McGill University
School of Computer Science
Sable Research Group

Velociraptor: A compiler toolkit for numerical programs targeting CPUs and GPUs

Sable Technical Report No. sable-2013-5 supercedes 2012-3

Rahul Garg and Laurie Hendren

November 16, 2012

Contents

List of Figures

List of Tables

Abstract

Developing compilers that allow scientific programmers to use multicores and GPUs is of increasing interest, however building such compilers requires considerable effort. We present Velociraptor: a portable compiler toolkit 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 accelerator constructs. A compiler developer uses Velociraptor by generating VRIR for key parts of an input program. Velociraptor does the rest of the work by optimizing the VRIR code, and generating LLVM for CPUs and OpenCL for GPUs. Velociraptor also provides a smart runtime system to manage GPU resources and task dispatch.

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

1 Introduction

Two trends in scientific computing have become mainstream. First, array-based languages such as MATLAB [**?**], Python/NumPy [**?**] and R [**?**] have become extremely popular for scientific and technical computing. The second trend is that accelerators such as general purpose graphics processing units (GPGPUs) and many-core processors such as Xeon Phi have become popular, with thousands of scientific papers published utilizing GPUs.[1] However, programming GPUs is still not accessible to the mainstream scientific programmer.

Making GPUs more accessible to the general MATLAB or Python user has been of increasing interest to the programming language design and implementation community. However, writing compilers targeting hybrid CPU+GPU systems is challenging. The compiler writer has to implement new code generation backends, analysis and transformation infrastructure and possibly a runtime component to manage GPU resources. Currently, every compiler project targeting hybrid systems is forced to write their infrastructure from scratch, making programming language and compiler research substantially harder. To address this problem we have implemented a compiler toolkit called **Velociraptor** aimed at making it easy to build just-in-time (JIT) compilers targeting hybrid CPU+GPU systems. Velociraptor is a reusable and portable compiler infrastructure that can be used to build compilers for a variety of array-based languages and that works on a variety of hardware configurations.

A key ingredient in our design is our intermediate representation called VRIR. VRIR is a new domain-specific, high-level IR that has been carefully designed to be easy to generate from high-level languages for numeric computing. One of the defining features of VRIR is the flexible array datatype and rich array operator semantics to support its goal of being a DSL specialized for numeric computing where arrays are extremely important. Further, it is not tied to a particular language and many operators are parameterized with various flags to accommodate variations between languages. VRIR is flexible enough to be able model a large subset of numerical computing semantics from MATLAB and Python/NumPy. Finally, VRIR also makes it very easy to target both CPUs and GPUs by supporting serial constructs, parallel programming constructs and high-level constructs for GPUs in the same IR.

Velociraptor provides three key components. First, it provides an analysis and transformation

[1] `http://www.hgpu.org` lists over 7000 GPU-related publications currently

infrastructure for VRIR. We have implemented common analyses such as live variable analysis, and also implemented some transformations such as array-bounds checks elimination and automatic use of on-chip shared memory for GPUs. Second, we provide code generation backends targeting CPUs and GPUs. Many compiler writers want to target a wide variety of hardware platforms and Velociraptor enables them to do so. Velociraptor builds upon portable technologies such as LLVM for CPUs and OpenCL for GPUs and we have carefully designed our approach to ensure that the system works on a variety of hardware configurations. Finally, we also provide a smart runtime system to manage GPU resources and task dispatch to support the compiler.

To demonstrate the reusability of Velociraptor, we have used it in two projects. The first project is a JIT compiler that takes as input annotated Python code and generates CPU and GPU Python code using Velociraptor. This demonstrates how a new compiler could leverage Velociraptor to do both the CPU and GPU code generation. The other is an extension of McVM [**?**], a just-in-time compiler and interpreter for MATLAB language. In this project, we use Velociraptor to provide support for GPUs, while CPU code generation is done by McVM.

Our contributions are as follows:

A reusable and portable compiler toolkit for CPUs and GPUs: We have designed and implemented Velociraptor, a reusable and portable just-in-time compiler toolkit for targeting CPUs and GPUs for array-based languages. The reusability of our toolkit distinguishes it from previous compiler efforts for GPUs.

New IR, VRIR: VRIR is a new domain-specific, high-level IR for numeric programs that is easy to generate from high-level languages like MATLAB while also providing constructs to target modern hardware like multi-core CPUs and GPUs.

Smart compiler and runtime: We have implemented an optimizing compiler toolkit with various analysis and optimizations such as array bounds-check elimination and automatic use of on-chip shared memory for GPUs. The runtime manages GPU resources, performs task dispatch and performs optimizations such as overlapping communication and computation where possible.

Two case studies of using Velociraptor: We have built two case studies, McVM and Python, of using Velociraptor. These projects serves both as an illustration of the reusability of Velociraptor in compilers for two different languages and also provide a useful contribution for users of McVM and Python respectively. We evaluate the performance of Velociraptor and the implemented optimizations using benchmarks from both McVM and Python on hardware from multiple vendors.

Our project is heavily inspired from the success of toolkits such as LLVM [**?**]. LLVM has enabled many exciting developments in the compiler community by providing a shared and highly reusable infrastructure. Our hope is that Velociraptor will similarly enable other compiler developers to further the state of compiler research in the domain of using modern hardware from high-level numeric computing languages. Compiler writers interested in integrating Velociraptor for new languages or implementations can do so with minimal effort and reuse the infrastructure provided. On the other hand, compiler writers interested in experimenting with compiler optimizations can do so by extending Velociraptor, and can benefit any compiler project using Velociraptor.

The remainder of this paper is structured as follows. In Section 2 we present the overall design and in Section 3 we present the VRIR. Section 4 provides key background on our building blocks, LLVM and OpenCL. Section 5 describes our Velociraptor code generation infrastructure, along with its run-time system. We demonstrate our toolkit with two integration examples in Section 6. Experimental results are presented in Section 7. We finish with related work in Section 8 followed by conclusions and future work in Section 9.

2 Using Velociraptor in compilers

A typical just-in-time compiler for a dynamic language targeting CPUs (and not GPUs) takes as input program source code, converts into its intermediate representation, does analysis (such as type inference), possibly does code transformations and finally generates CPU code. We call this type of compiler as a host compiler while Velociraptor is called the embedded compiler.

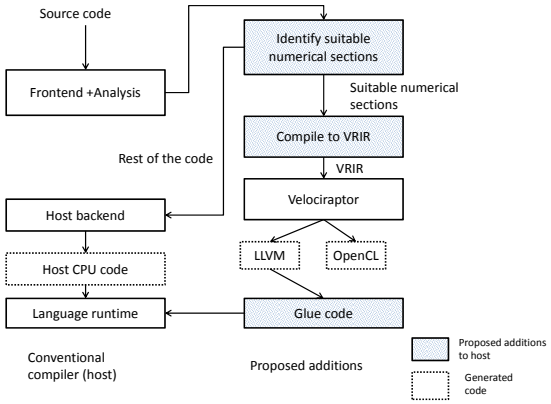

Figure 1: Possible design of a host compiler utilizing Velociraptor. The left of the picture has parts of a conventional host compiler, while the right indicates proposed modifications

We have built Velociraptor toolkit to simplify the building of host compilers that want to compile numerical computations to CPUs and GPUs. The key idea is that Velociraptor is embedded inside the host compiler and 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. Velociraptor can be used in various ways by different host compilers. Existing JIT compilers with mature CPU backends may choose Velociraptor for compiling only the GPU sections of the program, while new projects may choose Velociraptor to generate code for numeric sections for both CPUs and GPUs. We illustrate two case studies of these two types of uses in Section 6.

Figure 1 demonstrates how our approach cleanly integrates with a host compiler, with the proposed additions inside the overlay box on the right, and the conventional passes on the left. Our design goal was to ensure that the host compiler requires minimal work to integrate Velociraptor. The modified host compiler can be structured as follows. The host compiler first performs its regular frontend and analysis activities, including analyses such as type inference performed by many JIT compilers for dynamic languages today. Then the compiler developer introduce a new pass in the host 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. The identified numeric sections are outlined into functions and the host compiler compiles the outlined functions to VRIR functions. The host compiler also provides a small amount of glue code to expose aspects such as representations of arrays and the memory manager used in the host compiler's language runtime to Velociraptor.

Velociraptor takes over the major responsibility of compiling VRIR to CPU and GPU code. Portability is a key concern for us and we chose to build our CPU and GPU backends upon portable technologies like LLVM and OpenCL for CPUs and GPUs respectively. We provide some background on these technologies in Section 4. For each VRIR function, Velociraptor compiles it to LLVM and OpenCL as required and returns a pointer to a function to the host compiler. The host 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 host compiler in its normal fashion.

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 numerical 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 the 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. Host compilers can build the XML representation of VRIR and pass that to Velociraptor, or alternatively the host 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 host compiler normally, using the host compiler's CPU-based code generator and do not go through Velociraptor.

In the remainder of this section we presents the important details of VRIR. A detailed specification and examples are available on our website.[2] 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 is discussed in Section 3.5 and memory management is discussed in Section 3.6.

[2]URL submitted as additional material.

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.

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 host 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 discuss how our design decision of typed VRIR can be accommodated in host compilers for dynamically typed languages in Section 3.2.1. The supported datatypes are presented in Section 3.2.2

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 will often perform type inference before code generation. For example, McVM dynamically specializes the code of a function at runtime, based on types of the actual parameters [**?**]. The McVM project shows that a type-specializing JIT compiler can infer the types of a large number of variables at runtime. Part of the reason of the success of McVM is that while MATLAB is dynamically typed, scientific programmers don't often use very dynamic techniques in the core numeric functions compared to applications written

in more general purpose languages like JavaScript where dynamic techniques are more common. If a compiler is unable to infer types, users are often willing to add a few type hints for performance critical functions to their program. We can look at examples such as the Julia [**?**] language, which is gaining popularity, or the widely used Python compilation tool Cython [**?**], both of which allow optional static type declarations. Finally, in cases where the host compiler is unable to infer the types of variables, it simply will not generate VRIR and instead can use its regular code generation backend for CPUs as a fallback instead of using Velociraptor.

3.2.2 Datatypes

We have carefully chosen the datatypes in VRIR to be able to represent useful and interesting numerical computations. The type system includes scalars and arrays, which are the building blocks for numeric computing. Domains, inspired from languages like X10 [**?**], are provided to represent iteration domains and are a superset of the *range* datatype from Python. Tuples provide simple heterogeneous containers. Function types allow the passing of other numeric datatypes and they also support providing functions as inputs to allow expressing optimizers and solvers that take other functions as inputs. The details are as follows:

Scalar types: VRIR has real datatypes 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 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. To explain the strided-view layout, consider an n-dimensional array A with base address $base(A)$. Then, the strided-view layout specifies that the n-dimensional index vector $(i_1, i_2, ..., i_n)$ is converted into the following address: $addr(A[i_1, i_2, ..., i_d]) = base(A) + \sum_{k=1}^{d} s_k * i_k$ (in a 0-based indexing scheme). The values s_k are called strides of the array. Languages like MATLAB do not have strided views. In the case of McVM, all matrices are column-major and the VRIR generated by McVM simply declares all variables as column-major.

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 array dimensions were 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 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. Languages like MATLAB do not have an explicit domain type but some languages like X10 do.

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 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 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.

3.3 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:

All n indices provided: If each index is an integer, then the address selected is calculated using rules of the layout of the array (row-major, column-major or stride-based). If one or more indices is a range, then a slice of the array in the corresponding domain is selected instead of an individual element in that dimension. Similar to NumPy, but unlike MATLAB, array slices in VRIR do not create copies, but instead create new strided views of the same data. Thus, host compilers implementing languages like MATLAB must insert explicit copy instructions where necessary.

Less than n indices provided: If $d < n$, 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.

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.

Enabling/disabling index bounds checks: Array bounds-checks can be enabled/disabled for each individual indexing operation. This allows the host 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 host compiler, controls whether one-based or zero-based indexing is used for the language.

Slicing arrays - Data sharing or copying: Consider an array slicing operation such as $A[m : n]$. Some languages like Python have array slicing operators that return a new view over the same data while other languages like MATLAB return a copy of the data. We support both possibilities in VRIR through a boolean attribute of array index nodes.

3.3.2 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 reduction operators 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. Velociraptor transforms VRIR to a lower-level IR that exposes some lower-level details. Researchers interested in experimenting with compiler optimization algorithms for GPUs can do so by extending Velociraptor directly with new transformation passes from VRIR to Velociraptor's backend IR.

3.4.1 Parallel programming constructs

Targeting multicores and GPUs is one of the main goals of Velociraptor and both require expressing parallelism in the computation. Our aim is to support high-level constructs that are easily understood and used by scientific programmers. Therefore, we have selected three parallel constructs that are small variations of existing serial constructs and may be familiar to many programmers from other parallel programming literature. The supported constructs are:

Parallel-for 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.

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.

Implicitly parallel operators: VRIR has a number of implicitly parallel built-in operators such as matrix addition, matrix multiply and reduction sum of an array. Velociraptor takes advantage of parallel libraries for these operators where possible.

3.4.2 Accelerator sections

Any statement list can be marked as an *accelerated section*, and it is a hint to Velociraptor that the code inside the section should be attempted to be executed on a GPU if present. Velociraptor and VRRuntime manage all the required data transfers between the CPU and GPU automatically and ensure that the correct data is available to both CPU and GPU at necessary points. 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, such as 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 to perform more optimizations or to employ more sophisticated scheduling strategies than those possible for single loop-nest sections.

3.5 Error reporting

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. In VRIR, we have chosen to report some errors and the model is similar to throwing an exception. 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.

Two types of errors are supported for CPU serial code: array bounds checks and integer division by zero exceptions. VRIR also provides some error reporting for parallel CPU-loops and GPU sections. Multiple exceptions can happen in parallel inside such sections, but only one of them will be reported, and 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 the serial case, but these will still help the programmer in debugging while lowering the execution overhead and simplifying the compiler.

If an error occurs, then a cleanup function specified by the host compiler can be invoked. This cleanup function takes array variables local to the function as input and does not return any output. This is primarily meant as a way to perform any memory cleanup required by the host and may not be required by all hosts. After cleanup is finished in the VRIR function that generated the error, the error is propagated up the call chain until the entire call-chain of VRIR functions return after performing any cleanup. After that, it is up to the host to decide what to do with the error code. In languages like Python, it is possible to write a wrapper that raises the appropriate exception from the errorcode. For accelerator sections and parallel-loops, the cleanup function, if any, will be inserted after the parallel-for loop or accelerator section.

3.6 Memory management

Different host implementations have different memory management schemes. VRIR and Velociraptor provide automatic cleanup of scalar variables, but for array variables we allow the host to use the appropriate memory management scheme. Hosts can insert explicit instructions in VRIR for array allocation, array deallocation as well as reference counting, as required. This scheme allows for implementation of various scenarios:

1. Host compilers which require explicit memory management can insert explicit allocation and deallocation instructions.
2. Hosts which require reference counting can insert instructions to increase or decrease reference counts.
3. Hosts which utilize a garbage collector such as Boehm GC can skip insertion of memory deallocation instructions altogether.

4 Background

In order to provide a general-purpose tool which could be retargeted for a variety of CPU and GPU processors, we designed Velociraptor to build upon two excellent infrastructures, LLVM for

the CPU backend, and OpenCL for the GPU backend. To provide some background for the rest of this paper, we give a brief overview of these technologies.

We use the OpenCL [**?**] v1.1 programming model. The OpenCL API is an industry standard API for parallel computing on heterogeneous systems. Different vendors can provide drivers that implement the API. Currently, implementations exist for CPUs, GPUs and FPGAs. In OpenCL, the CPU controls one or more compute devices. The OpenCL API provides functions to manage various resources associated with the device.

OpenCL programs are written as *kernel functions* in the OpenCL kernel language. The OpenCL kernel language is based upon the C99 language. However, some restrictions are imposed. For example, function pointers and dynamic memory allocation are both disallowed in kernel functions. Kernel functions are represented as strings in the application, and the application requests the device driver to compile the kernel to device binary code at runtime.

OpenCL kernel functions describe the computation performed by one thread, called one *work-item* in OpenCL terminology. Kernel functions are invoked by the CPU on a 1,2 or 3 dimensional grid of work items, with each work item executing the same kernel function but each having its own independent control flow. This model maps naturally to data-parallel computations. Grids of work items are organized in 1,2 or 3-dimensional *work groups*. Work items in a work group can synchronize with each other and can read/write from a shared memory space called *local memory*, which is intended as a small programmer managed cache. Work items from different work groups cannot synchronize and cannot share local memory.

The kernel functions operate upon *buffer* objects created by the CPU through the OpenCL API. For devices such as discrete GPUs, buffers are typically allocated in the GPU's onboard memory. The application then needs to copy the input data from the system RAM to the allocated memory objects. Similarly, results from the kernel computations also need to be copied back from buffer objects to system RAM.

We use LLVM for implementing our CPU backend. LLVM is a compilation toolkit with many parts. LLVM's input format is a typed SSA representation called LLVM IR. LLVM can be used either as an analysis and transformation framework on this IR, and/or as a code generation toolkit for either static or just-in-time compilation. LLVM has been used in many academic and commercial projects in a variety of different contexts. For example, LLVM is used by Apple in many different products including their Xcode toolchain for iOS devices.[3] LLVM has also been used for code generation by many OpenCL implementations, such as those from AMD, Apple, Intel and Nvidia. We chose LLVM as the CPU backend due to its maturity, clean design and portability.

5 Implementation of Velociraptor

In order to automatically generate high quality code, Velociraptor has two important components: (1) a compiler framework for analysis, transformation and code generation; and (2) a runtime library designed to manage GPU resources and task dispatch.

[3] `http://llvm.org/Users.html`

5.1 Compiler framework

The challenge in the design of Velociraptor was to identify and implement the important IR simplifications, and the key analyses and optimizations that need to be performed on this IR. In this section we first describe the overall design of the compiler framework and then discuss three important phases: alias analysis, array bounds-checks elimination and load vectorization.

5.1.1 Design

Velociraptor's compiler operates in multiple passes and the overall picture is shown in Figure 2. First, we simplify VRIR into a slightly cleaner form, where we convert implicit array allocations in operators such as matrix multiplication into explicit allocations. This form enables simpler implementation of some flow analysis, and in the future may permit optimizations such as moving memory allocations outside of loops. Next, we perform some standard analysis such as live variables and reaching definitions 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. Details of the alias analysis are given in Section 5.1.2. Next, we perform array bounds-check elimination according to algorithm in Section 5.1.3. Array bounds-checks are particularly expensive on GPUs which are often poor at control flow and also impedes efficient code generation, particularly on some VLIW-based GPUs.

Next, for GPU sections, we perform *load vectorization*, an optimization where we attempt to merge loads into a single packed load. This optimization provides better effective memory load bandwidth on some GPUs, particularly AMD GPUs, and is detailed in Section 5.1.4. Finally, we have implemented code generators for CPUs and GPUs and these generate LLVM and OpenCL respectively. The GPU sections are replaced by a sequence of calls to VRRuntime that calls the generated OpenCL kernels for user-written loops and our OpenCL library function routines for standard VRIR library functions.

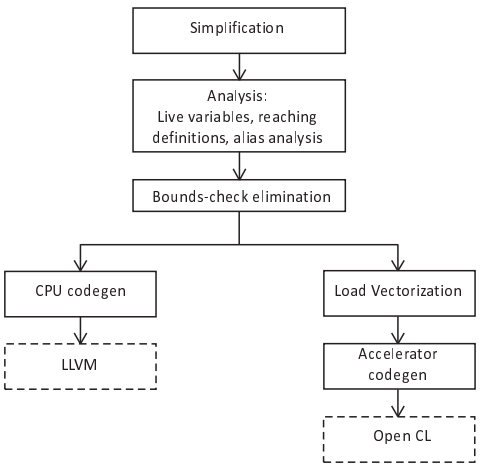

Figure 2: Compiler analysis, transformation and code generation infrastructure provided by Velociraptor

5.1.2 Memory aliasing and alias analysis

In VRIR, it is possible for two arrays to be views on the same underlying data. It can either simply be a case of two VRIR array references pointing to the same array, or it may be two different views (with different strides or number of dimensions) on the same underlying linear buffer. This was done to accommodate systems like NumPy, where many operations are defined by-reference rather than by-copy and memory aliasing is common. However, aliasing in VRIR is simpler than aliasing in languages like C/C++. Unlike C pointers, VRIR array structures carry information about the size of dimensions of the memory region being pointed to. Further, the VRIR type system does not contain pointers and arbitrary integers cannot be converted to/from pointers.

We have implemented a flow-insensitive intraprocedural alias analysis that maintains a may-point-to map from array variables to allocation sites. Our analysis is a forward analysis and propagates information from function parameters to the body. One of the limitations in Velociraptor is that as an embedded compiler, it does not have access to the call chain of the entire application and thus does not know all the call-sites of VRIR functions. Thus, generally Velociraptor makes a conservation assumption that all array parameters may-point-to the same allocation site. However, we have provided optional annotations that allow the host compiler to specify a may-point-to map for function parameters. For example, in MATLAB language arrays are not aliased and a MATLAB-based host can pass that information to Velociraptor.

5.1.3 Array bounds-check elimination

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, techniques.

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 basic approach is that we identify a subset S of simple analyzable subscript expressions inside the loop. We attempt to move the tests out of the loop and generate a test T outside the loop to test if all of the subscripts in S are within bounds. We generate a new version L' of the loop with the bounds-check for subscripts in S disabled. If the generated test T determines that all subscripts in S are within bounds, then L' is executed otherwise the original loop L is executed.

The second approach belongs to the class of techniques called loop versioning and we have chosen an implementation where we generate two versions: L and L'. . Theoretically, given the set S, we can have $2^{|S|}$ cases at runtime where each bounds is determined to be either safe or potentially out-of-bounds. Thus, theoretically, we could have generated $2^{|S|}$ versions of the loop corresponding to each case. However, Velociraptor is a JIT compiler and compilation time is an issue. Therefore, we have chosen the two-version approach because each generated loop adds to the compilation cost for backend phases, and would have been particularly expensive for the OpenCL backend. Secondly, most programs in production are not expected to hit array-bounds violations and thus all the tests are expected to pass. Thus, a finer grained approach will not likely yield any benefit in production code.

Now we describe how we identify the set S and how we generate the test T. We implemented this approach in Velociraptor for nested loops where the loop-bounds of inner loops are affine expressions of loop indices of outer loops. Array accesses in VRIR can be multidimensional with a subscript expression in each dimension. Without optimizations, we need to generate a bounds-check for each subscript expression. Consider an array reference $A[..., c_0 * i + c_1, ...]$. Let i be one of the loop indices and c_0 and c_1 be constants. The lower bound and upper bound of the subscript expression can be computed outside the loop as an integer maximization problem using an integer linear programming (ILP) solver. If the size of the array A does not change within L, then the bounds-check can be performed outside the loop. Velociraptor collects all affine subscript expressions for which the bounds-check can be performed outside the loop into the set S. Velociraptor then generates code to compute the lower and upper bound of all subscripts in S using calls to an ILP solver. The test T is simply a conjunction of the success of bounds checks of all the subscripts in S.

Our solution requires solution of an ILP, which can be an expensive operation if not controlled. The cost of solving ILPs increases with the number of variables and number of constraints. The variables in our ILP problems are loop variables in the original program. We control the cost of the ILP by limiting the number of variables in the problem. Currently, we have limited the number to five variables, which allows five-level loop nests, which is sufficient for most programs. If a deeper loop-nest is encountered, then the optimization is still applicable to an innermost five loops. For example, if a seven-level loop-nest is encountered, then the tests in the innermost loop can still be moved five levels up. With our implemented approach, the cost of solving the ILP added negligible runtime overhead.

5.1.4 Load vectorization

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. Consider an array A with 64-bit floating-point elements and two array read operations inside a loop: $A[i]$ and $A[i+1]$. For contiguous arrays, we attempt to merge the two loads into a single 128-bit load and we call the transformation *load vectorization*. For vectorized loads, Velociraptor generates the OpenCL builtin function *vload*, which can be used for unaligned loads, and thus Velociraptor does not need to verify that the generated 128-bit accesses happen on 128-bit boundaries.

Load vectorization is performed as follows. First, we identify all row-major or column-major arrays A such that A is read inside the loop, but A or any array potentially aliased with A is not written inside the loop. Next, the compiler collects all references with affine index expressions inside the loop for which bounds checks have been eliminated. We compute the distance for all possible pairs of collected array references and identify pairs with a distance of one (i.e. consecutive accesses). We merge and replace identified consecutive accesses with a single load.

5.2 VRRuntime

In addition to the optimizations, for good performance Velociraptor requires a well-designed runtime system that exposes the correct OpenCL abstractions and implements the required services in an effective and efficient manner.

VRRuntime offers a number of services. First, VRRuntime presents a simplified API to Veloci-

raptor. OpenCL is a very verbose API, and the full functionality of the API does not need to be exposed to Velociraptor. VRRuntime offers a simplified wrapper, such as simplifying the task of compiling OpenCL code to device binary code through the driver.

Second, VRRuntime presents a single task-queue abstraction to Velociraptor. Tasks that can be queued in the queue include predefined library routines, OpenCL kernels generated from user-code and data synchronization tasks. VRRuntime runs a task dispatcher in a background thread that continuously dispatches tasks from the queue to the GPU.

Finally, VRRuntime takes care of all the data transfers required between the CPU and GPU. Velociraptor and VRRuntime operate on a concept of ownership of variables. The idea is that a variable can be owned by the CPU code, or by VRRuntime, but not both at the same time. At the program point where a task is enqueued, Velociraptor inserts code to hand over the ownership of operand variables to VRRuntime. While VRRuntime has ownership of a variable, it is free to perform data transfers to/from the GPU required for completing any enqueued tasks. When the CPU needs to regain the ownership of a variable, it issues a call to VRRuntime to release the ownership of the variable. VRRuntime then ensures that any pending tasks related to the variable are completed, performs any necessary data transfers, and then releases ownership of the variable.

Thus, VRRuntime offers a number of services designed to hide the complexity of GPU book-keeping from the code generator.

VRRuntime has several important optimizations that allow for good performance:

Asynchronous implementation: VRRuntime has been been architected to allow CPU and GPU to work in parallel at the same time, with as little synchronization as possible. Enqueuing a task in VRRuntime is a non-blocking operation. When a task is enqueued in VRRuntime, a future token is returned. The thread that enqueued the operation can then continue to do other useful work till the program point where it requires the enqueued task to be finished. At that point, the enqueuing thread can wait upon the future token. The task queue is processed by a separate thread.

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 standard copy-on-write optimization and has been implemented in VRRuntime.

Data transfers in parallel with GPU computation: VRRuntime can do some data transfers at the same time as a computation may be executing on the GPU. VRRuntime dispatches calls to the GPU in a non-blocking fashion by using OpenCL's robust event facility. When a kernel call is dispatched, instead of waiting upon the completion of the call, VRRuntime examines the next task and initiates the data transfers if possible. Thus, the data transfer overheads can be somewhat mitigated by overlapping them with computation.

6 Two case studies of using Velociraptor: Python and McVM

To aid in the design of Velociraptor, and to demonstrate two different uses of the framework, we used Velociraptor with two very diverse projects. The first was building a proof-of-concept Python compiler for core array-based computations, and the second was extending an existing VM/JIT for MATLAB to handle GPU computations. We summarize these two projects in the next two subsections.

6.1 Proof-of-concept Python compiler

We have written a proof-of-concept compiler for a numeric subset of Python, including the NumPy library, that integrates with the standard Python [**?**] interpreter. We handle scalars and NumPy arrays but do not handle data structures like dictionaries, tuples or user-defined classes. The idea is that only a few functions in a Python program will be compiled, while the rest of the program will be interpreted by the Python interpreter. We currently require the programmer to define the type signature of the function through a decorator we have defined, and then infer 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 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 provided a construct to annotate blocks of code to be executed on the GPU. Such blocks are indicated by delimiting them through a pair of "magic" function calls called *gpu_begin*() and *gpu_end*(). The compiler removes such calls and instead converts them into a statement list and marks the list as GPU-executable.

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 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 to VRIR in a separate pass. We wrote a little glue code in C++ to wrap the function pointers returned by Velociraptor into Python functions. We have also 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 were able to develop this compiler quickly by using Velociraptor. The frontend of the compiler is about about one-tenth the size of the combined size of Velociraptor and RaijinCL, which shows that Velociraptor allows compiler writers to quickly build compilers by providing considerable inbuilt functionality. In this case study, our compiler was implemented in about 4000 lines of C++ andd 300 lines of Python code.

6.2 Extending McVM

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.

For both of these constructs, we utilized Velociraptor to generate the code while using McVM's existing code generation facilities for rest of the program. Our extensions require new passes in McVM and these passes are run after McVM has performed type inference. Our implementation 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 compiler 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, values 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 calling code for correctness.

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 analysis 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, integrating Velociraptor required only about 5000 lines of C++ code.

7 Experimental results

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.

7.1 Python performance

Although the main point of this paper is the design and implementation 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 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: 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. Thus, we tested a total of 8 variations (4 CPU versions and 4 accelerated versions). 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 I as speedups over PyPy.

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	3.3	4.1	9.76	11.5	8.3	9.6	8.3	9.6
	M2	3.1	3.5	11.0	11.9	8.7	9.9	8.7	9.9
julia	M1	35.7	36.3	213.8	215.1	1518	1497	1498	1522
	M2	36.9	37.8	170.6	172	740.3	764.5	735.0	773.1
growcut	M1	13.2	17.8	42.8	58.6	114.7	168.6	116.7	170.1
	M2	11.6	15.3	37.3	49.9	91.3	138.8	90.9	137.4
rosen-der	M1	15.3	18.4	31.9	39.2	32.5	34.2	32.9	34.3
	M2	12.9	16.0	25.0	29.7	25.8	27.4	25.9	27.3

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

Some of our observations are as follows:

1. Serial unoptimized CPU code generated by Velociraptor is between 3 to 37 times faster than PyPy depending upon the benchmark.

2. Bounds-check optimization provides between 1% to 35% performance improvement on CPUs 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.

3. Parallel CPU code generated by Velociraptor was generally two to six times faster than serial CPU code generated by Velociraptor. Thus, Velociraptor easily allows the addition of multi-core backends. We used quad-core processors and more than four times speedup can be explained by the fact the CPUs are hyperthreaded and run eight threads.

4. Optimized GPU code generated by Velociraptor can be up to seven times faster than generated optimized parallel CPU code on some benchmarks. However, data transfer overheads in

benchmarks such as rosen-der and arc-distance can also lead to slowdowns. Thus, GPU-offload may not be suitable for all problems.

5. Our compiler optimizations yielded between 1% and 52% performance improvement over baseline GPU code generated by Velociraptor.

7.2 McVM performance

For McVM, we looked at benchmarks used by previous projects such as McVM and MEGHA [**?**] and chose four parallelizable benchmarks and added GPU annotations. The benchmarks in this section are longer and more complex, and 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. For multi-core performance with Velociraptor, we test two variations: Velociraptor optimizations enabled and disabled. For accelerated performance, we tested four versions as detailed in the previous section.

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	3.23	3.23	3.23	3.23
	M2	1.0	0.99	0.99	2.37	2.35	2.52	2.50
nb1d	M1	1.58	1.51	1.51	2.92	2.93	3.01	3.01
	M2	1.30	1.32	1.32	3.36	3.38	3.62	3.61
nb3d	M1	0.2	0.52	0.52	1.64	1.64	1.72	1.72
	M2	0.22	0.71	0.71	2.42	2.40	2.63	2.63
fdtd	M1	0.15	0.58	0.58	1.92	1.92	1.97	1.97
	M2	0.13	0.56	0.56	2.05	2.06	2.17	2.20

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

We used Mathworks MATLAB R2013a 64-bit, which is a commercial JIT compiled implementation, as the baseline and report performance in Table II as speedups over MATLAB.Some observations are as follows:

1. 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.

2. Compiler optimizations do not play a role in these benchmarks because the benchmark use vector-style array operators instead of explicit loops, whereas our compiler optimizations are targeted at explicit for-loops.

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 [**?**] and PyOpenCL [**?**]. 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 [**?**] 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 [**?**] was a dynamic compiler that compiled annotated Python loops to AMD's deprecated CAL API. Theano [**?**] is a Python library that compiles expression trees to CUDA. In addition to GPU code generation, it also includes features like symbolic differentiation. Parakeet [**?**] is a compiler that takes as input annotated Python code and generates CUDA for GPUs and LLVM for CPUs. MEGHA[**?**] 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 [**?**] 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 [**?**] 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. Further, 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 accelerators is to compile bytecode of high-level VMs to accelerators. This is complementary to our approach and there are two recent examples of this approach. AMD's Aparapi [**?**] 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 arrays are allowed with Java's simple indexing operators. Dandelion [**?**] is a LINQ style extension to .net that cross-compiles .net bytecode to CUDA. The programming style for LINQ is quite different than the loop and array-operator based approach we provide in our work.

To our knowledge, there has not been prior work on embeddable or reusable compilers for accelerators. The only work in this area that we are aware of is NOVA [**?**]. 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 system in two systems.

To summarize, in contrast to previous research, our system is an embeddable and reusable compiler targeting both CPUs and GPUs that is not limited to one programming language or runtime. 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 system is meant to complement and enable, not compete, with other compiler researchers. Had our tools (Velociraptor code generator, runtime and RaijinCL library) existed earlier, most of the previously mentioned systems could have used it while focusing their time elsewhere. For example, MEGHA's automatic identification of accelerator sections, the type inference work of Parakeet or Theano's work on symbolic facilities are all complementary to our work. We believe availability of our toolkit will free other researchers to spend their time on many other open problems in programming language design and implementation rather than spend time writing backends or design accelerator runtimes.

9 Conclusions and Future Work

In this paper we have presented a toolkit for enabling compiler writers to effectively generate CPU/GPU code for a variety of array-based languages. Our first key design decision was to provide a clean and high-level intermediate representation, VRIR, which can be used to express a wide variety of core array-based computations and many different array accessing modes.

The second key component is the Velociraptor code generation tool and it association runtime library. 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. The generated code and associated runtime library takes care of many issues, including the communication between the CPU and GPU, capturing errors, reducing communication costs, and overlapping CPU and GPU computations.

To demonstrate the toolkit, we used it in two very different projects. The first is a proof-of-concept compiler for Python, where we used both the CPU and GPU code generation capabilities of Velociraptor. This showed that the toolkit could handle array-based computations from Python, and that very significant performance improvements were possible. The second project was using the toolkit to extend a MATLAB JIT, from McVM, to handle GPU computations. This showed that the toolkit was useful in extending the functionality of a pre-existing compiler, and that the VRIR also captures appropriate MATLAB array operations and indexing modes.

Now that the toolkit is established and we have two prototype applications of the toolkit, we plan to continue working on refining those prototypes, and adding further optimizations and transformations to the generated code with appropriate performance studies with a broad range of CPU-GPU hybrid systems and on a much larger benchmark set.

The toolkit will be made publicly available, and it is our hope that other compiler groups will use the toolkit and that we can further refine and add to its functionality based on those experiences. Such applications could include adding functionality to existing projects, such as compiling R[**?**], or for implementing solutions for other domain-specific languages which have an array-based core.

References

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

[2] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *SciPy 2010*, June 2010.

[3] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *CoRR*, abs/1209.5145, 2012.

[4] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. In *PPOPP 2011*, pages 47–56, 2011.

[5] Maxime Chevalier-Boisvert, Laurie Hendren, and Clark Verbrugge. Optimizing MATLAB through just-in-time specialization. In *CC 2010*, pages 46–65, 2010.

[6] Alexander Collins, Dominik Grewe, Vinod Grover, Sean Lee, and Adriana Susnea. NOVA : A functional language for data parallelism. Technical report, Nvidia Research, 2013.

[7] Rahul Garg and José Nelson Amaral. Compiling Python to a hybrid execution environment. In *GPGPU 2010*, pages 19–30, 2010.

[8] Andreas Klöckner. Pycuda. `http://mathema.tician.de/software/pycuda`.

[9] Andreas Klöckner. Pyopencl web page. `http://mathema.tician.de/software/pyopencl`.

[10] Kronos.org. The OpenCL Specification. `http://www.khronos.org/opencl`.

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

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

[13] Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. Evaluating the Design of the R Language - Objects and Functions for Data Analysis. In *ECOOP 2012*, pages 104–131, 2012.

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

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

[16] Gallagher Pryor, Brett Lucey, Sandeep Maddipatla, Chris McClanahan, John Melonakos, Vishwanath Venugopalakrishnan, Krunal Patel, Pavan Yalamanchili, and James Malcolm. High-level GPU computing with Jacket for MATLAB and C/C++. *Proceedings of SPIE (online)*, 8060(806005), 2011.

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

[18] R-project.org. The R Project for Statistical Computing. `http://www.r-project.org`.

[19] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. In *SOSP'13: The 24th ACM Symposium on Operating Systems Principles*, 2013.

[20] Alex Rubinsteyn, Eric Hielscher, Nathaniel Weinman, and Dennis Shasha. Parakeet: A just-in-time parallel accelerator for Python. In *HotPar 12*, 2012.

[21] SciPy.org. NumPy: Scientific Computing Tools for Python. `http://numpy.scipy.org/`.

[22] Dag Sverre Seljebotn. Fast numerical computations with Cython. In Gaël Varoquaux, Stéfan van der Walt, and Jarrod Millman, editors, *Proceedings of the 8th Python in Science Conference*, pages 15 – 22, Pasadena, CA USA, 2009.

[23] X10.org. X10: Performance and Productivity at Scale. `http://X10-lan.org`.

Velociraptor provides an optimizing compiler toolkit for generating CPU and GPU code and also provides a smart runtime system to manage the GPU. An important contribution of the thesis is a new dynamic compilation technique called region specialization that is particularly designed for numerical programs. Region specialization first performs region detection analysis, a novel compiler analysis which identifies regions of code that may be interesting for the compiler to analyze, such as loops and library calls involving arrays. Region detection analysis also identifies parameters such as shapes We present Velociraptor: a portable compiler toolkit 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 specically designed for numeric computations, with rich support for arrays, plus support for high-level parallel and accelerator constructs. A compiler developer uses Velociraptor by generating VRIR for key parts of an input program. Velociraptor does the rest of the work by optimizing the VRIR code, and generating LLVM for CPUs and OpenCL for GPUs. Velociraptor also provides a Additionally, programming support tools such as debuggers and profilers are much more readily available on the CPU. However, we expect that many new and exciting algorithms will continue to be written for the GPU, as people learn to map their algorithms onto the stream programming model. The "Further Reading" section of this chapter includes references to some of the exciting things people are implementing on GPUs. These programs include ray tracing, photon mapping, fluid-flow solvers, cloth simulation, and global illumination calculations. Also, we expect the GPU to continue to evol