array addition and matrix multiplication. Overall, the rich support of array constructs in VRIR makes it easy to generate from array-based languages.

3.3 Parallelism for CPUs and GPUs

VRIR supports parallel-for loops, which are loops defined over a multi-dimensional domain where each iteration can be executed in parallel. 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 by the containing compiler.

Any statement list can be marked as an *GPU 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.

4. Toolkit Components

Velociraptor consists of a code-generation component that takes VRIR as input and generates CPU and GPU code. The code generation component is further sub-divided into two separate components. The first component is a target-independent analysis and transformation infrastructure called libVRIR described in Section 4.1. VRIR and analysis information generated from this component is then fed into our dynamic backend VRdino described in Section 4.2 which does further optimization and code-generation. The code generation component is complemented by a smart GPU runtime called VRruntime and it is described in Section 4.3.

4.1 Target Independent Analysis and Optimizations: libVRIR

The first compiler component is a library called libVRIR that includes several analysis and transformation passes. libVRIR is not tied to any particular backend and can be used to build both static and dynamic tools. libVRIR is written in C++ and it only depends on ANTLRv3 C runtime. The minimal library dependence ensures that libVRIR is very easy to build and is typically the first component that a compiler writer integrating Velociraptor will interact with or explore. We also provide a simple utility called *vrirfront* built using libVRIR that takes as input VRIR textual representation and attempts sanity checking such as parsing and simplification. This tool can be used by compiler writers during development to verify that the VRIR generated by their tools can pass basic testing.

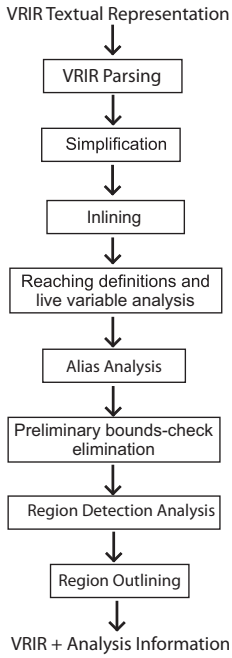

Figure 5: libVRIR analysis and transformation library for VRIR

The passes included in libVRIR are shown in *Figure 5*. The first phase is a parser for VRIR textual representation that generates VRIR AST data-structure. This is followed by a simplification pass that breaks down complex expressions such as complex array expressions. One of the goals of the simplification pass is to ensure that expressions that may allocate arrays, such as array addition, are top-level expressions in statement and not sub-expressions. This simplifies subsequent analysis phases. Simplification pass is a mandatory pass and should be called by any tool using libVRIR. Simplification is followed by several optional analysis passes such as live variable analysis.

We also have a preliminary bounds-check elimination pass that implements two approaches of eliminating bounds-checks. The first approach is based on the observation that many loops often have known integer constant (such as 0 or 1) as loop lower bound, and the loop step is a positive known constant. Consider an array reference $A[i]$, where A is a one-dimensional array and i is the loop index. In such cases, depending on the indexing scheme (0 or 1 based) and the values of the loop lower bound and loop step, the lower bound check as well as the negative indexing check may be eliminated. The second approach is based on elimination of redundant checks.

libVRIR implements a region detection analysis, which we described in a previous publication [7]. The idea is to identify regions which the compiler may be able to optimize or specialize using information about values or shapes of certain variables at runtime. However, given the cost of runtime specialization, such regions should contain potentially expensive operations such as loops or array library operations. Further, these regions should be analyzable and self-contained. We call such regions as potentially interesting regions (PIRs). libVRIR provides an optional pass to identify such PIRs, and also another optional pass to outline these regions into separate functions.

4.2 VRdino: Dynamic Backend

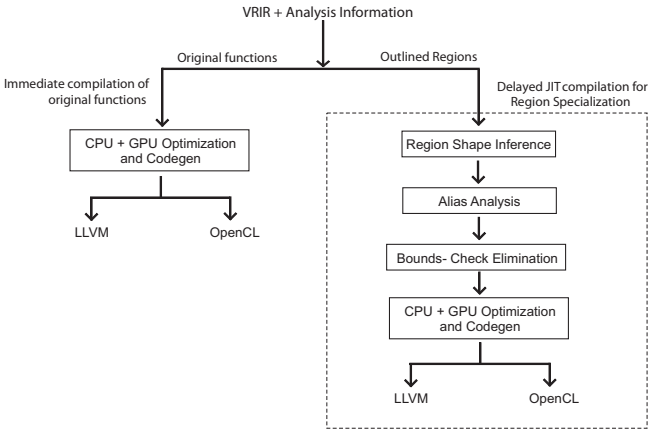

Figure 6: VRdino dynamic backend for Velociraptor

The next component is a dynamic code generation backend called VRdino that takes VRIR output from libVRIR and generates LLVM and OpenCL for CPUs and GPUs respectively. VRdino performs runtime specialization of regions identified using region detection analysis in libVRIR. The original functions are immediately compiled to LLVM and OpenCL. However, the identified regions are not immediately compiled. Instead, they are replaced with stub functions. When program execution reaches the stub function, the stub function invokes VRdino. VRdino examines the actual value of the function parameters and based on this information, performs shape inference and alias analysis. VRdino then performs bounds-check elimination based upon shape inference information. Finally, these regions are compiled to CPU and GPU code. Thus, region specialization allows VRdino to perform sophisticated optimizations at runtime.

VRdino depends upon libVRIR, LLVM, OpenCL driver, OpenBLAS and RaijinCL. RaijinCL is a GPU matrix operations library that we have described in a previous publication [6].

4.3 VRruntime: Dynamic Runtime for VRdino

VRruntime efficiently manages GPU memory, CPU-GPU synchronization and task dispatch. VRruntime includes many optimizations such as avoiding redundant transfers, using all available GPU execution queues and asynchronous dispatch of work to the GPU. VRruntime is tightly integrated with VRdino and is built as part of the build process of VRdino. VRruntime is completely transparent to the containing compiler, and the containing compiler does not need to do any work to use VRruntime.

5. Embedding Velociraptor

As we discussed previously, integration Velociraptor into a containing compiler requires integrating Velociraptor into the code generation infrastructure and the language runtime. These are described in Section 5.1 and Section 5.2 respectively.

5.1 Integrating Velociraptor into Code Generation

Integrating Velociraptor into the code generation infrastructure requires inserting new passes in the containing compiler. These new passes determine the numerical sections in the program and compile these to VRIR. VRIR is a typed IR and therefore these new passes will be called after analysis such as type inference performed by many JIT compilers for dynamic languages.

Velociraptor compiles VRIR to CPU and GPU code and returns a function pointer corresponding to the compiled version of each VRIR function. The containing compiler replaces calls to the outlined function in the original code with calls to the function pointer returned by Velociraptor. Non-numerical parts of the program, such as dealing with file IO, string operations and non-array data structures are handled by the containing compiler in its normal fashion.

5.2 Exposing Language Runtime to Velociraptor through Glue Code

Apart from integrating Velociraptor into the containing compiler's code generation, the compiler writer integrating Velociraptor into the containing compiler also needs to provide a small amount of glue code to expose the language runtime to Velociraptor. Velociraptor is not a virtual-machine by itself and instead integrates upon the existing language runtime through the glue code. The glue code has several components: exposing the array object representation to Velociraptor, and exposing memory management constructs to Velociraptor.

Array data-structures in languages such as MATLAB and in Python/NumPy are more than just plain pointers to data. We will refer to these array data-structures as array objects. In addition to the pointer to the data, array objects might contain information such as size (and potentially the layout) of the array as well as metadata such as reference counts for memory management. Different language runtimes will use different structures for representing array objects. However, Velociraptor needs to know the structure of the array objects in order to generate code as well as to implement the standard library functions such as matrix multiply.

We take the following approach. Velociraptor source code depends upon a header file $vrbinding.hpp$, but we do not supply this header file. Instead, the header file is supplied by the compiler writer integrating Velociraptor into the containing compiler. A template of the required header file is provided by Velociraptor and needs to be filled-in by the compiler writer. Through this header file, the compiler writer provides typedefs from the language runtime's array objects to type-names used in Velociraptor. The compiler writer also provides the implementation of macros and functions to access the fields of the array object structure. In Velociraptor implementation, accessing the fields of array object is only done via these macros and functions, whose names are provided in the template. The body of any required functions can be supplied in a separate source file. The compiler writer also provides a representation of the array object structure in LLVM. A simple way of generating the LLVM representation is to run the C or C++ API header files of the language runtime through the clang tool and asking it to generate LLVM IR. The resulting LLVM IR file can be examined and the relevant structure definitions in LLVM can be extracted.

Finally, we discuss memory management. Different language implementations may have different memory management schemes. For example, some implementations may depend upon reference counting while some others may depend upon conservative techniques such as the Boehm GC. Typically, array object allocation routines will require custom logic that depends upon the memory management scheme. Thus, Velociraptor does not provide any object allocation routines by itself. Instead, we define abstract APIs for object allocation and the language implementation needs to supply the implementation. Similarly, for language runtimes that use reference counting, we define abstract APIs for reference count (ref-count) increment and ref-count decrement in Velociraptor which then need to be implemented by the compiler writer who is integrating Velociraptor.

6. Case-Studies of Using the Toolkit

To illustrate how to use Velociraptor, we provide two case studies: (1) a prototype JIT compiler for Python, and (2) an extension of McVM [3] for MATLAB. In both of these case-studies, we offer support for targeting multi-core CPUs and GPUs through language extensions exposed to the programmer. We provide parallel for-loop constructs and also GPU sections whose semantics matches that of VRIR discussed in Section 3. Both of these dynamic compilers target multi-core CPUs and GPUs by integrating Velociraptor, including libVRIR, VRdino and VRruntime.

However, these case-studies have many important differences including the source language accepted, the way the numerical sections are identified, the places where Velociraptor is used and the glue code. We discuss PyRaptor in Section 6.1 and McVM extension in Section 6.2 respectively and summarize performance and development experience in Section 6.3 and Section 6.4 respectively.

6.1 Python Integration: PyRaptor

We have written a proof-of-concept compiler called PyRaptor for a numeric subset of Python, including the NumPy library, that integrates with CPython, the standard Python [12] interpreter. PyRaptor is implemented as an *extension module* for CPython version 3.2 and above, and does not require the user to modify the installed CPython or NumPy. An extension module is a dynamically loadable library (DLL), written in a language such as C/C++ using the CPython C API such that CPython knows how to load and interact with the library.

6.1.1 Identification of Numerical Sections and Code Generation

The idea is that the programmer identifies potentially expensive numeric computations, outlines them into functions and calls PyRaptor to compile these functions. Rest of the program is still interpreted by the CPython interpreter. Thus, in this prototype, we ask the programmer to identify and outline numeric sections.

PyRaptor is a two-phase compiler. The first phase is type inference and second phase is code generation. We ask the programmer to add type signatures to each function to be compiled. This signature is added using the function meta-data syntax introduced in Python 3. In the type inference phase, PyRaptor first calls a Python standard library module called *ast* which provides functionality to construct an AST from Python source code. Using the type signature of the function, PyRaptor then infers the type of the local variables by a simple forward analysis. We require that the type of a variable not change within a function. PyRaptor can handle scalar numerical datatypes, NumPy arrays and tuples.

In the second phase, PyRaptor compiles annotated functions to VRIR and then calls Velociraptor to generate CPU and GPU code. PyRaptor relies on Velociraptor for all code generation including serial CPU code, parallel CPU code and GPU code.

6.1.2 Glue Code

Python's NumPy library has a class called *ndarray* that represents an n-dimensional array. This class is implemented as a C struct. We provided implementations of Velociraptor's abstract APIs for accessing the fields of the NumPy ndarray objects. We also needed to provide the LLVM implementation of the C structure so that VRdino can understand and manipulate the structure. This was done by running clang on the relevant NumPy C header file and then copying the LLVM IR generated by Clang.

Next, we needed to tackle memory management. Velociraptor declares but does not define the API functions required for allocation of array objects and we provided the implementation of these functions for NumPy in the glue code. CPython uses reference-counting for memory management and we provided glue code to expose the reference counting mechanism of the interpreter to VRdino.

Finally, we provided a dispatcher function which acts as a bridge between the CPython interpreter and the functions compiled by Velociraptor. Once a Python function is compiled, PyRaptor replaces the original Python function with a dispatcher function. The dispatcher takes as arguments the arguments of the Python function it is replacing, as well as the function ID of the generated function. The dispatcher unboxes the CPython objects received as arguments of the function into their C counterparts and calls the compiled function. Similarly, the dispatcher boxes the result of the function call into the CPython objects and returns them to the interpreter.

6.2 Extending McVM: A Dynamic JIT Compiler for MATLAB

McVM is a virtual machine for the MATLAB language. McVM includes a type specializing just-in-time compiler and performs some optimizations such as bounds-check elimination. Prior to this work, McVM generated LLVM code for CPUs only.

6.2.1 Identifying Numerical Sections

Unlike PyRaptor, we do not need type declarations because McVM automatically infers the type of the functions. Further, unlike PyRaptor, the programmer does not need to identify numerical sections. We have implemented a number of simple heuristics to automatically identify sections of code to be compiled to VRIR and handed over to Velociraptor. These heuristics are called after McVM has finished type inference. The first heuristic is to identify functions that can be completely converted to VRIR. If the body of a function satisfies all the constraints given above, then the function is compiled to VRIR. If the first heuristic is not applicable to a given function, then McVM looks for parfor-loops and GPU sections in the code. If the body of the parfor-loop or the GPU section satisfies the constraints above, then these are outlined into a separate function and compiled to VRIR. Otherwise, we remove the GPU annotations and also convert any parallel-loop to serial for-loops in any code that cannot be compiled to VRIR. The final rule is that if there is a loop that calls a function foo, and if foo will be compiled VRIR, and if the body of the loop satisfies the typing constraints above, then the loop is outlined and compiled to VRIR. It is important to capture such loops and compile to VRIR, because it allows us to communicate to Velociraptor and it's specialization infrastructure that the function foo may be called inside a loop.

6.2.2 Glue Code

McVM has a template class for representing arrays, and it defines a number of concrete classes as template specializations for various element datatypes. We exposed these classes to Velociraptor both by providing macros for field access as well as providing a hand-written LLVM representation of these array classes. For memory management, McVM relies on Boehm GC for memory management. We ensured that our implementation of Velociraptor's array allocation functions calls the required GC functions. No other work was required to integrate with the Boehm GC.

6.3 Performance

Due to space constraints, we decided to focus more on the description of our tools and only give a high-level overview of performance results. We have compared the performance of Velociraptor generated code on a variety of MATLAB and Python benchmarks against state-of-the-art compilers. For MATLAB, we compared the performance against MathWorks MATLAB and McVM, both of which are JIT compiled. For Python, we compared against Cython [13] static compiler which generates C++ from an annotated dialect of Python.

The benchmarks used in our study will be made available on our website and contain a diverse set of benchmarks used by previous researchers as well as ports of some of the parallel computing dwarves [1] to Python. We found that the serial code generated by Velociraptor either outperforms or is generally competitive with these compilers. For example, on one machine we found that Velociraptor had a geometric mean speedup of 1.76 and 1.09 over Cython and McVM respectively. Turning to parallelism, parallel CPU code generated by Velociraptor was found to further improve performance. We found a speedup of 3.5 and 2.13 over the serial code generated by Velociraptor on two different machines with a quad-core Intel and dual-module AMD machine respectively. Finally, we tested the GPU code generation on GPUs from two vendors and found that in some problems the GPU code generated by Velociraptor offered a speedup of upto 3 over the parallel CPU code.

The key takeaway is that the user of our toolkit can build compilers that have state-of-the-art performance for serial code and provide further speedups by easily targeting multi-cores and GPUs.

6.4 Development Experience

Overall, we found that using Velociraptor enabled some saving in the development time of both compilers. As an indirect estimate, the number of lines of code in PyRaptor, and the number of lines of code required to extend McVM, were both less than about 25% of the number of lines in Velociraptor. This lends some credence to the hypothesis that the effort of integrating Velociraptor is far less than the amount of functionality and code implemented in our tools.

7. Alternate Use-Cases

Due to the modular design of our toolkit, parts of the toolkit can potentially be used in tools other than dynamic compilers. In particular, libVRIR can be reused in both static and dynamic tools. We describe two such usages here.

7.1 VeloCty: A Static Backend for Velociraptor

Although Velociraptor was designed with JIT compilers in mind, we also experimented with using it in the static compiler context. Velo**C**ty [9] is a static compiler toolkit, built using Velociraptor, which generates C++ code from MATLAB or Python functions which were annotated as important by the user.

Velo**C**ty uses the VRIR from Velociraptor as a common IR from which C++ code is generated. We generate VRIR for MATLAB using the McLAB [2, 4, 5] toolkit, which also provides static type and shape inference.We reuse the type inference and VRIR generator from PyRaptor for Python.

Along with the generated code, a wrapper function is also generated, in the source language of MATLAB or Python, to interface with the generated code. There are two optimizations that are performed on the generated code. The first optimization is the elimination of array bounds checks that are performed inside loops. We use the loop-info collector provided by Velociraptor to implement this optimization. This optimization supplements the preliminary bounds check optimization of Velociraptor. The second optimization is the elimination of redundant memory allocations during array operations. Additionally, we also support naive parallelism using OpenMP pragmas based on annotations for shared variables provided by the user.

The Velo**C**ty project shows that the VRIR and front-end support of Velociraptor can be used in static compiler contexts.

7.2 Research Idea: Tooling Using Region Detection Analysis

Function inlining, region detection analysis and region outlining available in libVRIR can do some interesting refactoring of the code such that potentially heavy computations in the code are easy to identify. While in Velociraptor, we use the refactoring to identify functions which should be dynamically specialized, such automatic refactorings may also be useful for static compilers, particularly for profile-guided tools. For example, a profile guided tool may simply instrument the source code so that only the shapes of critical shape variables and value of critical value variables is recorded at the entry point of the outlined function. This instrumentation can be much lower overhead than profiling the shape at multiple points inside the region body.

8. Conclusion

Velociraptor provides a collection of tools that may be utilized by compiler writers to build compilers and other tools for array-based languages. Velociraptor is modular and divided into libVRIR and VRdino and associated VRruntime. libVRIR provides a reusable analysis and transformation infrastructure that can be utilized by both static and dynamic tools. VRdino provides a dynamic backend that can be used by compiler writers to build or extend dynamic backends for various array-based languages. In addition to discussing the components, we also discussed several case studies of using tools provided by Velociraptor. We hope that we have motivated you, the reader, to try out our tools in your projects.

References

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] A. Casey, J. Li, J. Doherty, M. Chevalier-Boisvert, T. Aslam, A. Dubrau, N. Lameed, A. Aslam, R. Garg, S. Radpour, O. S. Belanger, L. J. Hendren, and C. Verbrugge. McLab: An extensible compiler toolkit for MATLAB and related languages. In *C3S2E'10*, pages 114–117, 2010.

[3] M. Chevalier-Boisvert, L. Hendren, and C. Verbrugge. Optimizing MATLAB through just-in-time specialization. In *CC 2010*, pages 46–65, 2010.

[4] J. Doherty and L. Hendren. McSAF: a static analysis framework for MATLAB. In *proceedings of ECOOP 2012*, pages 132–155, 2012.

[5] A. Dubrau and L. Hendren. Taming MATLAB. In *proceedings of OOPSLA 2012*, pages 503–522, 2012.

[6] R. Garg and L. Hendren. A portable and high-performance general matrix-multiply (GEMM) library for GPUs and single-chip CPU/GPU systems. In *Proceedings of PDP 2014*, pages 672–680, 2014.

[7] R. Garg and L. Hendren. Just-in-time shape inference for array-based languages. In *ARRAY'14*, 2014.

[8] R. Garg and L. Hendren. Velociraptor: an embedded compiler toolkit for numerical programs targeting CPUs and GPUs. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques*, PACT '14, 2014.

[9] S. Jagdale. Velocty : A static optimising compiler for MATLAB and NumPy. Master's thesis, McGill University, April 2015.

[10] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO 2004*, pages 75–86, 2004.

[11] MathWorks. MATLAB: The Language of Technical Computing. `http://www.mathworks.com/products/matlab/`.

[12] Python.org. Python Programming Language: Official Website. `http://python.org`.

[13] D. S. Seljebotn. Fast numerical computations with Cython. In G. Varoquaux, S. van der Walt, and J. Millman, editors, *Proceedings of the 8th Python in Science Conference*, pages 15 – 22, Pasadena, CA USA, 2009.