Velociraptor: An Embedded Compiler Toolkit for Numerical Programs Targeting CPUs and GPUs

Rahul Garg
School of Computer Science
McGill University
Montreal, Canada
rahul.garg@mail.mcgill.ca

Laurie Hendren
School of Computer Science
McGill University
Montreal, Canada
hendren@cs.mcgill.ca

ABSTRACT

Developing just-in-time (JIT) compilers that that allow scientific programmers to efficiently target both CPUs and GPUs is of increasing interest. However building such compilers requires considerable effort. We present a reusable and embeddable compiler toolkit called Velociraptor that can be used to easily build compilers for numerical programs targeting multicores and GPUs.

Velociraptor provides a new high-level IR called VRIR which has been specifically designed for numeric computations, with rich support for arrays, plus support for high-level parallel and GPU constructs. A compiler developer uses Velociraptor by generating VRIR for key parts of an input program. Velociraptor provides an optimizing compiler toolkit for generating CPU and GPU code and also provides a smart runtime system to manage the GPU.

To demonstrate Velociraptor in action, we present two proof-of-concept case studies: a GPU extension for a JIT implementation of MATLAB language, and a JIT compiler for Python targeting CPUs and GPUs.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Compilers

Keywords

GPU hybrid systems; Compiler framework for Array-Based Language; MATLAB; Python

1. INTRODUCTION

Array-based languages such as MATLAB [13], Python (particularly NumPy [22]) and R [19] have become extremely popular for scientific and technical computing. However, many implementations of such languages were not designed for performance.

Better performance can be achieved in two ways. First, some implementations such as CPython are not even JIT-compiled. In this case, a JIT compiler targeting serial CPU execution needs to be retrofitted to the language implementation. If there is an existing JIT compiler, further performance enhancement will require a good analysis and transformation infrastructure. Second, languages and compilers can be evolved to target modern hardware such as multi-core CPUs and highly parallel co-processors such as GPUs and Xeon Phi. In this case, new compilation infrastructures targeting multi-core CPUs and GPUs need to be written. A GPU runtime will also be required to manage GPU task dispatch, as well as GPU memory and data transfers between CPU and GPU.

Building JIT compilers for numerical programs targeting CPUs and GPUs can require substantial effort. Our goal in this work is to simplify the work of building dynamic compilers for numerical programs targeting CPUs and GPUs. Our suggested solution to simplify the problem of building such compilers is to provide an embeddable, reusable compiler toolkit specialized for compiling numerical array-based programs to CPUs and GPUs. Once the toolkit is built, it can be reused in multiple projects.

We have designed and implemented such a toolkit called Velociraptor that demonstrates the feasibility of this approach. Velociraptor consists of two pieces. The first piece is an optimizing dynamic compiler infrastructure that generates LLVM and OpenCL for CPUs and GPUs respectively. The second piece is a smart and portable GPU runtime, VRRuntime, built on top of OpenCL. Velociraptor is designed to be embedded into a larger compiler, which we call the *containing compiler*. We also provide two case studies of integrating Velociraptor to demonstrate the utility of the approach.

We first examine some criteria that led to our design and then conclude this section with our contributions.

1.1 Design goals

Focus on numerical programs and arrays

We focus on numerical parts of the program. Numerical computations, particularly those utilizing arrays, are the performance critical part of many scientific programs. Numerical computations are also the most likely candidates to benefit from GPUs which are an important motivation behind this work. Array-based languages such as MATLAB and Python expose versatile array datatypes with high-level array and matrix arithmetic operators and flexible indexing schemes. Any proposed infrastructure or solution needs to have an intermediate representation (IR) with a rich array datatype.

PACT'14, August 24–27, 2014, Edmonton, AB, Canada.

ACM 978-1-4503-2809-8/14/08 ...$15.00.
http://dx.doi.org/10.1145/2628071.2628097.

IR support for parallelism and GPUs

We are focusing on languages such as MATLAB and Python, which are popular because they are productive. A typical user of such languages will usually prefer high-level solutions to targeting modern hybrid (CPU/GPU) hardware One example is that a programmer, or an automated tool, might simply specify that a particular computation or loop should be executed in parallel or that a section of code should be executed on the GPU. Such high-level constructs need to be represented in the intermediate representation (IR) of any proposed infrastructure. Compiling these high-level constructs to either parallel CPU code or to GPU kernels, as required by the construct, should be the responsibility of the proposed solution. In addition to code generation, any proposed solution will also need to automatically insert required runtime calls to data transfers, GPU task dispatch and synchronization.

Reusability across languages

Currently, every just-in-time (JIT) compiler project targeting hybrid systems is forced to write their infrastructure from scratch. For example, MATLAB compiler researchers don't benefit from the code written for Python compilers. However, the types of compiler and runtime facilities required for efficient implementation of numerical computations in various array-based languages is qualitatively very similar. Ideally, the same compiler codebase can be reused across multiple language implementations. A shared and reusable infrastructure that is not tied to a single programming language, and that everyone can contribute to, will simplify the work of compiler writers and also encourage sharing of code and algorithms in the community.

Integration with existing language runtimes

Existing implementations face the challenge of maintaining compatibility with existing libraries. For example, many Python libraries have some parts written in C/C++ and depend on the CPython's C/C++ extension API. In such cases, a gradual evolution may be more pragmatic than a complete rewrite of the language implementation. For example, a compiler for numerical subset of Python can be written as an add-on module to the Python interpreter instead of writing a brand-new Python implementation. Tools such as Numba [15] and Cython [23], which have become popular in the Python community, illustrate this approach.

Thus, our design criterion is that our solution needs to integrate with existing internal data structures of each language runtime, particularly the data structure representing arrays in the language. This design criterion immediately rules out certain designs. For example, one may consider implementing a standalone virtual-machine that exposes a bytecode specialized for numerical computations and then simply compile each language to this VM. However, a standalone VM will have its own internal data-structures that will be difficult to interface with existing language runtimes especially if reusability across multiple language implementations is desired. We have chosen an embeddable design with well-defined integration points so that our compiler framework can be easily integrated into existing toolchains while maintaining compatibility.

1.2 Contributions

System design

The first contribution was to define the overall system design. We proposed the design criteria of our work in Section 1.1. The details of our proposed system design are given in Section 2. The system design involved defining the distribution of work and interfacing between Velociraptor and the compiler into which Velociraptor is being embedded such that the compiler writer integrating Velociraptor into a containing compiler needs to do minimal work.

Domain-specific and flexible IR design

The second contribution was defining a domain-specific IR for Velociraptor specialized for numeric computing. As discussed in the design challenges, this IR needs to be flexible enough to model the semantics of multiple numerical programming languages. In particular, the array datatype provided in the IR needs to be rich enough to model various indexing schemes and array operators found in multiple programming languages such as MATLAB and Python/NumPy. Further, this IR needs to be simple to generate from a containing compiler to facilitate easy integration of Velociraptor. Finally, in addition to serial constructs, the IR needs high-level constructs to be able to express parallelism as well as expressing the parts of the program to be offloaded to a GPU. We address these challenges by defining a novel domain-specific IR called VRIR which is discussed in Section 3.

Compiler and runtime contributions

We have implemented an optimizing, multi-pass compiler toolchain that performs many analysis and transformations and produces LLVM for CPUs and OpenCL for GPUs. Our compiler infrastructure is described in Section 4. We also describe a novel memory reuse optimization for library functions such as array addition.

The compiler toolchain needs to be complemented by a GPU runtime. We have built VRRuntime, a GPU runtime that provides a high-level task-graph based API to the code generator. The runtime automatically handles all data transfers and task dispatch to the GPU. VRRuntime is implemented on top of OpenCL and is described in Section 5. VRRuntime provides asynchronous dispatch, automatic CPU-GPU data transfer management and GPU memory management. While similar techniques have been implemented in other runtimes, the implementation in the context of array-based programming languages required solving several subtle issues and is a new contribution.

Demonstrating the utility of Velociraptor

To demonstrate the reusability of Velociraptor, we have used it in two projects described in Section 6. The first case study is an extension of McVM [6], a just-in-time compiler and interpreter for MATLAB language. In this project, we embedded Velociraptor in McVM to enable support for GPUs, while CPU code generation is done by McVM. The second project is a JIT compiler designed as an add-on to the CPython interpreter. It takes as input annotated Python code and generates CPU and GPU Python code using Velociraptor. This demonstrates how a compiler could leverage Velociraptor to do both the CPU and GPU code generation.

We briefly describe our experience in integrating Velociraptor into these two toolchains and then report benchmark results from both these case studies.

2. SYSTEM DESIGN

The first challenge was to specify the services that should be provided by an embedded toolkit and how it would integrate into an existing compilation infrastructure.

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. Now consider that a JIT compiler developer wants to extend his/her compiler to target hybrid systems by embedding Velociraptor into the compiler. We call the resulting extended compiler as a *containing compiler* while Velociraptor is called the *embedded compiler*.

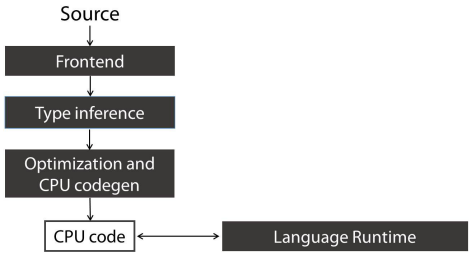

Figure 1: Possible design of a conventional compiler targeting CPUs. You want to extend this compiler to support multicores and GPUs

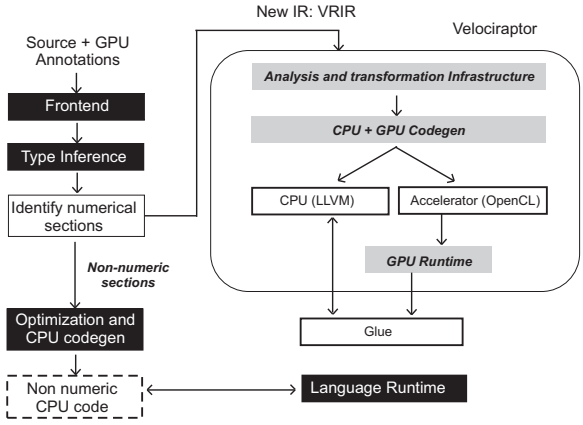

Figure 2: Containing compiler embedding Velociraptor, and showing the parts provided by Velociraptor

We have built the Velociraptor toolkit to simplify the building of containing compilers that want to compile numerical computations to CPUs and GPUs. The key idea is that Velociraptor is embedded inside the containing compiler and it takes over the duties of generating code for numerical sections of the program. Numerical sections of the program may include numerical computations that execute on the CPU, as well as numerical computations that are annotated to be executed on GPUs. Our design goal was to ensure that the containing compiler requires minimal work to integrate Velociraptor.

Figure 2 demonstrates how our approach cleanly integrates with a containing compiler and provides a detailed overview of the entire toolchain including the services provided by Velociraptor. The conventional passes are shown in the dark shaded boxes, while our toolkit, Velociraptor, is shown in the overlay box on the right. The compiler developer who wants to integrate Velociraptor into an existing JIT compiler only needs to add two new components.

First, the compiler developer introduces a new pass in the containing compiler. This new pass identifies numerical parts of the program, such as floating-point computations using arrays including parallel-loops and potential GPU sections, that it wants to handover to Velociraptor. This new pass will be performed after conventional frontend passes and analysis such as type inference performed by many JIT compilers for dynamic languages. The identified numeric sections are outlined into functions and the containing compiler compiles the outlined functions to VRIR functions.

The second addition to be done by the compiler writer is to provide a small amount of glue code to expose the language runtime to Velociraptor. The glue code has several components. Velociraptor needs to know about the representation of array objects in the runtime. The compiler writer provides several typedefs from the existing implementation's array object structure to some typenames Velociraptor implementation uses, and also provides the implementation of macros and functions to access the fields of the array object structure. In addition, the compiler writer provides a representation of the array object structure in LLVM. Finally, Velociraptor defines an abstract API for memory management, such as array object allocations and managing reference counts (if applicable) and the compiler writer provides the implementation of the API. Thus, Velociraptor integrates cleanly into the existing data-structures and does not require invasive changes that might break compatibility with external libraries.

Velociraptor takes over the major responsibility of compiling 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.

Velociraptor provides analysis and transformation infrastructure for VRIR as well as code generation backends for CPUs and GPUs. Portability is a key concern for us and we chose to build our CPU and GPU backends upon portable technologies, LLVM for CPUs, and OpenCL for GPUs. Behind the scenes, Velociraptor also provides a smart runtime system VRRuntime to efficiently manage GPU memory, CPU-GPU synchronization and task dispatch. However, VRRuntime is completely transparent to the containing compiler, and the containing compiler does not need to do any work to use VRRuntime.

3. DESIGN OF VRIR

A key design decision in our approach is that the numeric computations should be separated from from the rest of the program, and that our IR should concentrate on those nu-

merical computations. This decision was motivated by an internal 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. With VRIR, our objective is to cover the common cases of numeric computations. VRIR is designed to be easy to generate from languages like MATLAB or Python and the constructs supported will be familiar to any MATLAB or Python/NumPy programmer. We have defined C++ classes corresponding to each tree node type. We have also defined an XML-based representation of VRIR. Containing compilers can build the XML representation of VRIR and pass that to Velociraptor, or alternatively can use the C++ API directly to build the VRIR trees. Note that 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 section we present the important details of VRIR. A detailed specification and examples are available on our website.[1] The basic structure and constructs of VRIR are introduced in Section 3.1 and supported datatypes are introduced in Section 3.2. 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 and operators are introduced in Section 3.3. In addition to serial constructs, VRIR supports parallel and GPU constructs in the same IR and these are discussed in Section 3.4. Finally, error handling and memory management are discussed in Section 3.5 and in Section 3.6 respectively.

3.1 Structure of VRIR programs and basic constructs in VRIR

Modules

The top-level construct in VRIR is a *module* and each module consists of one or more functions. Functions in a module can call either the standard library functions provided in VRIR or other functions in the same module, but cannot call functions in other modules.

Modules are self-contained compilation units. This design choice was made because we are targeting OpenCL 1.1, and OpenCL 1.1 requires that the generated OpenCL program should be self-contained. We will be able to remove this restriction once we start targeting OpenCL 1.2, which separates compilation of OpenCL kernels from linking. However, important vendors such as Nvidia currently only provide OpenCL 1.1 implementations.

Functions

Functions have a name, type signature, a list of arguments, a symbol table and a function body. The function body is a statement list.

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

Statements

VRIR supports common statements such as assignments, for-loops, while-loops, conditionals, break and continue statements, return statements, expression statements and statement lists. 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 3.4.

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.

3.2 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.

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.

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 arrays of row- or column-major layouts, indices are converted into addresses using the regular rules for such layouts. The strided-view layout is inspired from the `ndarray` datatype in Python's NumPy library.

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 and only sacrifices small amount of flexibility for most practical cases.

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.

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. The components can be obtained using constant integer indexes into a tuple.

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.

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.

3.2.1 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 [6], will often perform type inference before code generation. 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 [4] language or the widely used Python compilation tool Cython [23]. 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.

3.3 Array indexing and operators

3.3.1 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 the number and type of index parameters, as well as several optional attributes defined for the array indexing nodes. Consider a n-dimensional array A indexed using d indices. VRIR has the following indexing modes:

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). 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.

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.

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.

Row-major, column-major and strided arrays

The layout of the array being indexed determines the arithmetic used for converting the indices to addresses.

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.

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.

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.

3.3.2 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 reduc-

tion as well as transcendental and trigonometric functions operating element-wise on arrays.

3.4 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. The constructs are as follows:

Parallel-for loops and atomics

Parallel-loops are loops defined over a multi-dimensional domain where each iteration can be executed in parallel. We also support some atomic constructs, such as compare and swap, inside a parallel-for loop.

Implicitly parallel operators

VRIR has a number of implicitly parallel built-in operators such as matrix addition and matrix multiply. Velociraptor takes advantage of parallel libraries for these operators where possible.

Parallel map

Parallel map takes as input a function f and n arrays or scalars as inputs. The function f must be a scalar function that takes n scalars as inputs and returns a scalar. At least one of the arguments to map should be an array, and all the input arrays should have the same size and shape. The operator applies the function f element-wise to each element in the input arrays in parallel, and produces an output array of the same shape.

Reduction sum and product

We support two reduce operators: sum and product of an array either along a given axis or for all elements of the array.

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.

3.5 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 is 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 still help the programmer in debugging while lowering the execution overhead and simplifying the compiler.

3.6 Memory management

Different containing compiler implementations have different memory management schemes. VRIR and Velociraptor provide automatic cleanup of scalar variables, but for array variables we allow the containing compiler to use the appropriate memory management scheme. Containing compilers can insert explicit instructions in VRIR for array allocation, array deallocation as well as reference counting, as required. Thus we support three models: manual memory management, reference counting, as well as conservative GCs such as Boehm GC. It is up to the containing compiler to select the memory management model.

4. COMPILATION ENGINE

A compiler developer using Velociraptor only needs to concentrate on generating the high-level VRIR, and then Velociraptor provides an optimizing compilation engine that compiles VRIR to LLVM code for CPUs and OpenCL for GPUs. Velociraptor is a multipass compiler with many analysis and optimization phases. The key compilation phases are shown in Velociraptor are shown in Figure 3.

The high-level nature of VRIR was necessary for implementing many of these optimizations. We have implemented some optimizations specific to array-based languages, such as bounds-check elimination and memory reuse optimizations for array operators. These optimizations cannot be implemented in lower-level IRs like LLVM which have no concept of higher-level array constructs that we include in VRIR. Many of the optimizations are common to both CPUs and GPUs and our design decision to have both CPU and GPU sections in the same unified IR (VRIR) makes implementing such optimizations simpler. Finally, VRIR also allows passing some bounds-check information from the host compiler. VRIR allows enabling/disabling bounds-checks on each array subscript separately. Thus, if the host compiler knows that some array references do not need to be bounds-

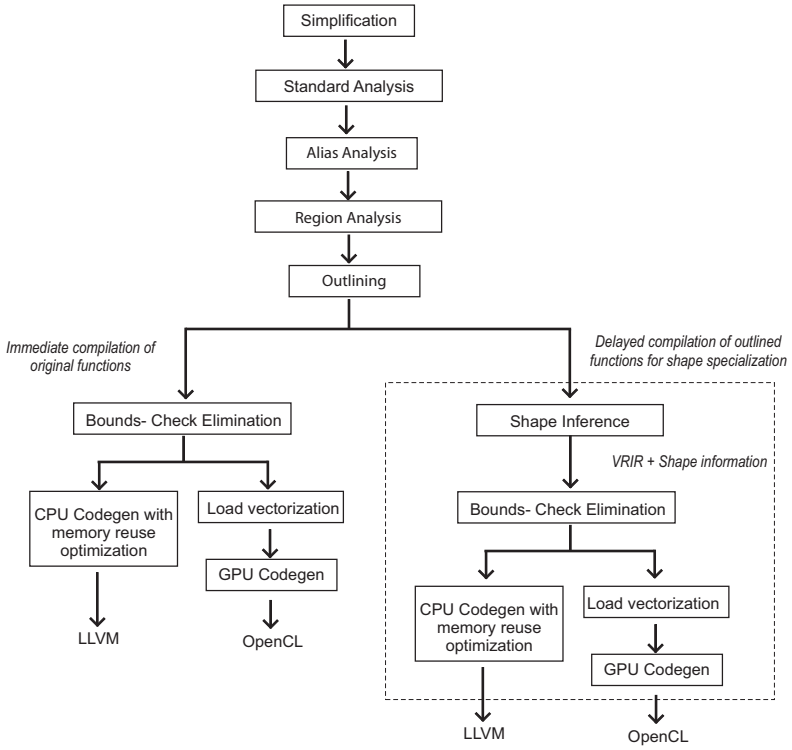

Figure 3: Compiler analysis, transformation and code generation infrastructure provided by Velociraptor

checked, then the host can simply encode this information in the VRIR given to Velociraptor.

Simplification and standard analysis

Velociraptor operates in multiple passes and the overall picture is shown in Figure 3. First, we simplify VRIR into a form similar to three address code. The simplification breaks down complex expressions involving array arithmetic, array indexing and function calls. Expressions which may perform implicit array allocation, such as matrix addition operators which return a new array as the result, are explicitly marked as allocating expressions. The simplifications enable simpler implementation of some flow analysis. Next, we perform some standard analysis such as live variables and reaching definitions analysis.

Alias analysis

We then perform alias analysis. Alias analysis is particularly important for languages like Python, where NumPy arrays may be aliased and having good aliasing information can enable the compiler to do better optimizations. We have implemented a fast, flow-insensitive, context-insensitive algorithm which proceeds in two phases. The algorithm operates upon an undirected graph with array variables and allocation sites as nodes and initially contains no edges. The first phase traverses statements inside the function and inserts edges as follows. We insert an edge between two variables when we encounter a reference array assignment of a variable to another variable. We insert an edge between a variable and an allocation site if the allocation is assigned to the variable. In the second phase, we find connected components in the graph which is a linear time operation in number of nodes and edges. Any two variables in the same connected component may be aliased.

Shape specialization

Many important optimizations depend upon accurate information about shapes in the program, such as sizes and strides of arrays as well as information about loop-bounds. We have implemented a just-in-time shape specialization in Velociraptor where the code is specialized based on shape information gathered at runtime. The analysis first performs *Region Analysis* that finds regions of code that may benefit from shape specialization. The identified regions are outlined into separate functions, and the compilation of these outlined functions is delayed until they are called. When the outlined functions are called, a shape inference pass determines the shape information and is used for code generation. Details of our JIT specialization system can be found elsewhere [9].

Array bounds-check elimination

Many array languages have array bounds-checks which can be a significant overhead, particularly on GPUs. There are many approaches of performing bounds-check elimination. However, Velociraptor is a JIT compiler and thus compilation time is an issue, thus we have chosen to implement two fast, but effective, intra-procedural techniques that are similar to previous work [14].

The first approach is based on elimination of redundant checks. Consider an array A and consider two references $A[expr_0, expr_1]$ and $A[expr_0, expr_2]$ which occur in the stated order inside the loop where $expr_0$, $expr_1$ and $expr_2$ are scalar expressions. If the value of $expr_0$ is the same in both array

references, then the bounds-check on the second instance of $expr_0$ can be eliminated.

The second approach is based upon moving some checks outside the loop. Consider a nested loop L. The compiler identifies a set S of array index expressions where each subscript is an affine function of loop variables. For each subscript in S, we generate a corresponding test outside the loop. We generate a new version L' of the loop where the bounds-check for subscripts in S are disabled. If the generated tests pass at runtime, then L' is executed otherwise the original loop L is executed.

Load vectorization

Next, for GPU sections, we perform *load vectorization*, an optimization where we attempt to merge loads into a single packed load. On some GPUs, effective memory read bandwidth for 128-bit loads is better than 64-bit or 32-bit loads. Load vectorization attempts to identify the scalar 32-bit or 64-bit loads that can be combined into wider loads. We perform load vectorization for two array load operations that have a distance of one (i.e. are contiguous), are known to be within bounds and and proven to not have any read-write dependencies in the loop.

CPU Code generation

The compiler compiles CPU code to LLVM. Considerable work was required to generate code for various constructs in VRIR. Some of the constructs that proved particularly challenging include parallel loops, the various flexible indexing schemes in VRIR, handling multiple return values, handling errors such as array out-of-bounds and finally handling library functions. VRIR has many library functions operating on arrays, such as array addition and we provide implementations of these library functions. Our implementation calls the BLAS where possible.

Memory reuse optimization for library functions

We have implemented a new optimization for CPU library functions. Consider the statement: $C = A + B$ where A, B and C are arrays. Language semantics of most array languages are that the expression $A + B$ will allocate a new array. A simple library function implementation will take A and B as inputs, allocate a new array of the right size, perform the computation and return the new array.

However, in some cases, the result array C may have a previous definition and may already be of the required size. In this case, we may be be able to avoid reallocating C. Thus, we have implemented a modified library function that takes C as input in addition to A and B, and checks if C is of the right size. The compiler only calls this optimized library function if C has a previous definition and if the array C is not aliased with other arrays. Overall, this optimization saves unnecessary memory allocations and reduces pressure on the memory allocator and garbage collector.

GPU code generation

Inside GPU sections, library functions as well as parallel loops are offloaded to the GPU. The GPU implementation of library functions is provided by RaijinCL [10] while parallel loops are compiled to OpenCL kernels. Velociraptor generates a task for VRRuntime for each computation to be offloaded to the GPU. Each VRRuntime task requires the specification of data dependences and Velociraptor generates this information. Control flow dependences, such as an if-conditional choosing between two parallel loops, usually need to be evaluated on the CPU. Velociraptor generates explicit synchronization calls to VRRuntime if required by control flow. Inside GPU sections, array allocations and operations such as array slicing are also handled by VRRuntime. Thus, our compiler framework is responsible for generation of correct code while our runtime manages the actual execution of the GPU kernels and management of GPU resources.

5. HIGH-PERFORMANCE RUNTIME

VRRuntime provides a simple GPU runtime API and takes over the responsibility of data transfers and task dispatch to the GPU, greatly simplifying the design of our compiler. VRRuntime provides a task queue to Velociraptor's compiler. Tasks encapsulate an operation such as a call to an OpenCL kernel generated by Velociraptor from a user-defined loop, a call to a library function such as matrix-multiply, or some other operation such as array assignment or slicing. Each task specifies the variables potentially read and written by the task. Based upon the data dependence information specified in the task, as well as actual aliasing information obtained at runtime, VRRuntime automatically determines the data transfers required between the CPU and GPU for completion of the tasks.

In contrast to previous GPU runtime research such as StarPU [2], our system is a higher level solution that is aware of high-level VRIR array operators. Providing the high-level API while providing optimizations such as asynchronous dispatch required solution of issues such as updating shape and alias information and these solutions are discussed in Section 5.2.

5.1 VRRuntime optimizations

VRRuntime has several important optimizations that allow for good performance:

Asynchronous dispatch

Enqueuing a task in VRRuntime is a non-blocking operation. The CPU thread that enqueued the operation can then continue to do other useful work until the result of the enqueued task is required on the CPU. At that point, the enqueuing thread can request the runtime to return a pointer to the required variable and VRRuntime will finish all necessary pending tasks and return the variable. Asynchronous dispatch allows the CPU to enqueue a large number of tasks to the OpenCL GPU queue without waiting for the first GPU task to finish. Inserting multiple kernels in the GPU's hardware job queue can keep it continuously busy, and lower the overheads associated with task dispatch.

Copy-on-write optimizations

Consider the case where variable A is explicitly cloned and assigned to variable B. VRRuntime does not perform the cloning operation until required. For each computation submitted to VRRuntime after the copy, VRRuntime examines the operands read and potentially written in the task using meta-information submitted to VRRuntime by the compiler. If VRRuntime does not encounter any computation that may potentially write A or B after the deep copy, then the deep copy operation is not performed. This is a variation of copy-on-write technique, where we can characterize

our technique as copy-on-potential-write. Copy-on-write has been implemented for CPU arrays in systems such as MATLAB [24], and for various CPU data-structures in libraries such as Qt, but we implement it for GPU arrays.

Data transfers in parallel with GPU computation

Consider when two tasks are enqueued to VRRuntime. After the first task is enqueued, instead of waiting for the first task to finish, VRRuntime initiates the data transfer required for the second task, if possible. Thus, the data transfer overheads can be somewhat mitigated by overlapping them with computation. Under the hood VRRuntime maintains two OpenCL queues because some OpenCL implementations require that data transfers be placed in a different OpenCL queue if they are to be performed in parallel with computations.

5.2 Maintaining shape and alias information

Shape information

Our runtime offers a higher level API that is aware of various array and matrix operators such as matrix multiplication and array slicing in VRIR. Thus, in contrast to APIs such as StarPU [2], our API allows the compiler to enqueue operators such as matrix multiplication without specifying any additional details such as the size of the operands.

However, the kernels are dispatched to the OpenCL queue, and OpenCL requires that the user of the API should specify the size of the execution domain of a kernel at the time of enqueuing the kernel. Thus, VRRuntime needs to automatically find out the shape of arrays involved in these higher-level operators at the time of enqueuing the kernel. In a simple CPU library call, the parameters such as array sizes are looked up at runtime from the array data-structure. In our case, due to asynchronous dispatch, the operands of a library call being enqueued may be the result of a previous task that has not yet completed and thus may not have been allocated yet. Further, the operands may never get allocated on the CPU if they are not utilized on the CPU. Thus, we cannot simply query CPU-side array data-structures to dispatch library calls on the GPU.

We solve the issue of shape information by maintaining the information about shapes of array variables in a separate table. The shape information maintained by VRRuntime is the information about shape of the array that would result if all the tasks that have been enqueued are completed. Whenever a task is enqueued, this shape information table is updated. For example, if a matrix multiplication call is enqueued, we update the shape information of the result array based upon shapes of the operands.

Alias and CPU-GPU synchronization information

Aliasing information is required to make correct decisions about mapping of variables to GPU buffers. Consider two arrays A and B. Some tasks may read/write A while some other tasks read/write B. Each array has an address range that it points to. Our strategy is that we want to allocate a unique GPU buffer for each CPU address range potentially accessed inside a kernel. Having a unique GPU copy of each address range ensures correctness when there is a data dependence within or across different kernels. If the arrays are not aliased, then we can simply allocate one GPU buffer corresponding to A, and a separate GPU buffer for B. However, if the arrays are indeed aliased, then we allocate a single GPU buffer that corresponds to the CPU address range encompassing both A and B.

We have chosen a compiler-assisted solution to the problem. Velociraptor analyzes the GPU section and finds all the arrays that may potentially occur inside the region and specifies this list to VRRuntime at the beginning of a GPU section. VRRuntime then constructs and maintains two tables of information. The first table contains information about array variables including the address range of the array, the GPU buffer corresponding to the array, the last task dispatched that might potentially write to the array, whether the CPU and GPU copy are synchronized and if not then whether the CPU or the GPU copy contains the freshest data. The second table has information about GPU buffers such as the CPU address range it corresponds to and the variables pointing to the GPU buffer. These tables are updated whenever a task is enqueued or a synchronization point is reached.

6. TWO CASE-STUDIES

6.1 McVM integration

McVM is a virtual machine for the MATLAB language. McVM is part of the McLAB project, which is a modular toolkit for analysis and transformation of MATLAB and extensions. McVM includes a type specializing just-in-time compiler and performs many analysis such as live variable analysis, reaching definitions analysis and type inference. Prior to this work, McVM generated LLVM code for CPUs only, and did not have any support for parallel loops or GPU computations. We added two language constructs to McVM. First, we added a parallel-for (parfor) loop. We have also provided *gpu_begin*() and *gpu_end*() section markers that indicate to the compiler that the section should be offloaded to the GPU, if possible.

In Section 2, we describe that a compiler writer must provide a pass to identify which sections to compile using Velociraptor, and also needs to provide glue code. We provide both of these pieces for McVM.

In this implementation, the method of choosing suitable numerical sections is to only invoke Velociraptor to compile parallel-for loops and GPU sections. We use McVM's existing code generation facilities for the rest of the program because McVM's LLVM based CPU code generator already implements some optimizations. We made the following modifications to McVM in order to support parfor loops and GPU sections. After McVM has finished type inference, McVM looks for parfor loops and GPU sections and first verifies that the code can be compiled to VRIR. For example, if some of the types were unknown, then they cannot be compiled to VRIR. If the code cannot be compiled to VRIR, then code is converted to serial CPU code and handled by McVM's usual code generator. If the code passes verification, then McVM outlines these new constructs into special functions, compiles them into XML representation of VRIR, then asks Velociraptor to compile and return the function pointers to the outlined code. The original code is then replaced by a call to the outlined function. In MATLAB, arguments are passed into functions by value. However, we have implemented calls to outlined functions as call-by-reference because the side effects in the outlined code

need to propagate back to the original function for correctness.

We have also provided glue code for exposing McVM's internal data-structures to Velociraptor. This required providing a McVM-specific implementation of Velociraptor's abstract memory allocation APIs, telling Velociraptor about MATLAB language semantics (such as using one-based indexing), exposing LLVM representation of various array classes and providing macros and functions for field access of array class members.

6.2 Python integration

We have written a proof-of-concept compiler for a numeric subset of Python, including the NumPy library, that integrates with the standard Python [18] interpreter.

As described in Section 2, any implementation needs to identify and outline numerical sections and needs to provide glue code. In our prototype, we require that the programmers outline suitable numerical sections manually. The idea is that only a few numerical functions in a Python program are compiled, while the rest of the program is interpreted by the Python interpreter. As for glue code, we have provided all the glue code necessary to interface Velociraptor with the necessary data structures in CPython and the NumPy library. We have provided glue code in C++ using the Python/C API for exposing NumPy arrays to Velociraptor, for exposing reference counting mechanism of Python interpreter to Velociraptor, and for reporting the out-of-bounds exceptions generated in code compiled by Velociraptor back to the user. We wrote a little glue code in C++ to wrap the function pointers returned by Velociraptor into Python functions.

Now we briefly describe the language constructs handled and extensions provided by our Python compiler. Our compiler can handle scalar numerical datatypes, NumPy arrays and tuples. Many of the commonly used constructs are handled including for-loops, while-loops, conditionals, various array indexing and slicing operators as well as many math library functions for arrays. We currently require that the programmer provides the type signature of the function through a decorator we have defined, and then our compiler infers the type of the local variables. We require that the type of a variable not change within a function. Some of these limitations can be removed if a just-in-time type specializing compiler is implemented. However, this is out of scope of this paper.

We have provided several extensions to the Python language to enable use of multi-cores and GPUs. First, we have defined a parallel-for loop by defining a special function *prange*, and have defined that any for-loop that iterates over a *prange* will run in parallel. Second, we have defined *prange* to return a multi-dimensional domain object rather than a single-dimensional range. Finally, we also provide constructs (*gpu_begin*() and *gpu_end*()) that act as markers to annotate blocks of code to be executed on the GPU.

In this compiler, all the code generation is handled by Velociraptor. The frontend of the compiler is written in Python itself. Python has a standard library module called *ast* which provides functionality to construct an AST from Python source code. Our frontend uses this module to construct untyped ASTs. Then our Python compiler performs type inference on this AST by propagating the types of the function parameters provided by the user into the body of the function. The function is then compiled by our Python compiler to VRIR in a separate pass.

6.3 Development Experience

For Python, where we required the programmer to manually separate the numerical sections, the development experience was very straightforward. For McVM, we found that building the outlining infrastructure was the most challenging aspect of the project. McVM maintains various internal data-structures holding information from various analyses such as live variable analysis. After outlining, all such analysis data-structures need to be updated and doing this correctly required some effort. However, once outlining was done, generating VRIR was straightforward.

Overall, we found that using Velociraptor enabled some saving in the development time of both compilers. As an indirect estimate, we can look at the number of lines of code (LoC) required. For McVM, the integration required about 5000 lines of code and we think that there is further possibility of reduction in LoC through better refactoring. Our Python compiler required only about 4000 lines of C++ and 300 lines of Python which is substantially smaller than Velociraptor. Velociraptor itself is about 11000 lines of code. Further, some of the implementation details of Velociraptor are quite subtle, such as asynchronous dispatch and require effort much beyond what the LoC metric suggests. Code reuse across different compilers really improved the development experience. For example, if a McVM test case failed and was found to be a Velociraptor bug, then fixing the bug often also benefited our Python compiler.

A secondary benefit of our design is that the technique of upfront identification and separation of well-behaved numerical sections from other parts of the program turned out to be very useful. Analysis and optimizations operating on VRIR can be more aggressive in their assumptions and can be simpler and faster to implement compared to an IR designed for representing the entire program. We have been able to very quickly prototype ideas about optimizations in Velociraptor without worrying about the full, and potentially very complicated, semantics of languages like MATLAB.

7. CASE STUDIES: EXPERIMENTAL RESULTS

Although the focus of this paper is the semantics of VRIR and design of the toolkit, we also wanted to see what the potential performance benefits could be. To get a first idea of the performance possibilities, we evaluated the performance of code generated by Velociraptor for both CPUs and GPUs on a variety of Python and McVM benchmarks.

7.1 Machine information

Our experiments were performed on two different machines containing GPUs from two different families. Descriptions of the machines are as follows:

1. Machine M1: Core i7 3820, 2x Radeon 7970, 8GB DDR3, Ubuntu 12.04, Catalyst 13.4
2. Machine M2: Core i7 920, Tesla C2050, GTX 480, 6GB DDR3, Ubuntu 12.04, Nvidia driver version 280.13.

Machine M1 and M2 contain two GPUs each. We used one Radeon 7970 and Tesla C2050 respectively for computation,

and reserve the other GPU in each machine for handling display duties. We used CPython v3.2, PyPy 2.1 and MATLAB R2013a 64-bit in tests. Each test was performed ten times and we report the mean.

7.2 McVM performance

For McVM, we looked at benchmarks used by previous projects such as McVM and MEGHA [16] and chose four parallelizable benchmarks and added GPU annotations. The benchmarks in this section make heavy use of implicitly parallel vectorized operators such as vector arithmetic operators. We distinguish three cases here: McVM performance without Velociraptor, multi-core performance with Velociraptor and GPU-accelerated performance with Velociraptor. We used MATLAB as the baseline and report performance in Table 1 as speedups over MATLAB.

All the benchmarks contained GPU sections. For the Velociraptor generated GPU version, we tested under four settings: Velociraptor compiler and runtime optimizations enabled, compiler optimizations disabled but runtime optimizations enabled, compiler optimizations disabled but runtime optimizations disabled and finally both compiler and runtime optimizations disabled.

However, in order to compare McVM and Velociraptor generated CPU code, we also ran the tests where we forced Velociraptor to only generate CPU code. We found that the CPU code generated by Velociraptor was never slower than the code generated by McVM's LLVM-based backend and thus there is no disadvantage in using Velociraptor even for CPU code.

For the GPU version, Velociraptor runtime optimizations provide up to 15% increase in performance. This is primarily due to asynchronous dispatch implemented in the runtime. Asynchronous dispatch allows the CPU to enqueue a large number of tasks without waiting for the GPU to finish. Synchronous dispatch adds overhead for waiting upon completion of kernel calls and this overhead can be significant for small kernels such as vector addition.

7.3 Python performance

We evaluated Python performance on four Python benchmarks. These benchmarks come from a set of Python benchmarks proposed by members of NumPy community. We added type, parallelization and GPU section annotations in these benchmarks.

We tested our compiler on three different versions of each benchmark: a serial CPU version, a parallel CPU version and a GPU version. For the CPU versions, we we tested the code generation in two cases: with Velociraptor optimizations enabled and disabled. For the GPU version, we tested under four settings similar to the ones discussed in previous section. Finally, we measured the the performance of the PyPy 2.1 implementation of Python. PyPy has its own JIT compiler and we found that PyPy was significantly faster than CPython on these benchmarks and thus we use PyPy as a realistic baseline. The results are presented in Table 2 as speedups over PyPy.

We observed that serial unoptimized CPU code generated by Velociraptor is between 3 to 37 times faster than PyPy depending upon the benchmark. Parallel CPU code generated by Velociraptor was generally two to six times faster than serial CPU code generated by Velociraptor. We used quad-core processors and more than four times speedup can be explained by the fact the CPUs are hyper-threaded and run eight threads. Optimized GPU code generated by Velociraptor can be up to seven times faster than generated optimized parallel CPU code provided that the data transfer overhead is not substantial.

Compiler optimizations provided between 1% to 35% performance improvement on CPUs and between 1% and 52% performance improvement on GPUs depending upon the benchmark. We observed that the compiler eliminated between 50 and 100 percent of array bounds-checks in the innermost loop in these benchmarks.

8. RELATED WORK

There has been considerable interest in using GPUs from dynamic array-based languages. The earliest attempts have been to create wrappers around CUDA and OpenCL API that still require the programmer to write the kernel code by hand and exposing a few vendor specific libraries. Such attempts include PyCUDA [11] and PyOpenCL [12]. The current version of MATLAB's proprietary parallel computing toolbox also falls in this category at the time of writing. Our approach does not require writing any GPU code by hand.

There has also been interest in compiling array-based languages to GPUs. Copperhead [5] is a compiler that generates CUDA from annotated Python code. Copperhead does not handle loops, but instead focuses on higher-order functions like map. jit4GPU [8] was a dynamic compiler that compiled annotated Python loops to AMD's deprecated CAL API. Theano [3] is a Python library that compiles expression trees to CUDA. In addition to GPU code generation, it also includes features like symbolic differentiation. Parakeet [21] is a compiler that takes as input annotated Python code and generates CUDA for GPUs and LLVM for CPUs. MEGHA[16] is a static compiler for compiling MATLAB to mixed CPU/GPU system. Their system required no annotations, and discovered sections of code suitable for execution on GPUs through profile-directed feedback. Jacket [17] is a proprietary add-on for MATLAB that exposes a large library of GPU functions, and also has a compiler for generating GPU code for limited cases. Numba [15] is a NumPy-aware JIT compiler for compiling Python to LLVM. The work has some similarities to our work on the CPU side, including support for various array layouts, but it is tied to Python and therefore does not support the indexing schemes not supported by Python. Numba also provides an initial prototype of generating CUDA kernels, given the body of the kernel (equivalent to the body of a parallel loop) in Python. However, it assumes the programmer has some knowledge of CUDA, and exposes some CUDA specific variables (such as thread-block index) to the programmer. Furthermore, unlike our work, it is not a general facility for annotating entire regions of code as GPU-executable and will mostly be useful for converting individual loop-nests to CUDA.

One possible approach to building a multi-language compiler for GPUs is to compile bytecode of high-level VMs to GPUs. This is complementary to our approach and there are two recent examples of this approach. AMD's Aparapi [1] provides a compiler that compiles Java bytecode of a method to an OpenCL kernel body, and the generated kernel can then be applied over a domain. Unlike VRIR, which contains multi-dimensional arrays and high-level array operators, Aparapi is a low-level model where only one-dimensional

Benchmark	Machine	McVM	Velociraptor CPU		GPU accelerated			
Compiler opts		N/A	No	Yes	No	Yes	No	Yes
Runtime opts		N/A	N/A	N/A	No	No	Yes	Yes
clos	M1	0.99	0.99	0.98	3.18	3.22	3.20	3.25
	M2	1.0	0.99	0.99	2.24	2.18	2.24	2.28
nb1d	M1	1.58	1.50	1.61	2.92	2.88	3.04	3.04
	M2	1.30	1.34	1.45	3.40	3.37	3.62	3.61
nb3d	M1	0.2	0.51	0.54	1.64	1.67	1.72	1.72
	M2	0.22	0.72	0.72	2.43	2.40	2.63	2.63
fdtd	M1	0.15	0.47	0.47	1.92	1.90	1.97	1.97
	M2	0.13	0.54	0.53	2.04	2.04	2.12	2.08

Table 1: Speedup of McVM + Velociraptor generated code over MATLAB JIT

Benchmark	Machine	CPU Serial		CPU Parallel		GPU accelerated			
Compiler opts		No	Yes	No	Yes	No	Yes	No	Yes
Runtime opts		N/A	N/A	N/A	N/A	No	No	Yes	Yes
arc-distance	M1	3.36	4.15	9.84	11.64	8.19	9.3	8.33	9.63
	M2	3.12	3.55	10.9	11.91	8.84	9.75	8.59	9.72
julia	M1	34.8	37.0	213.4	214.8	1517	1502	1525	1511.8
	M2	37.0	37.5	170.2	172.2	729.3	775.8	744	756
growcut	M1	13.3	18.5	42.8	60.1	118.1	168.4	116.6	173.4
	M2	11.5	15.4	37.9	49.7	90.3	137.9	93.2	137.7
rosen-der	M1	15.5	18.7	31.1	40.0	32.6	33.8	32.9	34.1
	M2	12.9	15.7	25.0	29.4	25.6	28.0	25.6	28.4

Table 2: Speedup of CPython + Velociraptor generated code over PyPy JIT

arrays are allowed with Java's simple indexing operators. Dandelion [20] is a LINQ style extension to .net that cross-compiles .net bytecode to CUDA. The programming style for LINQ is quite different from the loop and array-operator based approach in our work.

To our knowledge, there has not been prior work on embeddable or reusable compilers for GPUs. The only work in this area that we are aware of is NOVA [7]. NOVA is a static compiler for a domain-specific compiler for a new functional language and it generates CUDA code. Velociraptor and NOVA provide different programming styles (imperative loop and array-based vs functional respectively) and different compilation models (dynamic vs static respectively). Collins et al. claim that NOVA can be used as an embedded domain-specific language but did not show any such embedded uses whereas we integrated our toolkit in two compilers.

To summarize, in contrast to previous research, our toolkit is an embeddable and reusable compiler targeting both CPUs and GPUs and is not limited to one programming language. We have carefully considered the variations of array indexing schemes, array layout semantics and array operators of the programming languages and offer a complete solution for popular scientific computing languages.

However, while we described the differences from previous research, our toolkit is meant to complement and enable, not compete, with other compiler researchers. Had our tools existed earlier, most of the previously mentioned systems could have used it while focusing their time elsewhere. For example, MEGHA's automatic identification of GPU sections, the type inference work of Parakeet or Theano's work on symbolic facilities are all complementary to our work. We believe toolkits such as Velociraptor will free other researchers to work on various open problems in programming language design and implementation rather than spend the time writing backends or design GPU runtimes.

9. CONCLUSIONS AND FUTURE WORK

The development of JIT compilers for array-based languages to produce efficient CPU and GPU code is an important problem. In some cases, compiler writers may be writing entirely new language runtimes. In other cases, language implementors are more interested in evolving current language runtimes for the sake of compatibility and development time.

We first discussed some design goals of our project and how certain designs are ruled out because of constraints such as handling existing language runtimes. Based on our design goals, we proposed a toolkit-based approach to simplify the building of JIT compilers for array-based languages. We have presented an embedded toolkit that can be used by compiler writers inside their own compiler to compile numerical array-based code to both CPU as well as GPU code. The toolkit is reusable across two languages. We discussed a new high-level IR called VRIR to achieve this language portability. VRIR can be used to express a wide variety of core array-based computations and can contain code sections for both CPUs and GPUs. The toolkit is designed to be embedded inside existing language implementations. In order to simplify the evolution of existing codebases, our toolkit does not require rewriting of data-structures such as array representations inside the language runtime.

The toolkit provides optimization and code generation facilities as well as a GPU runtime. Compiler writers can simply generate VRIR for key parts of their input programs, and then use Velociraptor to automatically generate CPU/GPU code for their target architecture. Our compilation engine

and runtime library takes care of many issues, including the communication between the CPU and GPU as well as capturing errors. We have implemented many standard analysis and optimizations. While some general optimization techniques are well-known, we faced some challenges in adapting them to our framework. We discuss our solutions to problems such as as asynchronous dispatch of GPU kernels and reusing memory allocations in CPU library functions to our system.

To demonstrate the toolkit, we used it in two very different projects. The first project was using the toolkit to extend a MATLAB JIT, from McVM, to handle GPU computations. The second is a proof-of-concept compiler for Python, which was written as an add-on to the existing CPython implementation, where we used both the CPU and GPU code generation capabilities of Velociraptor.Thus, we showed that Velociraptor can handle the language semantics of two different languages, and that our APIs and implementation are generic enough to be interfaced with two different existing language runtimes.

For our framework, now that the toolkit is established and we have two prototype applications of the toolkit, we are working on refining those prototypes and adding further optimizations and transformations to the generated code as well as adding more backends such as a static CPU backend. We are also working on performance studies with a broad range of CPU-GPU hybrid systems and on a much larger benchmark set.

To summarize, we demonstrated that a toolkit based approach can work for building compilers for array-based languages. A shared, reusable toolkit can be very useful for the community and our presented compiler toolkit is the first such framework. We hope that other researchers will use our toolkit for their own compilers. Most importantly, we view our toolkit as a starting point and we hope that our work will spark the discussion in the community about the need and potential designs of such toolkits.

10. REFERENCES

[1] Advanced Micro Devices Inc. Aparapi. `http://code.google.com/p/aparapi/`.

[2] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, Feb. 2011.

[3] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: a CPU and GPU math expression compiler. In *SciPy 2010*, June 2010.

[4] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman. Julia: A fast dynamic language for technical computing. *CoRR*, abs/1209.5145, 2012.

[5] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. In *PPOPP 2011*, pages 47–56, 2011.

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

[7] A. Collins, D. Grewe, V. Grover, S. Lee, and A. Susnea. NOVA : A functional language for data parallelism. Technical report, Nvidia Research, 2013.

[8] R. Garg and J. N. Amaral. Compiling Python to a hybrid execution environment. In *GPGPU 2010*, pages 19–30, 2010.

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

[10] 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 22nd Euromicro International Conference on Parallel, Distributed and network-based Processing, Special session on GPU computing*, 2014.

[11] A. Klöckner. PyCUDA. `http://mathema.tician.de/software/pycuda`.

[12] A. Klöckner. PyOpenCL web page. `http://mathema.tician.de/software/pyopencl`.

[13] MathWorks. MATLAB: The Language of Technical Computing.

[14] N. T. V. Nguyen, F. Irigoin, C. Ancourt, and R. Keryell. Efficient intraprocedural array bound checking. Technical report, Ecole des Mines de Paris, 2000.

[15] T. Oliphant. Numba Python bytecode to LLVM translator. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2012. Oral Presentation.

[16] A. Prasad, J. Anantpur, and R. Govindarajan. Automatic compilation of MATLAB programs for synergistic execution on heterogeneous processors. In *PLDI 2011*, pages 152–163, 2011.

[17] G. Pryor, B. Lucey, S. Maddipatla, C. McClanahan, J. Melonakos, V. Venugopalakrishnan, K. Patel, P. Yalamanchili, and J. Malcolm. High-level GPU computing with Jacket for MATLAB and C/C++. *Proceedings of SPIE (online)*, 8060(806005), 2011.

[18] Python.org. Python Programming Language: Official Website.

[19] R-project.org. The R Project for Statistical Computing.

[20] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. In *SOSP'13: The 24th ACM Symposium on Operating Systems Principles*, 2013.

[21] A. Rubinsteyn, E. Hielscher, N. Weinman, and D. Shasha. Parakeet: A just-in-time parallel accelerator for Python. In *HotPar 12*, 2012.

[22] SciPy.org. NumPy: Scientific Computing Tools for Python.

[23] 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.

[24] L. Shure. Memory management for functions and variables. `http://blogs.mathworks.com/loren/2006/05/10/memory-management-for-functions-and-variables/`.