A compiler toolkit for array-based languages targeting CPU/GPU hybrid systems

Sable Technical Report No. sable-2012-3

Rahul Garg and Laurie Hendren

November 16, 2012

Contents

List of Figures

List of Tables

Abstract

This paper presents a compiler toolkit that addresses two important emerging challenges: (1) effectively compiling dynamic array-based languages such as MATLAB, Python and R; and (2) effectively utilizing a wide range of rapidly evolving hybrid CPU/GPU architectures.

The toolkit provides: a high-level IR specifically designed to express a wide range of array-based computations and indexing modes; Velociraptor, a CPU/GPU code generator and runtime library; and RaijinCL, a portable autotuning GPU library for key BLAS routines. A compiler developer uses the toolkit by generating VelociraptorIR for key parts of an input program, and using Velociraptor to automatically generate CPU/GPU code.

The toolkit leverages OpenCL and LLVM for GPU and CPU code generation respectively, and can thus be used for a wide variety of target architectures. To demonstrate different possible uses of the toolkit, the paper presents a proof-of-concept CPU/GPU Python compiler, and a GPU extension of a MATLAB JIT.

1 Introduction

Two trends in scientific computing have become mainstream. First, array-based languages such as MATLAB [11], Python/NumPy [23] and R [21] have become extremely popular for scientific and technical computing. Such languages provide concise syntax for expressing array computations and they include many built-in operations such as matrix multiplication, which are implemented as wrappers around efficient libraries such as BLAS. The second trend is that GPU (Graphics Processing Unit) computing has become popular, with thousands of scientific papers published utilizing GPUs. 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 compiler communities. Unfortunately, in many cases, compiler writers are forced to write their GPU code generators as well as GPU management runtimes from scratch, making programming language and compiler research substantially harder.

Compiling for CPU/GPU hybrid systems presents many special challenges. GPU architectures are evolving at a rapid pace, which means that a successful compiler needs to be able to target a wide variety of architectures, and be flexible enough to incorporate new GPUs as they are released. Further, for array-based languages, only offering a code generation toolkit is not enough. Any GPU-based implementation of such languages must provide GPU-optimized library operations such as matrix multiplication. Due to the diversity and fast pace of evolution of GPU architectures, optimized libraries may not be available for GPUs that a user or compiler writer wants to use. For example, Nvidia provides a CUDA BLAS for their GPUs but does not provide an optimized OpenCL BLAS. On the other hand, AMD provides an OpenCL BLAS for their GPUs but it has poor performance on Nvidia GPUs. Many new GPUs with OpenCL implementations are also expected to arrive soon, and it is not clear if the vendors will provide optimized BLAS for those GPUs either. Thus, compiler writers are often forced to choose between architectures simply due to lack of portable libraries.

To solve these challenges our toolkit supports: (1) a shared, reusable and portable code generation infrastructure which enables rapid development of mixed CPU/GPU compilers for a variety of array-based languages and for a variety of GPU architectures, and (2) an auto-tuning library that generates high-performance codes for important matrix operations that are tuned to each machine. The four main contributions of this paper are as follows:

Overall design and IR: Our toolkit is intended to be easy to adapt into existing compilers, or to target for new compilers. Thus, we have defined a clear interface between the main host CPU compiler and our CPU/GPU toolkit. A key ingredient in this design is our intermediate representation called VRIR that is designed to be flexible enough to express semantics of numerical programs from multiple array-oriented languages. The second major feature of VRIR is that it is designed to explicitly represent both CPU and GPU code sections.

Velociraptor code generation and runtime system: The central component of our toolkit is the Velociraptor infrastructure that takes as input VRIR and generates LLVM [10] code for CPU code segments and OpenCL [9] kernel code for GPU code segments. Building upon portable technologies such as LLVM and OpenCL allows Velociraptor to be portable across many different architectures. Velociraptor also has a runtime component VRRuntime that automatically handles GPU book-keeping, such as maintaining a GPU task queue and data transfers.

RaijinCL auto-tuning library: To tackle the challenge of providing portable libraries, our toolkit includes an auto-tuning library RaijinCL for matrix operations such as matrix multiplication. RaijinCL can generate high-performance code for matrix operations on many GPU architectures. This allows optimized library implementations to be quickly created for new architectures. We evaluate RaijinCL on three different GPUs and show it has competitive performance when compared to vendor-specific libraries. Velociraptor uses RaijinCL for implementing matrix operations, but RaijinCL may also be used independently by anyone requiring a portable numerical GPU library.

Example uses of Velociraptor and RaijinCL: To demonstrate our toolkit we have used it in two projects, one is a new proof-of-concept compiler for Python, and the other is an extension of an existing JIT compiler for MATLAB. The Python example takes as input annotated Python source and generates both CPU and GPU code using Velociraptor and RaijinCL. This demonstrates how a new compiler could leverage our toolkit to do both the CPU and GPU code generation. The MATLAB example uses our toolkit to extend McVM [4], a just-in-time compiler and interpreter for MATLAB language, to handle GPUs. This serves both as an illustration of using our toolkit to extend an existing compiler, as well as providing a useful contribution for users of McVM. By demonstrating both Python and MATLAB systems, we demonstrate that Velociraptor is not limited to a single source language.

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, and Section 6 describes our RaijinCL self-tuning library. We demonstrate our toolkit with two integration examples in Section 7. We finish with related work in Section 8 followed by conclusions and future work in Section 9.

2 Proposed compiler toolchain

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 Language Level Compiler (LLC) because it is responsible for converting the semantics of the language into lower-level representations. Implementing or extending such a LLC to handle a mixed CPU/GPU system has many new challenges:

1. Generating GPU code requires implementing new code generation backends, one for each target GPU, which is a considerable amount of work without a reusable toolkit.
2. Since CPU/GPU systems require coordination between the CPU and GPU, a new runtime component may also be required to manage the GPU book-keeping, ensuring coherence between CPU and GPU data at synchronization points and maintaining a GPU task queue which preferably needs to be asynchronous to allow proper use of all execution units in a machine.
3. When the LLC is for a high-level array language, generating CPU code also requires considerable effort since higher level operations such as array operators and complex indexing modes need to be simplified to a lower-level representation such as LLVM IR.
4. Many compiler writers want to target a wide variety of hardware platforms. This requires a backend built upon portable technologies. In addition, implementation of array-based languages also requires the implementation of fast numerical libraries. However, currently libraries offered by GPU vendors are designed only for their platforms.

Our Velociraptor toolkit, VRIR and runtime library, have been designed to address these challenges and to make it easy for compiler developers to add GPU functionality to compilers for array-based languages. Figure 1 demonstrates how our approach cleanly integrates with a LLC, with the proposed additions inside the overlay box on the right, and the conventional passes on the left.

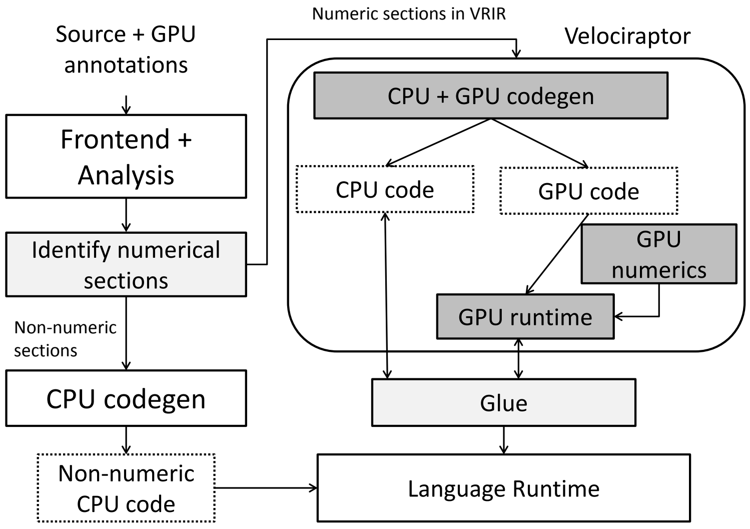


Figure 1: Possible design of a LLC utilizing Velociraptor

The key idea is that the LLC does minimal work - it must identify numerical parts of the program, such as floating-point computations using arrays, and the LLC must compile these parts to VRIR. The LLC must also indicate parallel-loops and potential GPU sections, and provide a small amount of glue code to provide the interface for the host representations of arrays and the host memory manager. Velociraptor takes over the major responsibility of performing CPU/GPU code generation, and the interaction between the CPU and GPU.

Non-numerical parts of the program, such as dealing with file IO, string operations and non-array data structures are handled by the LLC it its normal fashion.

Different LLCs can utilize Velociraptor for different purposes. An existing LLC with a mature CPU code generator may choose to utilize Velociraptor for only GPU code generation whereas a new LLC may choose to offload as much work as possible to Velociraptor, including generating CPU code for the numerical sections. We illustrate these two types of uses in Section 7.

Portability is a key concern for us and we chose to build upon portable technologies like OpenCL and LLVM, which are briefly covered in Section 4. Finally, we also provide a high-performance portable numerical library RaijinCL for avoiding the vendor lock-in created by vendor libraries.

3 Design of VRIR

In order to make Velociraptor both portable and useful for a variety of array-based languages, we have carefully designed its intermediate representation, VRIR Our objective was to create an IR which captures arrays and array-based computations in a way that maps well to the array-based source languages, but also provides enough structure and type information for effective code generation. Our objective is to cover the common cases of computations which map effectively to GPUs, and not to support a full general-purpose programming language. If a computation cannot be expressed in VRIR, then the host compiler simply compiles it normally, using the host LLC compiler's CPU-based code generator.

VRIR is a tree-based intermediate representation, with optional attributes for some node types. VRIR is designed primarily for array-based computations and supports many common array operators built directly into the representation. VRIR is typed and requires that the type of a variable should be fixed throughout the function, and should be statically known. The type of the result of all expressions occurring inside a function should also be statically known. It is the job of the host compiler to generate the VRIR, and the appropriate types. A JIT for a dynamically typed language will typically specialize the code based on run-time types, generating typed VRIR which is then compiled by our toolkit.

We have defined both an internal C++ representation, and an XML-based representation of VRIR. LLCs can build the XML representation of VRIR and pass that to Velociraptor, or alternatively LLCs can use the C++ constructors directly to build the VRIR trees.

3.1 Type System

The VRIR type system can be divided into four categories: scalars, arrays, domains and functions.

Scalar types provided are integer (32-bit and 64-bit, signed and unsigned), floating-point (32-bit and 64-bit), complex (currently 64-bit floating point components only) and boolean.

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. 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. For arrays of row- or column-major layouts, indices are converted into addresses using the regular rules for such layouts. Languages like MATLAB do not have strided views and do not need to provide space for storing strides. The value of array sizes and the values of array strides (if implemented) are not part of the type of the object, and are looked-up dynamically.

Domain types represent multidimensional strided rectangular domains. 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. Languages like MATLAB do not have an explicit domain type but some languages like X10 [26] do.

Function types specify the type signature of a function. VRIR functions have a fixed number of arguments and outputs. VRIR does not support first-class functions currently, so functions cannot be passed as first-class values.

3.2 Fundamental constructs

The top-level construct in VRIR is a *module*. A module consists of a sequence of functions. Functions consist of a type signature, symbol table, argument list and a statement list representing the body. Statement constructs provided include assignments, statement lists, expression statements, conditionals, for-loops, while-loops and parallel-for loops. Expression constructs provided include various arithmetic, comparison and logical operators, function calls (including standard math library calls), and array indexing operators.

3.3 GPU sections

VRIR aims to support CPU/GPU hybrid systems, so we allow for the case where some statements should execute on the CPU, and other statements on the GPU. We support this in VRIR as follows.

Any statement list can be marked as GPU-executable. Any matrix or vector library operations occurring in a GPU-executable statement list are attempted to be executed on the GPU. Any parallel loops occurring inside a GPU-executable statement list are also executed on the GPU (if possible) and must obey two conditions. First, there should not be any calls to user-defined functions inside such a parallel loop. Second, there should not be any memory allocation functions inside the parallel loop. Both these restrictions are made to allow the Velociraptor code generator to map the loop to an OpenCL kernel.

Statements other than library operations or parallel loops in the marked statement list are executed on the CPU. Velociraptor and VRRuntime manage all the data transfers between 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 statement list, the semantics are that all the variables are synchronized back to the CPU, though some optimizations are performed by Velociraptor and VRRuntime to eliminate unneeded transfers.

3.4 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 that is indexed using d indices. VRIR has the following indexing modes:

1. Let $d == n$, and 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).
2. Let $d == n$, and $k < n$ indices are one-dimensional domains. If an index is a domain, then a slice of the array in the corresponding domain is selected instead of an individual element in that dimension. Unlike MATLAB, array slices in VRIR do not create copies, but instead create new strided views of the same data. LLCs implementing languages like MATLAB must insert explicit copy instructions where necessary.
3. If $d < n$, and all of them are either integer or a domain, and the optional node attribute *flattened indexing* is either unspecified or false, then the rest $d - m$ indices are implicitly filled-in to be slices spanning the size of the corresponding dimension.
4. If $d < n$, and all of them are either integer or a domain, and the optional node attribute *flattened indexing* is true, then 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. Flattened indexing mode is inspired from MATLAB, while the non-flattened mode is required for NumPy.
5. An optional attribute called *negative indexing* affects all of the above rules. If the size of the k-th dimension of the array is u_k, then the index in the dimension must generally be in the set $[0, u_k - 1]$. However, if the negative indexing attribute of the node is set to true, then slightly different semantics is followed. If an index i_k is less than zero, then the index is converted into the actual index $u_k + i_k$. For example, the index -1 represents the last element in the dimension $u_k - 1$. Negative indexing exists in systems such as NumPy.
6. 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. However, unlike other indexing modes, indexing using arrays returns a copy of the data.

3.5 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, as well as operators for sum and product of the elements of arrays are also provided.

3.6 Error reporting

GPU computations 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. If an error occurs, then a cleanup function specified by the LLC 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 LLC and may not be required by all LLCs. 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 LLC 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.

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. The cleanup function, if any, will be inserted after the parallel-for loop or GPU section. 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.

Some LLCs may want to disable these additional safety checks. For example, the language semantics or user annotations may allow for unsafe indexing, or the LLC may perform transformations such as array bounds check elimination. Thus, we provide the ability for the LLC to disable error reporting at the module or function level through an optional binary attribute. Array bounds checks can also be disabled at the level of an individual array read or write node.

3.7 Memory management

Different LLCs have different memory management schemes. VRIR and Velociraptor provide automatic cleanup of scalar variables, but for array variables we allow the LLC to use the appropriate memory management scheme. LLCs 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. LLCs which require explicit memory management can insert explicit allocation and deallocation instructions.
2. LLCs which require reference counting can insert instructions to increase or decrease reference counts.
3. LLCs 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 [9] 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. However, we describe OpenCL only in the context of GPUs here. In OpenCL, the host (the CPU) controls one or more devices (GPUs in the context of this paper). 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 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 host 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 host 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. 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

Velociraptor is designed as a library to be used as one component of a compiler. Velociraptor operates in multiple passes, it first performs some simple analysis such as live variable analysis on VRIR and then lowers VRIR into a lower-level form simplifying constructs such as complex indexing operators, and then finally generates code for the CPU and GPU backends. For the CPU backend, Velociraptor currently generates multi-threaded LLVM code while for GPU backend, Velociraptor generates OpenCL. Calls to CPU or GPU library functions are inserted for array operators or library routines. For the GPU backend, a lot of the complexity has been offloaded to VRRuntime instead of dealing with GPU management directly in Velociraptor. Velociraptor is responsible for generating OpenCL code for any GPU parallel-loops, but all the GPU book-keeping is done by VRRuntime which exposes a very simple API to Velociraptor.

The code generation strategy is quite straightforward but two key components are explained in the next subsections: integration with LLCs and the implementation of VRRuntime.

5.1 Integration with LLCs

Velociraptor is not a virtual machine or a runtime execution environment on its own. Rather, Velociraptor is meant to be part of such a system, where Velociraptor generates code and deals with the runtime system of the LLC on the CPU and with VRRuntime on the GPU.

Since Velociraptor is intended to be used for a wide variety of array-based LLC's, Velociraptor itself does not implement details like the data structure used to represent n-dimensional array objects at runtime. Instead, it is designed to interface with the host LLC's data structures. For sake of brevity, we will call the data structure used by the LLC to implement n-dimensional array objects as RtArray while the word "array" will be used to refer to the regular linear array data-type. Velociraptor simply assumes that the RtArray is implemented by the LLC as a structure with at least the following components.

First, the RtArray must contain a pointer to a contiguous linear buffer of memory which stores the actual data of the array. Second, the RtArray must contain a pointer to a buffer of n integers containing the sizes of the n dimensions. If the LLC implements stride-based layouts, then the third necessary component is a pointer to a buffer of n integers containing the strides of the array. A LLCś implementation of RtArray may contain more components, such as object headers containing reference counts. Velociraptor does not care about what other data is stored inside the array data-structure. Instead, LLC must simply tell Velociraptor the offsets of the three pointers described above from the start of the RtArray data-structure. Data is passed in and out of Velociraptor generated code through pointers to RtArray data-structures.

Velociraptor does not know the full layout of the RtArray data-structure, nor does it know the memory management details of the LLC. Thus, additional support is required from the LLC. First, LLCs must provide an implementation of a memory-allocation API specified by Velociraptor. If additional functions such as reference counting are required by the LLC, then it must supply the implementation of those APIs as well.

With this background, the interface exposed by Velociraptor can be explained. Velociraptor takes as input a VRIR module and returns an array of function pointers corresponding to the compiled versions of these functions. Each of these function pointers has a generic signature, taking as input an array of void pointers and returning an error code. The length of the vector of void pointers is defined to be the sum of the the number of input and output arguments of the corresponding function in VRIR. Each entry may either be a pointer to an array data-structure of the LLC, or a pointer to a scalar value. Such a generic interface does add overhead of packing and unpacking arguments in each function call, but allows easier integration into LLCs with a simple and uniform interface.

5.2 Memory aliasing

Before explaining the internals of VRRuntime, we need to explain memory aliasing within the context of VRIR. An important detail to be understood within VRIR is that 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 RtArray, 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, there are fundamental differences between aliasing within the context of VRIR and aliasing in languages like C/C++. Unlike C pointers, RtArray structures carry information about the size of dimensions of the memory region being pointed to. Further, the VRIR type system does not contain pointers, nor does it allow array-of-arrays. Thus, there is no possibility of pointer-to-pointer, or of arbitrary integers being converted to/from pointers.

5.3 VRRuntime

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

Task dispatcher is normally asleep, but wakes up when it is notified of an event such as completion of a previous task, or enqueuing of a new task. When the task dispatcher wakes up, it determines if a task needs to be dispatches. If a task needs to be dispatched, it pops the top of the queue and dispatches the required actions to an OpenCL queue object associated with the GPU being used. Once it dispatches the top task, it also peeks at the new top element, the task dispatcher also dispatches any required data transfers to the OpenCL queue.

The task dispatch system is a multithreaded event-driven system. Instead of building our own event dispatch system, we used the open-source Qt toolkit's [18] signal and slot mechanism. Our system needs to handle two type of events. First, we needed to handle events generated by Velociraptor (such as enqueing tasks). Second, we needed to handle events generated by OpenCL driver, such as notification of completion of a GPU kernel. We created Qt signals for both of these events. For the Velociraptor events, the implementation was straightforward as events are being both generated and consumed from our framework. For the OpenCL events, we registered callbacks with OpenCL driver to be notified of OpenCL events. These callbacks notify a wrapper object, which then generates a Qt signal and wakes up the task dispatcher. The event system has to run in a multi-threaded environment. So we used Qt's QueuedConnection connections, which enqueue events at the receiver in a thread-safe manner.

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.

VRRuntime maintains several book-keeping data structures. First, for each variable, we maintain a reference count. This is not an absolute reference count, but instead it is a reference count difference from the beginning of the GPU section. Second, for each variable, we maintain information about the region of CPU memory it is pointing to such as the start and end points. This is first computed whenever a variable is handed over to VRRuntime by insepcting the associated RtArray data structure object. However, during the execution of the GPU section, the variable may need to be reallocated if required by the program, and this information is kept up-to-date. Third, we maintain information about the location of the array associated with the variable on the GPU. It is simply a pointer to the OpenCL buffer object, an index for the starting point and the span which must be equal to the span of the variable in the CPU memory. VRRuntime ensures that if variables are pointing to aliased regions of memory on the CPU, then they are also all pointing to the same GPU buffer with equivalent aliasing properties. The pointers to GPU buffers are maintained in a bi-associative table. Thus, for each GPU buffer, we also know the number of variables that are pointing to it. If the number of variables pointing to a buffer drops to zero, then the buffer can be deallocated. Finally, for each GPU buffer, we maintain a dirty bit indicating whether the buffer was potentially written, but was not synchronized with the CPU, during any GPU task dispatched so far. This information is used by VRRuntime to determine if a variable pointing to a buffer needs to be copied back when Velociraptor requests the ownership of the variable to be released.

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

5.4 VRRuntime performance optimizations

VRRuntime has four 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.

Reducing data transfers: For each task, VRRuntime knows which variables were potentially written. Consider the case of a variable V that has been copied to the GPU but which was not potentially written in any kernel or library function. In this case, if the CPU requests reading the variable V, then the data does not need to be copied back since the copy on the CPU is already the freshest version of the variable. Thus, transfer of some data can be avoided.

Copy-on-write optimizations: Consider the case where variable A is explicitly cloned and assigned to variable B. However, if variable A and B are never written-to after the operation, then the copy operation is not necessary. 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 Design and implementation of RaijinCL

Good performance for array-based computations requires not only good code generation for GPUs and CPUs, but also an efficient GPU-based library for the key numerical kernels such as BLAS (Basic Linear Algebra Subprograms) [15]. For the purposes of Velociraptor, it would be ideal to have a high-performance portable library that works well with OpenCL (the target code for our Velociraptor's GPU code generator).

Thus, as part of the Velociraptor project we have developed RaijinCL, an implementation of a subset of the BLAS API (modified to work with OpenCL), as well as useful functions such as transcendental functions operating element-wise on matrix objects. The implemented functions are GEMM (general matrix multiplication), GEMV (general matrix vector multiplication), element-wise matrix addition and multiplication, element-wise negation, scalar multiplication, matrix transpose, reduction functions for sum and product and element-wise math library functions such as logarithms and trigonometric functions. RaijinCL is a key component of our Velociraptor toolkit, but it is also a free-standing library that could be used by other applications.

Because our toolkit is intended for a wide range of hybrid CPU/GPU systems, it was important to us that it was both portable (hence the choice of of OpenCL) and performant for each different architecture. Thus, we designed RaijinCL as an autotuned OpenCL code generator. Upon installation on a specific CPU/GPU system, the autotuning system generates and tests hundreds of different versions of key routines such as GEMM (general matrix multiplication) and caches the best version. Autotuning parameters for GEMM includes tile sizes, OpenCL work group sizes, loop unroll factors, use of packed short-vector types (such as 128-bit float4 types), whether or not to use local memory as well as order of some operations (such as interleaving loads and arithmetic operations) where applicable.

For operations like GEMM, RaijinCL can handle all relevant BLAS parameters and variations. For example, both row major and column major layouts are supported, as are both transposed and untransposed inputs.

6.1 Performance

The main point of developing RaijinCL was to have a portable and OpenCL-based library which could be used with our OpenCL-based toolkit. Thus, we wanted to investigate if such a auto-tuned portable library could compete with highly-tuned vendor-specific libraries. Our experiments were performed on three different machines containing GPUs from three different families. Descriptions of the machines are as follows:

1. Machine M1: Core i7 3820, 2x Radeon 7970, 8GB DDR3, Ubuntu 12.04, Catalyst 12.4
2. Machine M2: AMD Phenom II X4 925, AMD Radeon 5850, Ubuntu 11.04, AMD Catalyst 12.10
3. Machine M3: Core i7 920, Tesla C2050, GTX 480, 6GB DDR3, Ubuntu 12.04, Nvidia driver version 280.13.

Machine M1 and M3 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. Machine M2 contained a single GPU and it performed both display and computation duties.

We compare the performance of RaijinCL, focusing on the GEMM (matrix multiply) routines as those are the most performance critical functions in the library. We measured the performance of RaijinCL on all three machines. We compare the performance against the following vendor libraries. AMD's OpenCL BLAS is a proprietary OpenCL library provided by AMD and has been tuned for their GPUs. However, as it is using the OpenCL API, the library runs on any OpenCL compliant device and thus we report the performance of AMD BLAS on all three machines as well. AMD's BLAS includes a tuner that can be run on the user machine, and we ran the tuner on all machines. The algorithms employed by the AMD tuner are not known. Nvidia does not provide an OpenCL BLAS. Nvidia only provides a BLAS called CUBLAS for their proprietary CUDA API and thus it only runs on Nvidia GPUs. Thus, we provide the results for CUBLAS only on machine M3. We used AMD BLAS v1.6 and CUBLAS v4 respectively.

We first compare the performance on DGEMM (double-precision matrix multiply). Performance on Machine M1, which contains the AMD Radeon 7970, is shown in Figure 2(a). We found that the kernel generated by RaijinCL is competitive with the vendor (AMD) BLAS, at least for large sizes. Results on Machine M2, which contains the older Radeon 5850 GPU, are shown in Figure 2(b). We found that RaijinCL found kernels competitive with AMD's BLAS on this machine as well. Results on Machine M3, which contains an Nvidia GPU, are shown in Figure 2(c). RaijinCL is slower than CUBLAS (based on the proprietary CUDA API) but performs many times better than AMD's BLAS (based on OpenCL). Algorithms in AMD's BLAS appear to be written with only AMD GPUs in mind, and display very poor performance on Nvidia hardware. However, RaijinCL tests a variety of kernels, and chose a very different kernel on Nvidia GPU.

Next we examine ZGEMM (complex matrix-multiply) performance. We again find that RaijinCL is competitive with AMD's BLAS on AMD GPUs in Machines M1 and M2, as shown in Figure 3(a) and Figure 3(b) respectively. Results from Machine M3, with the Nvidia GPU, are shown in Figure 3(c). RaijinCL achieves more than 75% of the performance of CUBLAS, while AMD's BLAS struggles and is more than ten times slower than RaijinCL. This result again highlights the importance of RaijinCL's autotuner.

6.2 Summary

Based on the performance results, we were quite pleased with the ability of RaijinCL's autotuner to adapt to a variety of quite different GPUs. Thus, RaijinCL serves well as a key component of our toolkit, and can be used both with our Velociraptor code generator, or as a stand-alone GPU library.[1]

7 Integration into compilers

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 second was extending an existing VM/JIT for MATLAB to handle GPU computations. We summarize these two projects in the next two subsections.

7.1 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 [20] 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 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

[1]RaijinCL has its own web page, the URL can be made available.

generated in code compiled by Velociraptor back to the user.

We were able to develop this compiler pretty 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.

7.1.1 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 are adapted from work by Garg et al. [6] but we have changed annotations for type declarations and GPU sections to match the syntax offered by our compiler.

We tested our compiler on three different versions of each benchmark: a serial CPU version, a parallel CPU version and a GPU version. For each of these versions, we tested the code generation under two settings. In the first setting, we enabled out-of-bounds checks as well as NumPy's negative indexing checks. Our compiler supports such safety checks inside both CPU and GPU parallel sections. In the second setting, we disabled both of these checks to measure the overhead of these checks. We mark these two settings as Safe and Unsafe respectively. Thus, we compiled and ran six different combinations of each benchmark (three versions, two settings). Finally, we measured the the performance of the Python interpreter on these benchmarks as a baseline. The results are presented as speedups over the Python interpreter.

We ran our tests on Machines M1 and M3 described in Section 6 in Table II and Table I respectively.

Benchmark	CPU Serial		CPU Parallel		GPU	
	Safe	Unsafe	Safe	Unsafe	Safe	Unsafe
matrixadd	85	89.34	297	320	501	495
matrixmul	94	97	194	287	91	97
stencil	147	152	263	310	271	283
mandelbrot	67	71	193	184	205	210

Table I: Speedup of Velociraptor generated code over Python interpreter on machine with Nvidia GPU

Benchmark	CPU Serial		CPU Parallel		GPU	
	Safe	Unsafe	Safe	Unsafe	Safe	Unsafe
matrixadd	44.5	145	291	302	43.8	43.6
matrixmul	68.7	135	245	432	907	1008
stencil	108	225	292	883	108	112
mandelbrot	53	55	190	197	592	589

Table II: Speedup of Velociraptor generated code over Python interpreter on machine with AMD GPU

Some of our observations are as follows:

1. Serial CPU code generated by Velociraptor is up to two orders-of-magnitude faster than the Python interpreter, even with the safety and negative indexing checks enabled.
2. Parallel CPU code generated by Velociraptor was generally two to four times faster than serial CPU code generated by Velociraptor. Thus, Velociraptor easily allows the addition of multi-core backends.
3. GPU code generated by Velociraptor can be up to four times faster than generated parallel CPU code on some benchmarks. However, data transfer overheads in some benchmarks can also lead to slowdowns. Thus, GPU-offload may not be suitable for all problems.

7.2 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.

7.2.1 McVM example

Benchmarks presented in Python section were primarily loop-based benchmarks. Thus, in this section we present a different sort of benchmark that is typical of a style of MATLAB programming where the programmer uses high-level array operators. The benchmark named *closure* is a vectorized benchmark and most of the work is being done by a sequence of matrix multiplication calls. We annotated the computationally intensive part of the benchmark as a GPU section, which was then handled by Velociraptor. We measured the performance on two machines against the CPU implementations of McVM as well as MathWorks MATLAB. Execution times are shown in Table III.Velociraptor powered McVM-GPU implementation is able to beat out CPU implementations.

Machine	MATLAB	McVM	McVM (GPU)
M1	0.0652	0.09	0.03
M3	0.084	1.21	0.07

Table III: Execution time (in seconds) of a MATLAB benchmark using Velociraptor

As most of the work is performed by matrix multiplication, two factors decide the performance. First, RaijinCL's matrix multiplication routine is invoked. However, the matrix sizes in the benchmark are somewhat small (450) and GPUs are not efficiently utilized, but still perform faster than CPUs. Second, VRRuntime was able to avoid unnecessary data transfers of intermediate results that were not needed on the CPU. Without that optimization, performance would have suffered. This benchmark shows that there is potential for performance improvement for MATLAB benchmarks, including those written in a manner that uses high-level array operations.

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 [7] and PyOpenCL [8]. 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 [3] 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 [6] was a dynamic compiler that compiled annotated Python loops to AMD's deprecated CAL API. Theano [2] is a Python libraries that compiles expression trees using its own syntax to CUDA. In addition to GPU code generation, it also includes features like symbolic differentiation. Parakeet [22] is a compiler that takes as input annotated Python code and generates CUDA for GPUs and LLVM for CPUs. MEGHA[17] 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 [19] 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 [16] 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.

In contrast to all the work discussed in previous paragraph, our system is not limited to one programming language or runtime. We also believe we are the first system to carefully consider the variations of loop indexing schemes and array layout semantics of the programming languages and offer a complete solution for GPUs. Further, we have a defined mechanism for reporting errors such as out-of-bounds exceptions even for code running on the GPU, while previous efforts ignore this possibility. For example, jit4GPU does not support negative indexing scheme in Python and does not support out-of-bounds exceptions. Further, most of the previous compiler efforts (with the exception of Jacket) were tied to a single hardware vendor (in most cases Nvidia, and in one case AMD) while our system is portable to a wide variety of vendors due to the use of OpenCL as the backend. Previous systems depend upon a proprietary vendor numerical libraries, while our system is complemented by our own autotuning libraries. Many previous systems (except Jacket and MEGHA) have focused on single loop nests, while our system enables compilation of entire regions to GPU code. Thus, our system opens up possibilities such as better management of data transfers and better scheduling. Our GPU runtime is also architected to be more flexible than the previously mentioned systems because we enable CPU and GPU to work in parallel without blocking each other where possible. In some cases, our runtime can perform data transfers in parallel with computations, in contrast to most previous systems like MEGHA.

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 GPU region, the type inference work of Parakeet or Theano's work on symbolic facilities are all complementary to our work. Toolkits like LLVM have enabled a lot of activity in JIT compilers for many different languages because a reusable infrastructure has freed compiler writers from common aspects of JIT compilation. Similarly, we believe availability of our toolkit will also free other researchers to spend their time on many other open problems in programming language design and implementation rather than spend time writing GPU backends or design GPU runtimes. Unlike LLVM however, we have designed an IR specifically for array-based languages because we believe such languages are particularly suitable for data-parallel world of GPU computing.

We presented results for GEMM routines generated by RaijinCL. There has been considerable work in the field of writing GEMM for GPUs. Nakasato et al. [13] described a GEMM routine for AMD Radeon 5870 GPU written in AMD's low-level CAL IL assembler. However, CAL IL has since been deprecated and it is also not clear if their routines will work on other GPUs. Volkov et al. [24] presented hand-tuned GEMM routines for some Nvidia GPUs and their code is now included in CUBLAS. However, their algorithm appears to be too specific to Nvidia GPUs. Du et al.[5] present SGEMM and DGEMM (but not ZGEMM) routines on two GPUs: Tesla C2050 and a Radeon 5870. Their implementation for the AMD GPU was inspired by Nakasato's work and was handwritten. Their Nvidia version was partially autotuned (ported from CUDA version described by Nath et al. [14]) but some Nvidia-specific hand-tuned code was added. Our autotuner handles both architectures automatically and we have shown results on a third architecture (Radeon 7970) as well, all generated by the tuner. We also show results for ZGEMM while they did not cover ZGEMM. Weber et al. [25] present SGEMM and DGEMM routines tuned for Radeon 7970 GPU. Our autotuner generates code that performs somewhat better on SGEMM, and similar on DGEMM. However, they do not present results for ZGEMM. Also, it is our understanding that their routines are meant to be a prototype and do not handle matrices of all sizes. AMD's proprietary OpenCL BLAS also includes an autotuner, though the mechanisms are not known. The performance results presented for AMD BLAS reported in this work already include the tuning done by AMD's tuner, and our tuner is competitive as shown. For CPUs, autotuning is a well-known technique and ATLAS [1] is a well-known example of this approach.

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.

The final key component is the RaijinCL portable autotuning library. This allows compiler writers to easily generate an efficient library for specific CPU/GPU architectures. RaijinCL fits naturally into the overall toolkit, but can also be used as a free-standing library.

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[12], or for implementing solutions for other domain-specific languages which have an array-based core.

References

[1] Clint Whaley Antoine, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27:2001, 2000.

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

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

[5] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*, 38(8):391 – 407, 2012.

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

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

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

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

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

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

[12] 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.

[13] Naohito Nakasato. A fast GEMM implementation on the cypress GPU. *SIGMETRICS Perform. Eval. Rev.*, 38(4):50–55, March 2011.

[14] Rajib Nath, Stanimire Tomov, and Jack Dongarra. An improved Magma GEMM for Fermi graphics processing units. *International Journal of High Performance Computing Applications*, 24(4):511–515, November 2010.

[15] netlib.org. BLAS (Basic Linear Algebra Subprograms). `http://www.netlib.org`.

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

[17] 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.

[18] Qt Project. Qt Project. `http://qt-project.org/`.

[19] 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.

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

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

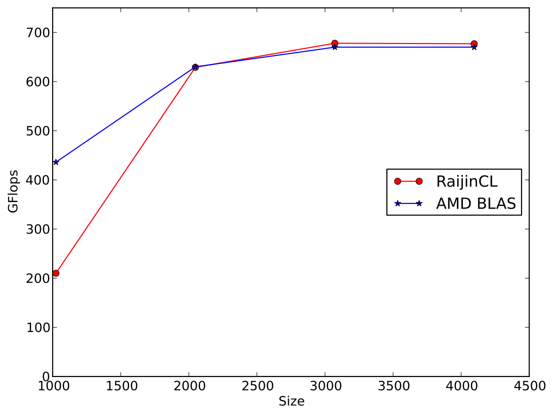
[22] Alex Rubinsteyn, Eric Hielscher, Nathaniel Weinman, and Dennis Shasha. Parakeet: A just-in-time parallel accelerator for python. In *HotPar 12*, 2012.

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

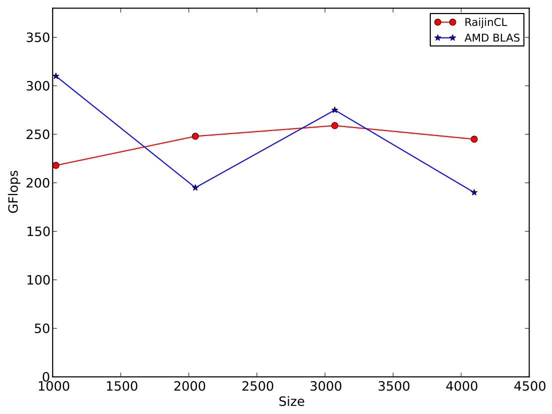
[24] Vasily Volkov and James W. Demmel. Benchmarking gpus to tune dense linear algebra. In *SC '08*, pages 1–11, 2008.

[25] Rick Weber and Gregory Peterson. A trip to Tahiti: Approaching a 5 Tflop SGEMM using 3 AMD GPUs. In *Symposium on Application Accelerators in High Performance Computing (SAAHPC), 2012*, 2012.

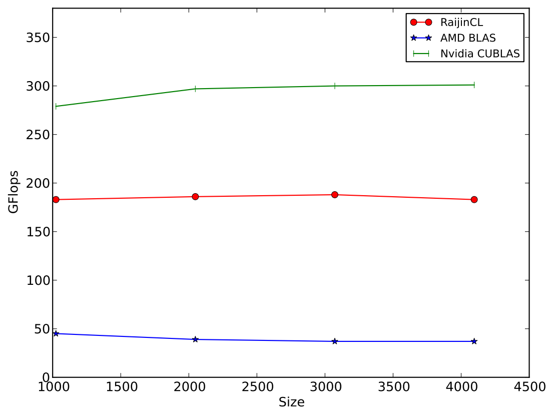
[26] X10.org. X10: Performance and Productivity at Scale. `http://X10-lan.org`.



(a) Radeon 7970 (Machine M1)

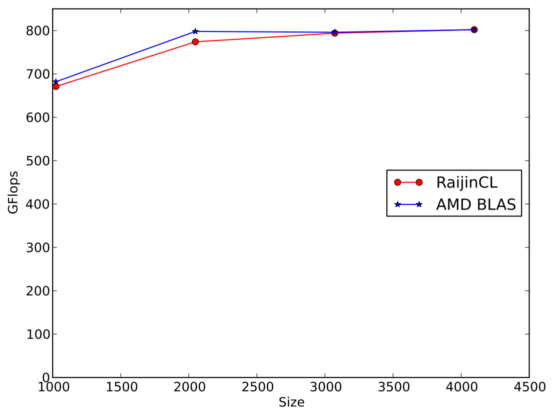


(b) Radeon 5850 (Machine M2)

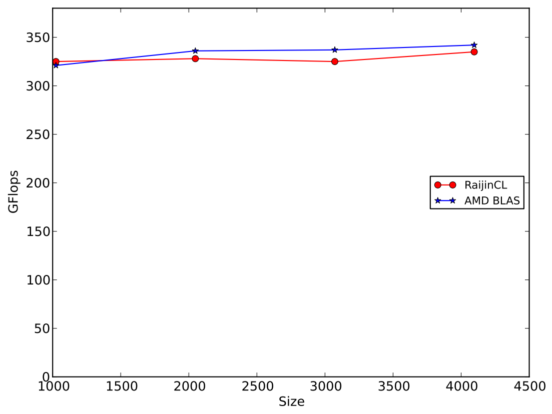


(c) Tesla C2050 (Machine M3)

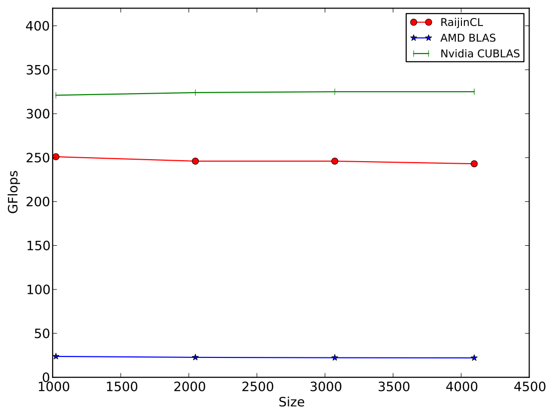
Figure 2: Performance of DGEMM



(a) Radeon 7970 (Machine M1)



(b) Radeon 5850 (Machine M2)



(c) Tesla C2050 (Machine M3)

Figure 3: Performance of ZGEMM