A TOOLKIT FOR BUILDING DYNAMIC COMPILERS FOR ARRAY-BASED LANGUAGES TARGETING CPUS AND GPUS

by

Rahul Garg

School of Computer Science
McGill University, Montréal

April 2015

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Abstract

Array-based languages such as MATLAB and Python (with NumPy) have become very popular for scientific computing. However, the performance of the implementations of these languages is often lacking. For example, some of the implementations are interpreted. Further, these languages were not designed with multi-core CPUs and GPUs in mind and thus don't take full advantage of modern hardware. Thus, developing just-in-time (JIT) compilers for these languages that allow scientific programmers to efficiently target both CPUs and GPUs is of increasing interest.

However building such compilers requires considerable effort. Prior to this thesis, there were no reusable compiler toolkits for array-based languages even though many of the compilation challenges are similar across languages. This thesis is about a set of two novel and reusable tools, Velociraptor and RaijinCL, that simplify the work of building JIT compilers for array-based languages targeting both CPUs and GPUs. Velociraptor is a reusable, embeddable dynamic compiler toolkit while RaijinCL is an auto-tuning high-performance matrix operations library.

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

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 and values of variables which are critical for inferring the properties of the region. Region specialization dynamically generates specialized code for the region based upon the runtime value of parameters identified by region detection analysis.

To demonstrate that Velociraptor is not tied to a single language or compiler toolchain, we present two case studies of using Velociraptor: a proof-of-concept Python compiler targeting CPUs and GPUs, and a GPU extension for a MATLAB JIT. We evaluated both these compilers built using Velociraptor against production-level static and dynamic compilers on a variety of benchmarks. We demonstrate that the CPU code generated by our toolkit is either competitive with, or outperforms, the available tools and that our parallel CPU and GPU code generation can provide further speedups.

RaijinCL was implemented to support matrix library operations found in array-based languages, such as matrix multiplication, on the GPU. RaijinCL an auto-tuning high-performance matrix operations library that runs across GPUs from many vendors. We present detailed experimental results from many different GPU architectures to show that it is competitive with vendor tuned libraries. Finally, while much of the literature about matrix libraries for GPUs is for discrete GPUs, we also demonstrate a prototype extension to RaijinCL to maximize system performance on single-chip CPU/GPU systems by using both the CPU and GPU.

Résumé

Les langages orienté-tableaux tels que MATLAB et Python (avec NumPy) sont de plus en plus populaires pour le calcul scientifique. Par contre, la vitesse de calcul des implémentations de ces langages laisse souvent à désirer ; par exemple, certains de ces langages sont implémentés avec un interpréteur. De plus, ces langages ne sont pas conçus pour bénéficier des avantages du matériel modernes, tels que les CPUs multi-coeurs et les GPUs. C'est pourquoi il y a un intérêt grandissant pour le dévelopement de compilateurs en ligne (JIT) pour ces langages qui permetterait aux développeurs scientifiques d'efficacement cibler les CPUs et GPUs.

Bien sûr, la construction de tels compilateurs demande des efforts considérables. Avant cette thèse, il n'existait aucune suite d'outils de compilation réutilisable pour les langages orienté-tableaux, et ce malgré qu'ils partagent plusieurs défis communs. Cette thèse porte sur deux nouveaux outils réutilisables, Velociraptor et RaijinCL, qui ont pour but de simplifier le travail de construction de compilateurs en ligne pour des langages orienté-tableaux ciblant les CPUs et GPUs. Velociraptor est un ensemble réutilisable et imbricable d'outils de compilation ; RaijinCL est une librairie auto-réglable d'opérations matricielles de haute performance.

Velociraptor offre une nouvelle représentation intermédiaire de haut niveau appelée VRIR qui a été spécifiquement conçue pour le calcul numérique. VRIR offre un support avancé pour les tableaux ainsi que des fonctionalités pour le parallélisme de haut niveau et pour le calcul sur GPU. Le développeur d'un compilateur utilise Velociraptor en générant du VRIR pour les sections clés d'un programme. Velociraptor offre également un compilateur optimisant qui génère du code pour CPU et GPU, ainsi qu'un système d'exécution intelligent pour la gestion du GPU.

Une contribution importante de cette thèse est une nouvelle technique de compilation dynamique appelée spécialisation de régions et qui est conçue spécifiquement pour le calcul numérique. La spécialisation de régions effectue tout d'abord une analyse de détection de régions, une nouvelle analyse de compilation qui identifie les régions de code qui peuvent être intéressante à analyser par le compilateur, tels que des boucles et des appels de routines sur des tableaux. L'analyse de détection de régions identifie également les paramètres, tels que la forme et la valeurs de variables, qui sont essentiels pour inférer les propriétés de la région. La spécialisation de régions génère dynamiquement du code spécialisé pour la région basé sur les valeurs des paramètres observées à l'exécution durant l'analyse de détection de région.

Afin de démontrer que Velociraptor n'est pas lié à un seul langage ou une seule chaîne de compilation, nous présentons deux études de cas : un compilateur pour Python qui cible CPUs et GPUs, et une extension GPU pour un compilateur en ligne pour MATLAB. Nous avons évalué ces deux compilateurs construits avec Velociraptor en les comparant à des compilateurs industriels (statiques et dynamiques) sur une variétés de programmes. Nous démontrons que la vitesse du code CPU généré par notre suite d'outils est compétitive ou même surpasse les performances offertes par les outils existants. De plus, notre génération de code parallèle pour CPU et pour GPU peut offrir des gains de vitesse supplémentaires.

RaijinCL a été développé afin de permettre l'exécution sur GPU des opérations matricielles trouvées dans les langages orienté-tableaux, telles que la multiplication de matrices. Nous présentons des résultats détaillés d'expérimentations sur plusieurs architectures de GPUs et nous démontrons qu'elle est compétitive avec les librairies optimisees des fabricants. Finalement, nous démontrons une extension prototype à RaijinCL qui permet de maximiser la performance système sur une puce CPU/GPU en utilisant autant le CPU que le GPU.

Acknowledgements

This thesis would not have been possible without the help from many people. First and foremost, I would like to thank my supervisor Prof. Laurie Hendren for her supervision. Without her supervision, I would likely never have finished the thesis because I would have picked a new project every week. She has taught me a lot about compilers and also about being a good researcher. I would like to thank McLab project members for discussing ideas and also for building tools and infrastructure such as Natlab and McVM that all of us could use.

The technical work is but a small part of an undertaking such as this thesis. There are many people who supported me in this journey. I would like to thank my wife Prachi for her patience and understanding. She is a strong woman and I would not have been able to do this without her support. I would also like to thank my parents for the words of encouragement and faith throughout the years. I also want to particularly thank my former labmate Amina who has been like a sister to me. Finally, I would like to thank my sister Pallavi who was always there for me.

Table of Contents

Abstract i

Résumé iii

Acknowledgements v

Table of Contents vii

List of Figures xv

List of Tables xvii

List of Abbreviations xix

1 Introduction 1

1.1 Design goals . . . 3

1.1.1 Focus on numerical programs and arrays . . . 3

1.1.2 Support for parallelism and GPUs . . . 3

1.1.3 Reusability across languages . . . 4

1.1.4 Integration with existing language runtimes . . . 4

1.2 Contributions . . . 5

1.2.1 System design . . . 5

1.2.2 Domain-specific and flexible IR design . . . 5

1.2.3 Region specialization . . . 6

1.2.4 Runtime optimizations . . . 6

1.2.5 Auto-tuning matrix library for GPUs . . . 6
1.2.6 Demonstrating the utility of Velociraptor . . . 7

2 Background **9**
2.1 LLVM . . . 9
2.2 OpenCL . . . 10
2.3 MATLAB language and implementations . . . 11
2.4 Python and NumPy . . . 11

3 System design **13**
3.1 Overall design . . . 13
3.2 Provided components . . . 15
3.2.1 Velociraptor . . . 15
3.2.2 RaijinCL . . . 16
3.3 Integrating Velociraptor into code-generation . . . 17
3.4 Exposing language runtime to Velociraptor through glue code . . . 17

4 Design of VRIR **21**
4.1 Structure of VRIR programs and basic constructs in VRIR . . . 22
4.1.1 Modules . . . 22
4.1.2 Functions and scoping rules . . . 23
4.1.3 Statements . . . 24
4.1.4 Expressions . . . 25
4.2 Supported datatypes . . . 25
4.2.1 Scalar types . . . 25
4.2.2 Array types . . . 26
4.2.3 Domain types . . . 26
4.2.4 Tuple type . . . 27
4.2.5 Void type . . . 27
4.2.6 Function types . . . 27
4.2.7 Dynamically-typed languages and VRIR . . . 27
4.3 Array indexing . . . 28

4.3.1 Array layouts . . . 28
4.3.2 Types of indices supported . . . 29
4.3.3 Array indexing attributes . . . 30
4.4 Array operators . . . 32
4.5 Support for parallel and GPU programming . . . 32
4.5.1 Parallel-for loops . . . 32
4.5.2 Atomic operations . . . 34
4.5.3 Parallel library operators . . . 35
4.5.4 Accelerated sections . . . 35
4.6 Error reporting . . . 36
4.7 Memory management . . . 36

5 Compilation engine **39**
5.1 Overview . . . 40
5.2 Simplification . . . 42
5.3 Inlining . . . 42
5.4 Alias analysis . . . 43
5.5 Loop information collection . . . 44
5.6 Preliminary bounds-check elimination . . . 44
5.7 CPU Code generation . . . 46
5.7.1 Code generation for index expressions . . . 47
5.7.2 Code generation and memory reuse optimization for library functions 47
5.7.3 Parallel CPU code generation . . . 48
5.8 GPU section code generation . . . 49

6 Region specialization **51**
6.1 Motivating examples . . . 52
6.2 High-level overview . . . 54
6.2.1 Potentially Interesting Region (PIR) . . . 54
6.2.2 Critical variables and inferred variables . . . 55
6.2.3 Data-flow analysis . . . 56

6.2.4 Multi-phase design . . . 57
6.3 Region detection analysis . . . 58
6.3.1 Flow-facts and notation . . . 59
6.3.2 Assignment statements . . . 60
6.3.3 If/else conditionals . . . 65
6.3.4 Loops . . . 66
6.4 Effect of combining inlining, region detection and region outlining . . . 69
6.5 Region shape inference . . . 71
6.5.1 Flow analysis . . . 71
6.5.2 Dealing with array growth . . . 73
6.6 Array bounds-check elimination . . . 73
6.6.1 Lower bounds checks . . . 75
6.6.2 Negative indexing checks . . . 75
6.6.3 Upper bounds checks . . . 76
6.6.4 Array growth . . . 76
6.7 Other optimizations enabled by region specialization . . . 76
6.7.1 Replacing dynamic lookup of array shapes and symbolic loop-bounds with constants . . . 76
6.7.2 Alias information passed to OpenCL compiler . . . 77

7 GPU runtime **79**
7.1 VRruntime API . . . 80
7.1.1 Array variables . . . 80
7.1.2 Tasks . . . 80
7.1.3 Array management . . . 81
7.1.4 Synchronization . . . 82
7.2 VRruntime data structures . . . 82
7.2.1 Array variable table . . . 82
7.2.2 GPU buffer table . . . 83
7.2.3 Event table . . . 83
7.3 GPU-agnostic VRruntime optimizations . . . 83

7.3.1 Asynchronous dispatch . . . 83
7.3.2 Copy-on-write optimizations . . . 84
7.3.3 Data transfers in parallel with GPU computation . . . 84
7.4 Increasing GPU utilization by using multiple GPU queues . . . 84
7.4.1 Determining peak floating-point performance . . . 86
7.4.2 Determining number of hardware queues . . . 86
7.4.3 Automatically using multiple hardware queues . . . 87

8 RaijinCL **89**
8.1 Introduction . . . 89
8.2 Contributions . . . 90
8.3 Library design . . . 92
8.3.1 Auto-tuning . . . 92
8.3.2 Asynchronous API . . . 93
8.3.3 Control over resource allocation . . . 93
8.3.4 Easy deployment . . . 94
8.4 GEMM kernels . . . 94
8.4.1 GEMM Background . . . 95
8.4.2 Codelets and search parameters . . . 96
8.5 Matrix-vector multiply kernels . . . 98
8.6 Transpose kernels . . . 99
8.7 Experimental results and analysis of GPU kernels . . . 100
8.7.1 Results . . . 100
8.7.2 Observations and analysis . . . 101
8.8 Tuning for single-chip CPU+GPU systems . . . 107
8.9 Results for single-chip CPU+GPU systems . . . 109
8.9.1 Performance . . . 109
8.9.2 Power analysis on Ivy Bridge . . . 110
8.9.3 Observations and performance guidelines . . . 112
8.10 Conclusions . . . 113

9 Two case-studies **115**
9.1 McVM integration . . . 115
9.1.1 McVM language extensions . . . 116
9.1.2 Integrating Velociraptor into McVM . . . 117
9.2 Python integration: PyRaptor . . . 119
9.2.1 Language extensions . . . 119
9.2.2 PyRaptor: Design, components and glue code . . . 122
9.3 Development Experience . . . 123

10 Compiler and runtime evaluation **125**
10.1 Benchmarks . . . 126
10.1.1 McVM benchmarks . . . 126
10.1.2 Dwarf based Python benchmarks . . . 127
10.1.3 Python NumFocus benchmarks . . . 127
10.2 Experimental setup . . . 128
10.3 Serial CPU performance against baseline compilers . . . 128
10.4 Parallel-loop performance . . . 130
10.5 GPU performance . . . 130
10.6 Region specialization . . . 132
10.7 Using multiple hardware queues . . . 132
10.8 Conclusions . . . 133

11 Related Work **135**
11.1 Building compilers for multiple languages . . . 135
11.2 Compilers for array-based languages for GPUs . . . 137
11.3 Multi-language compilers targeting GPUs . . . 138
11.4 Shape inference and region specialization . . . 138
11.5 Matrix libraries for GPUs . . . 140
11.6 Conclusion . . . 142

12 Conclusions and Future Work **143**

Bibliography **147**

Appendices

A VRIR textual syntax grammar **153**

B Rules for region detection analysis **163**

List of Figures

3.1 Possible design of a conventional compiler targeting CPUs. 13
3.2 Extending the containing compiler through provided tools 14
3.3 Containing compiler embedding Velociraptor, and showing the parts provided by Velociraptor 15

4.1 Example of a Python function with type declaration 23
4.2 VRIR generated from example Python code 24
4.3 Memory address offset of each array element from base of the array in three arrays of equal logical dimensions but different layouts 29
4.4 Example of array growth in MATLAB 31
4.5 An example of legal use of shared variables in parallel-for loop in pseudo-Python syntax 34
4.6 An example of illegal use of shared variables in parallel-for loop in pseudo-Python syntax 34

5.1 Compiler analysis, transformation and code generation infrastructure provided by VRcompiler 41
5.2 Example of LLVM IR code generation 46

6.1 Sample program written in pseudo-MATLAB syntax 53
6.2 Sample program written in pseudo-MATLAB syntax 54
6.3 An example in pseudo-Python code showing critical shape and value variables calculated by region detection analysis 56
6.4 A statement list may contain zero or more PIRs. Here showing a list with two PIRs 59

6.5 Example showing how combining inlining, region detection and region outlining performs an interesting refactoring 70

8.1 Tiles computed by a work-item, consecutive or offset elements 96
8.2 GPU throughput versus problem size for SGEMM 102

9.1 McVM before and after Velociraptor integration 118
9.2 Stencil example in plain Python and with our extensions 120
9.3 Integration of PyRaptor with CPython 122

List of Tables

8.1 Percentage of peak obtained on each device in best case 100
8.2 Machine configurations. All machines are 64-bit and use DDR3 memory . . 103
8.3 Optimal parameters for SGEMM 103
8.4 Optimal parameters for DGEMM 104
8.5 Optimal Parameters for CGEMM 104
8.6 Optimal Parameters for ZGEMM 105
8.7 Hybrid CPU+GPU tuning results (Gflops) 110
8.8 Tuned parameters for GPU kernel and load distribution for hybrid CPU+GPU GEMM 111
8.9 Measured peak power for various GPU kernels and CPU/GPU load distributions 111

10.1 Speedup of CPU code generated over baseline compilers Cython and McVM on machine M1 129
10.2 Speedup of parallel code generated by Velociraptor compared to serial code generated by Velociraptor 130
10.3 Speedup of hybrid CPU + GPU code generated by Velociraptor over parallel CPU code generated by Velociraptor 131
10.4 Speedup obtained by enabling region specialization over not enabling region specialization on machine M1 133

B.1 Data-flow rules for region detection analysis operating on assignment statements 167

List of Abbreviations

AST abstract syntax tree

BLAS basic linear algebra subsystem

GEMM general matrix multiply

GPU graphics processing unit

IR intermediate representation

JIT just-in-time

SSA single static assignment

VM virtual machine

Chapter 1
Introduction

Array-based languages such as MATLAB [23], Python (particularly NumPy [38]) and R [34] have become extremely popular for scientific and technical computing. The hallmark of these languages is that they provide powerful array abstractions for numerical programs. However, these languages were designed for productivity first and the implementations of these languages often suffer from poor performance. Evolving existing language implementations of array-based languages to provide better performance is a substantial challenge facing many of these languages.

This performance challenge has two aspects. The first aspect is that language implementations need to add capabilities such as just-in-time (JIT) compilation with good analysis and optimization passes. For example, reference implementations of both R and Python are interpreted. The initial implementaton of MathWorks MATLAB was also interpreted and Octave [14] has only recently added an experimental JIT compiler.

The second aspect of the performance challenge is to take advantage of modern hardware. Many of these languages were designed and implemented in the single-core era whereas modern hardware consists of multi-core CPUs and GPUs. For instance, nearly every laptop and desktop available today has a multi-core CPU and one or more GPUs capable of general-purpose computations. At the high-end of the spectrum, at the time of writing 75 out of 500 supercomputers are accelerated using GPUs and other many-core accelerators. The existing language implementations only allow harnessing a small fraction of the computational capacity of these machines. However, programming this modern

hardware remains a challenge and requires use of low-level APIs such as OpenCL [18]. A typical MATLAB or a NumPy programmer is more likely to be a domain expert, such as a biologist or a chemist, rather than a programming expert. Thus, there is interest in extending array-based languages to provide simple high-level language constructs that allow the programmer to utilize modern hardware and transfer the responsibility of generating low-level code to the compiler. Such extensions again require substantial compiler support.

Addressing both these performance challenges will require substantial effort from the compiler community. Further, both of these challenges are shared across all dynamic array-based languages but so far compiler development for these languages has happened in isolation with the compiler framework built for one language not benefiting users of other languages.

In this work, we introduce a collection of novel tools and techniques that help compiler writers to extend implementations of array-based languages to efficiently target CPUs and GPUs. Our tools are designed to be reusable across different compilers, and not tied to a single source programming language. The central piece of our tools is an optimizing compiler toolkit targeting CPUs and GPUs called Velociraptor. We implemented an optimizing, multi-pass compiler toolchain called VRcompiler that performs many analysis and transformations and produces LLVM for CPUs and OpenCL for GPUs. Velociraptor also contains a smart runtime system called VRruntime for managing GPU memory, data transfers to/from GPU as well as task dispatch to the GPU. Velociraptor includes many optimizations such as avoiding redundant data-transfers and automatically using multiple GPU command queues.

The next piece is a portable, high-performance matrix operation library for GPUs called RaijinCL. RaijinCL is an auto-tuning library that automatically generates optimized OpenCL kernels for a given GPU, and achieves performance similar to vendor tuned libraries while being completely portable. The unifying themes of all our tools are portability and specialization.

We want our tools to be portable across programming language implementations. Velociraptor is designed as a reusable toolkit that can be easily adapted and integrated into different language implementations. We show two case-studies about integrating Velociraptor into implementations of two different languages. We also designed our tools to be

portable across GPUs from different vendors. Therefore all our GPU tools are built upon portable technologies such as OpenCL, described in *Chapter 2*.

In order to achieve good performance, our tools adopt the concept of specialization in two contexts. First, we specialize based upon the properties of the user's machine. RaijinCL is an auto-tuning library that generates GPU code for matrix operations specialized for the user's machine. Some of the GPU runtime optimizations in Velociraptor are also driven by information collected about the user's machine. Second, Velociraptor introduces a new technique called region specialization, where the compiler generates a specialized version of the program based upon information obtained at runtime about shapes of arrays and loop-bounds.

We first examine some desirable properties, or design goals, that any proposed solution to the problem of building compilers for array-based languages targeting CPUs and GPUs should have. Then we conclude this chapter with the contributions of this work.

1.1 Design goals

1.1.1 Focus on numerical programs and arrays

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

1.1.2 Support for parallelism and GPUs

We are focusing on languages such as MATLAB and Python, which are popular because they are productive. A typical user of such languages will usually prefer high-level solutions for targeting modern hybrid (CPU/GPU) hardware One example is that a programmer,

or an automated tool, might simply specify that a particular computation or loop should be executed in parallel or that a section of code should be executed on the GPU. Thus, the intermediate representation (IR) of any proposed compiler infrastructure should be able to represent such high-level concepts.

Compiling these high-level constructs to either parallel CPU code or to GPU kernels, as required by the construct, should be the responsibility of the proposed solution. Managing GPU resources, as well as synchronization between the CPU and the GPU should also be handled by any proposed solution. Finally, array-based languages have operators such as matrix multiplication built into the language and a GPU implementation of these constructs also need to be provided.

1.1.3 Reusability across languages

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

1.1.4 Integration with existing language runtimes

Existing implementations face the challenge of maintaining compatibility with existing libraries. For example, many Python libraries have some parts written in C/C++ and depend on the CPython's C/C++ extension API. In such cases, a gradual evolution may be more pragmatic than a complete rewrite of the language implementation. For example, a compiler for numerical subset of Python can be written as an add-on module to the Python interpreter instead of writing a brand-new Python implementation. Tools such as Numba [29]

and Cython [39], which have become popular in the Python community, illustrate this approach.

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

1.2 Contributions

1.2.1 System design

The first contribution was to define the overall system design. We proposed the design criteria of our work in Section 1.1. The details of our proposed system design are given in *Chapter 3*. The system design involved defining the distribution of work and interfacing between Velociraptor and the compiler into which Velociraptor is being embedded such that the compiler writer integrating Velociraptor into a containing compiler needs to do minimal work.

1.2.2 Domain-specific and flexible IR design

The second contribution was defining a domain-specific IR called VRIR for Velociraptor, which is specialized for numeric computing. In particular, the array datatype is rich enough to model many of the properties of multiple programming languages such as MATLAB and Python's NumPy library. The IR is high-level and simple to generate from a containing compiler to facilitate easy integration of Velociraptor. Finally, in addition to serial con-

structs, the IR has high-level constructs toto express parallelism as well as expressing the parts of the program to be offloaded to a GPU. VRIR is described in *Chapter 4*.

1.2.3 Region specialization

We have implemented an optimizing, multi-pass compiler toolchain that performs many analysis and transformations and produces LLVM for CPUs and OpenCL for GPUs. Our compiler infrastructure is described in *Chapter 5*. The key novel contribution in the compiler is a new technique called region specialization. Region specialization consists of two phases. The first phase is called region detection analysis where VRcompiler identifies interesting regions of code in the program. In the second phase, VRcompiler collects some information, such as shapes of some array variables, and uses this information to generate specialized code for the region. The technique is described in *Chapter 6*.

1.2.4 Runtime optimizations

The compiler toolchain needs to be complemented by a GPU runtime. We have built VRruntime, a GPU runtime that provides a high-level task-graph based API to the code generator. The runtime automatically handles all data transfers and task dispatch to the GPU. VRruntime is implemented on top of OpenCL and is described in *Chapter 7*. VRruntime provides asynchronous dispatch, automatic CPU-GPU data transfer management and GPU memory management. While similar techniques have been implemented in other runtimes, the implementation in the context of array-based programming languages required solving several subtle issues and is a new contribution.

1.2.5 Auto-tuning matrix library for GPUs

RaijinCL is an auto-tuning library for matrix operations such as matrix multiply, matrix-vector multiply, transpose and other operations. RaijinCL is the first auto-tuning library for GPUs that works on recent GPUs from all major desktop GPU vendors in a single unified code-base. We also present extensive experimental results and discussion from a variety

of architectures. This includes discussion of Intel GPUs that are not discussed in previous literature on the topic.

In addition, we have also implemented a prototype extension to RaijinCL to single-chip CPU/GPU systems optimized for maximizing system throughput as opposed to only GPU throughput. Our experiments show that kernels optimized for GPU-only throughput may not work well in hybrid CPU/GPU situations.

1.2.6 Demonstrating the utility of Velociraptor

To demonstrate the reusability of Velociraptor, we have used it in two projects described in *Chapter 9*. The first case study is an extension of McVM [10], a just-in-time compiler and interpreter for MATLAB language. In this project, we embedded Velociraptor in McVM to enable support for GPUs, while CPU code generation is done by McVM. The second project is a JIT compiler designed as an add-on to the CPython interpreter. It takes as input annotated Python code and generates CPU and GPU Python code using Velociraptor. This demonstrates how a compiler could leverage Velociraptor to do both the CPU and GPU code generation. We briefly describe our experience in integrating Velociraptor into these two toolchains and then report benchmark results from both these case studies.

Chapter 2
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 portable infrastructures, LLVM for the CPU backend, and OpenCL for the GPU backend. We give a brief overview of these technologies. We also give a brief background about MATLAB and Python programming languages including Python's NumPy library.

2.1 LLVM

We use LLVM [22] 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.

2.2 OpenCL

We use the OpenCL [18] 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.

2.3 MATLAB language and implementations

MATLAB language is the language of MathWorks MATLAB computing system. MATLAB [23] language is a dynamically typed array-based language. MATLAB provides many powerful built-in operators for arrays such as matrix multiplication, element-wise array arithmetic and array slicing.

MATLAB language does not have a published specification and the behavior of the MathWorks MATLAB implementation and the user documentation provided by MathWorks provides an implicit language definition. MathWorks MATLAB was initially interpreted, but recent versions are JIT compiled.

MathWorks MATLAB is not the only implementation of the MATLAB language. GNU Octave [14] is a popular open-source interpreted implementation and recently an experimental JIT compiler has also been introduced. McVM, initially developed by Chevalier-Boisvert et al. [10], is a JIT compiled implementation of the MATLAB language. McVM is a type-specializing compiler that specializes functions based upon the type of the operands received at runtime. Experimental results from Chevalier-Boisvert et al. suggests that while MATLAB is a dynamically typed language, types of many variables can be successfully inferred. McVM targets serial CPU execution and uses LLVM as the backend compiler.

2.4 Python and NumPy

Python [33] is a general purpose, object-oriented, dynamically typed programming language. Python is popular in many different domains including scripting for system administration, server-side web development and as extension language for applications such as Blender3D.

In this work, we use Python version 3.2. Python language has many different implementations. The reference Python implementation is called CPython and it is also the most widely used implementation of the language. CPython converts to its own bytecode format and then interprets the bytecode. The bytecode format is not typed, and many of the instructions are quite generic. Thus, the interpreter is not very fast and loops written in Python can often be hundreds or thousands of times slower than equivalent loops in com-

piled languages such as C, C++ or Java. CPython exposes a C/C++ API that can be used by application programmers to write libraries in C/C++ that can create Python types, functions and objects that can then be used by regular Python code running in CPython. Libraries written in C/C++ using CPython APIs are called *extension modules*.

Out-of-the-box Python does not include any array datatype suitable for numerical applications. NumPy [38] is a popular extension module that provides a rich multi-dimensional array datatype. NumPy also provides many library functions, such as matrix multiplication, that are wrappers around high performance libraries such as BLAS and LAPACK. SciPy is a collection of Python libraries and extension modules for technical computing that uses NumPy for array operations. NumPy and SciPy, combined with Python's clean syntax and good general purpose foundations, have become popular for technical computing. For example, numerous books about NumPy and SciPy have been written, and several conferences are organized every year about using Python in technical computing domains.

Chapter 3
System design

3.1 Overall design

Consider a typical just-in-time (JIT) compiler for a dynamic language targeting CPUs shown in Figure 3.1. Such a compiler takes program source code as input, converts into its intermediate representation, does analysis (such as type inference), possibly does code transformations and finally generates CPU code.

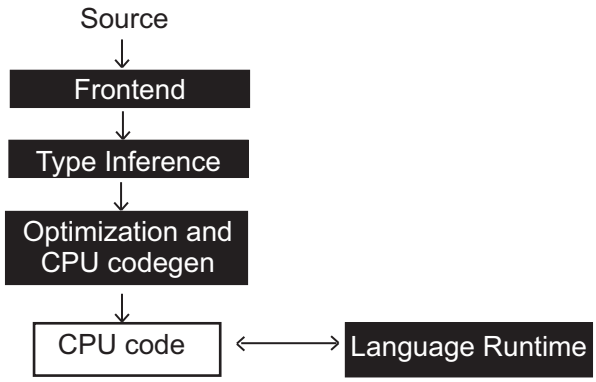


Figure 3.1 Possible design of a conventional compiler targeting CPUs.

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

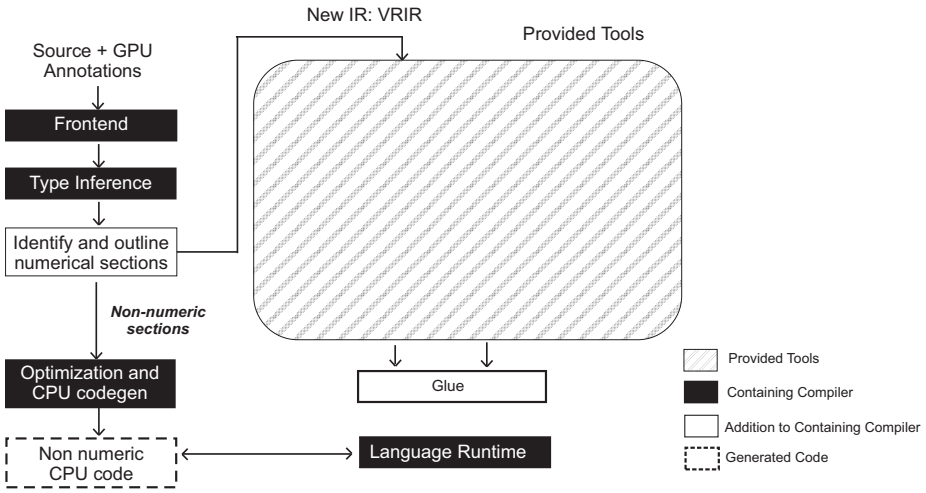


Figure 3.2 Extending the containing compiler through provided tools

introduce parallelization for multi-cores and also efficiently target GPUs. We have built two tools, the Velociraptor toolkit and RaijinCL GPU matrix operations library, to simplify the building of compilers that want to efficiently compile numerical computations to CPUs and GPUs. The key idea is that our tools are embedded inside the original compiler and take 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.

We call the compiler that embeds our tools as the *containing compiler*. Our design goal was to ensure that the containing compiler requires minimal work to integrate our tools. *Figure 3.2* demonstrates how our approach cleanly integrates with a containing compiler. The figure shows the original parts of the containing compiler in dark. The compiler writer who is integrating our tools into the containing compiler needs to add some new components to the containing compiler, and these are shown in white boxes. The tools described in this thesis are in the shaded box on the right.

The containing compiler identifies numerical sections in the code, outlines them and compiles them to a new IR called VRIR. Our tools (Velociraptor and RaijinCL) provide the compilation toolchain and supporting infrastructure necessary to compile and execute

VRIR on CPUs and GPUs. We provide an overview of the tools we provide in Section 3.2. The process of integrating Velociraptor with the code generator of the containing compiler is described in Section 3.3. The compiler writer integrating our tools also needs to provide some *glue code* and the required glue code is described in Section 3.4.

3.2 Provided components

The various components of the system are shown in *Figure 3.3*. We provide two components: Velociraptor and RaijinCL.

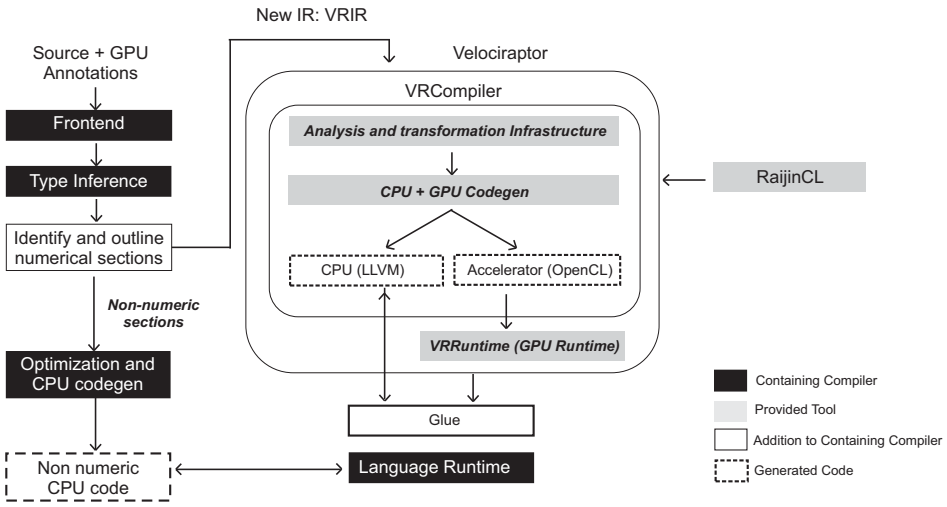


Figure 3.3 Containing compiler embedding Velociraptor, and showing the parts provided by Velociraptor

3.2.1 Velociraptor

Velociraptor consists of an optimizing compiler framework called VRcompiler and a smart GPU runtime called VRruntime. The VRcompiler takes VRIR as input and provides analysis and transformation infrastructure as well as code generation backends for CPUs and GPUs. Portability is a key concern for us and we chose to build our CPU and GPU back-

ends upon portable technologies, LLVM for CPUs, and OpenCL for GPUs. A high-level overview of VRcompiler is provided in *Chapter 5*. The VRcompiler implements some standard analysis such as reaching definitions and live variable analysis, but the distinguishing factor of VRcompiler is that it implements several key analyses and optimizations that are particularly beneficial for array-based languages such as bounds-check elimination. Some of these optimizations are performed using a new technique called region specialization introduced in this work. The VRcompiler identifies interesting regions of code using a new analysis called region detection analysis, and specializes the identified regions using shape and alias information obtained at runtime. The region detection and specialization is described in detail in *Chapter 6*.

One of the main motivations of this work was to provide tools to easily target GPUs. VRcompiler provides an optimizing GPU backend that can compile GPU sections identified in VRIR to OpenCL. The compilation toolchain is complemented by a smart GPU runtime called VRruntime. VRruntime efficiently manages GPU memory, CPU-GPU synchronization and task dispatch. VRruntime includes many optimizations such as avoiding redundant transfers, using all available GPU execution queues and asynchronous dispatch of work to the GPU. However, VRruntime is completely transparent to the containing compiler, and the containing compiler does not need to do any work to use VRruntime. The GPU compilation support in VRcompiler and VRruntime are described in Section 5.8.

3.2.2 RaijinCL

RaijinCL is a high-performance auto-tuning GPU matrix operations library and it provides the GPU implementation of operations such as matrix multiplication and matrix transpose. Velociraptor automatically generates calls to RaijinCL whenever required, and the containing compiler does not need to directly interact with RaijinCL. However, RaijinCL is a standalone library and not tied to Velociraptor. RaijinCL can be used by any application that needs fast array operations on the GPU. RaijinCL is described in detail in *Chapter 8*.

3.3 Integrating Velociraptor into code-generation

As shown in *Figure 3.2*, the compiler developer who wants to integrate Velociraptor into a containing compiler introduces several components in the containing compiler. The first new component 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 and outlines them into a new function. This new component will be called after conventional frontend passes and analysis such as type inference performed by many JIT compilers for dynamic languages. The second new component of the containing compiler compiles the outlined functions to VRIR.

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

3.4 Exposing language runtime to Velociraptor through glue code

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

Array data-structures in languages such as MATLAB and in Python/NumPy are more than just plain pointers to data. We will refer to these array data-structures as array objects. In addition to the pointer to the data, array objects might contain information such as size

(and potentially the layout) of the array as well as metadata such as reference counts for memory management. Different language runtimes will use different structures for representing array objects. However, Velociraptor needs to know the structure of the array objects in order to generate code as well as to implement the standard library functions such as matrix multiply.

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

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

Thus, to summarize, our tools integrate cleanly into the language implementation, and require minimal or no changes in the internal data structures of the existing implementa-

tion typically exposed in the C/C++ API ensuring compatibility with external libraries that may depend upon the API. We have built two-case studies (McVM and PyRaptor) about integrating Velociraptor and RaijinCL into a containing compiler and these are discussed in *Chapter 9*.

Chapter 4
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 informal study of thousands of MATLAB programs using an in-house static analysis tool. We have found that typically the core numeric computations in a program use a procedural style and mostly use scalar and array datatypes. We have based the design of VRIR on this study and on our own experiences in dealing with scientific programmers.

VRIR is a high-level, procedural, typed, abstract syntax tree (AST) based program representation. Each node in the AST may have certain attributes and may have children. Attributes of a node can be of four types: integers (such as integer id of a symbol), boolean, floating point (such as value of a floating-point constant occuring in the code) or a string (such as a name). We have defined C++ classes corresponding to each tree node type. We have also defined an S-expression based textual representation of VRIR. Containing compilers can generate the textual representation of VRIR and pass that to Velociraptor, or alternatively can use the C++ API directly to build the VRIR trees. The syntax for the textual representation for a VRIR node consists of the node name, followed by attributes and then children which are themselves VRIR nodes. The detailed syntax specification of the textual representation for VRIR can be found in *Appendix A*. The specification uses the grammar notation used by ANTLRv3 parser generator.

VRIR is not meant for representing all parts of the program, such as functions dealing

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

In the remainder of this chapter we present the important details of VRIR. The basic structure and constructs of VRIR are introduced in Section 4.1 and supported datatypes are introduced in Section 4.2. VRIR supports a rich array datatype which is flexible enough to model most common uses of arrays in multiple languages such as MATLAB or Python. Array indexing is introduced in Section 4.3 while standard array operators such as matrix multiplication are discussed in Section 4.4. In addition to serial constructs, VRIR supports parallel and GPU constructs in the same IR and these are discussed in Section 4.5. Finally, error handling and memory management are discussed in Section 4.6 and in Section 4.7 respectively.

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

4.1 Structure of VRIR programs and basic constructs in VRIR

4.1.1 Modules

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

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

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

Figure 4.1 Example of a Python function with type declaration

stop value, while some languages like MATLAB include the stop value.

4.1.2 Functions and scoping rules

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

Functions have a name, type signature, a list of arguments and a function body. The function body is a statement list. Similar to MATLAB or Python, VRIR supports multiple return values for functions. An example Python function with type declaration is shown in *Figure 4.1* and corresponding VRIR in S-expression form is shown in *Figure 4.2*.

Each function also has a symbol table where each symbol has a string name and a type. Inside the body of the function, only the integer IDs of symbols are referenced. VRIR has only three scopes: function scope, module scope and standard library scope. However, given that modules only consist of either functions defined within the module or declarations of external functions, and that the standard library also consists solely of functions and no data variables, the name lookup semantics is considerably simplified. Generally, all VRIR expressions reference variables using symbol IDs defined in the function's symbol table. However, a few constructs involve calling functions, such as function calls and construction of function handles. In these cases the function name is stored as a string attribute in the AST. The function name is looked up first in the function scope (either a local function handle or a recursive call), then at module scope and finally at the standard library scope.

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

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

Figure 4.2 VRIR generated from example Python code

4.1.3 Statements

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

VRIR has a number of structured control-flow constructs. If-else statements have a condition expression, an if-body and an optional else-body. For-loops are modelled after for-loops over range expressions in Python where a loop index takes on values in a specified range. The range expression is evaluated before the execution of the for-loop and thus the number of iterations is fixed before the loop begins execution except if a break, return or exception is encountered. In VRIR, we have defined a for-loop that iterate over multidimensional domains where the domain expression is evaluated before the loop begins execution. While-loops are more general loops and similar to while-loops in languages such as C or

Python. Each while-loop has a test expression and a loop-body. If the condition is true, then the body is executed and the process is repeated until the condition becomes false. Both for-loops and while-loops may exit early if a break, continue or return statement is encountered or if an exception is thrown.

One of the main goals of VRIR is to provide constructs for parallel and GPU computing through high-level constructs and we describe the supported constructs in Section 4.5.

4.1.4 Expressions

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

4.2 Supported datatypes

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

4.2.1 Scalar types

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

4.2.2 Array types

Array types consist of three parts: the scalar element-type, the number of dimensions and a layout. The layout must be one of the following three: row-major, column-major or strided-view. For example, a two-dimensional array of doubles in row-major format is written as *(arraytype float64vtype 2 row)* in VRIR s-expression syntax. The effect of layouts on address computations for array subscripts is described in Section 4.3.

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

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

4.2.3 Domain types

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

specialization of domains is called a range type, and is provided for convenience.

4.2.4 Tuple type

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

4.2.5 Void type

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

4.2.6 Function types

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

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

4.2.7 Dynamically-typed languages and VRIR

Languages such as MATLAB and Python/NumPy are dynamically typed while we require that the types of all variables be known in VRIR. A good JIT compiler, such as McVM [10],

will often perform type inference before code generation. For example, McVM dynamically specializes the code of a function at run-time, based on types of the actual parameters [10]. The McVM project shows that a type-specializing JIT compiler can infer the types of a large number of variables at runtime. Part of the reason of the success of McVM is that while MATLAB is dynamically typed, scientific programmers do not often use very dynamic techniques in the core numeric functions compared to applications written in more general purpose languages like JavaScript where dynamic techniques are more common. If a compiler is unable to infer types, users are often willing to add a few type hints for performance critical functions to their program such as in the Julia [8] language or the widely used Python compilation tool Cython [39]. Finally, if the containing compiler is unable to infer the types of variables, it can use its regular code generation backend for CPUs as a fallback, instead of using Velociraptor.

4.3 Array indexing

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

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

4.3.1 Array layouts

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

0	1	2
3	4	5

(a) Address of elements in a 2x3 row-major array

0	2	4
1	3	5

(b) Address of elements in a 2x3 column-major array

0	2	4
7	9	11

(c) Address of elements in a 2x3 strided array with strides (7,2)

Figure 4.3 Memory address offset of each array element from base of the array in three arrays of equal logical dimensions but different layouts

s_k are called strides of the array.

The key idea is that multidimensional arrays are just an abstraction over linear memory addresses. The memory address of each element is the sum of the address of the base of the array with an offset calculated based on the index of the element. Arrays of the same logical dimensions but with different layouts assign different offsets to the same index. We present examples of row-major, column-major and strided arrays in *Figure 4.3* to illustrate this difference. In this example we are showing three arrays, each of two rows and three columns.

4.3.2 Types of indices supported

Integer indices

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

Slicing arrays - Data sharing or copying

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

Using arrays as indices

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

4.3.3 Array indexing attributes

Negative indexing

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

Enabling/disabling index bounds checks

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

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

Figure 4.4 Example of array growth in MATLAB

Flattened indexing

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

Zero or one-based indexing

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

Array growth

Consider an array-write at a particular index that is outside the current bounds of the array. In some languages, such as NumPy, an out-of-bounds exception is thrown. However, in MATLAB, the array is grown such that array-write is to a valid index. An example of MATLAB behavior is shown in *Figure 4.4*. We have defined an optional binary attribute for array growth that can be specified for each array subscript dimension separately. If not specified, then arrays are assumed to not grow.

4.4 Array operators

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

4.5 Support for parallel and GPU programming

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

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

4.5.1 Parallel-for loops

Parallel-for loops are loops defined over a multi-dimensional domain where each iteration can be executed in parallel. A number of restrictions are placed on parallel loops in order to execute them safely in parallel. Parallel for-loops cannot contain break, continue or return statements because these do not make sense in a parallel context. Some restrictions are placed on errors such as out-of-bounds errors and these are detailed in Section 4.6. Parallel

for-loops can contain other parallel-for loops, but the implementation converts any inner parallel-for loops into serial loops.

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

Shared variables cannot be assigned-to inside parallel-for loops but indexed writes to shared arrays is still permitted. An example showing legal use of shared variables in VRIR is shown in *Figure 4.5*. An example with illegal use of shared variables is shown in *Figure 4.6*.

In order to understand the reasoning behind preventing reassignment to shared array objects, we need to distinguish between *array objects* and *data arrays*. The arrays in languages such as C/C++ are just pieces of memory, which we can call as the *data array*, whereas arrays in languages such as MATLAB and Python are objects. The array object may have many member fields such as the pointer to the data array, an array of integers for sizes and potentially memory management related metadata. We want that the shared array objects should be immutable inside a parallel-for loop. However, the data array pointed to by the object can be mutable and can be read/written by multiple threads. An indexed write only modifies the data array, while a reassignment may change the array object itself.

Disallowing reassignment to shared variables prevents modification of the shared array object and simplifies reasoning for both the programmer and Velociraptor. If the shared array object is immutable, then that implies that the size and strides of the array remain the same throughout the execution of the parallel loop. This simplifies reasoning for the programmer. For example, if *Figure 4.6* was legal, then the size and contents of the array *b* will be completely unknown after the loop has finished because the iterations may execute in any order. This is generally not desirable for the programmer.

Allowing reassignment to shared arrays would have also caused issues for the compiler. Let us assume we allowed assignment to shared variables and one thread attempts to reassign a shared array. Reassignment operation may require that the thread modifies one or more shared array objects. While one thread is performing the reassignment, the other threads need to be prevented from reading or writing the shared array objects to ensure that

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

Figure 4.5 An example of legal use of shared variables in parallel-for loop in pseudo-Python syntax

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

Figure 4.6 An example of illegal use of shared variables in parallel-for loop in pseudo-Python syntax

the threads are not reading/write partially formed array objects. In such a situation, we may need to insert locks around each use or definition of shared array variables which will have a significant impact on performance. Thus, disallowing reassignment or reallocation of shared variables simplifies the programming model for the programmer as well as the implementation.

4.5.2 Atomic operations

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

4.5.3 Parallel library operators

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

4.5.4 Accelerated sections

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

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

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

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

4.6 Error reporting

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

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

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

4.7 Memory management

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

Dealing with reference counts is particularly tricky. We had earlier considered a scheme where we defined explicit reference increment and reference decrement statements in VRIR and the containing compiler was required to explicitly insert these statements. This scheme has several drawbacks. Consider a statement such as $D = A * B + C$ where A, B, C and D are arrays. Here, the semantics dictates that a temporary array $t1$ will be created as follows: $t1 = A * B; D = t1 + C$. The reference count of $t1$ needs to be decremented once it is no

longer needed. If we impose the requirement that the containing compiler should explicitly insert reference increment and decrement instructions in VRIR, then we also impose the burden of simplification of the complex array expressions before generating VRIR so that all temporary arrays are explicit. This increased burden goes against our goals of requiring minimal work from the containing compiler.

We have moved the burden of dealing with reference counting to Velociraptor. Velociraptor will automatically generate calls to the reference increment and decrement functions using rules inspired from CPython's reference counting mechanism. As discussed above, correct implementation of reference counting needs to carefully consider any temporary arrays in the computation. We have defined a simplification pass in Velociraptor, described in *Chapter 5*, that simplifies the IR. The simplification pass is done before code generation. During code generation, Velociraptor uses the following rules on the simplified VRIR to insert code for reference increment and decrement:

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

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

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

Implementation of memory management requires some glue code to be supplied by the containing compiler, which is discussed in *Chapter 3*.

Chapter 5
Compilation engine

A compiler developer using Velociraptor only needs to concentrate on generating the high-level VRIR, and then Velociraptor provides an optimizing compilation engine called VRcompiler that compiles VRIR to LLVM code for CPUs and OpenCL for GPUs. In VRcompiler, we focus on high-level techniques and leave the work of low-level optimizations such as instruction selection, instruction scheduling and register allocation to LLVM and OpenCL driver for CPUs and GPUs respectively.

The distinguishing factor for Velociraptor is that we implement some analysis and optimizations particularly relevant to array-based languages, such as region specialization, shape inference and bounds-check elimination. The high-level nature of VRIR was instrumental in efficiently implementing many of these optimizations because semantic information about constructs such as array index expressions is lost while translating to lower-level targets like LLVM IR. Also, many of the optimizations are common to both CPUs and GPUs and our design decision to have both CPU and GPU sections in same IR (VRIR) led to a unified compiler design, where many of the analysis and optimization phases are shared across all backends.

The next section provides a high-level overview of VRcompiler and also provides a roadmap for how the compiler toolchain discussion is organized. This overview is followed by some of the standard analysis, code generation and optimization phases of the toolchain. More specialized topics, such as region specialization and GPU code generation, are described in more detail in other chapters. The techniques described in this chap-

ter are variations of existing techniques. Therefore, the focus of this chapter is to discuss the specific design choices in VRcompiler rather than give a detailed description of these algorithms.

5.1 Overview

The key compilation phases in VRcompiler are shown in *Figure 5.1*. The compiler first performs simplification and function inlining, described in Section 5.2 and Section 5.3 respectively. The objective of these phases is to simplify the code to make it easier for the rest of the toolchain to work on. After these transformations, we perform standard analysis such as reaching definitions, live variable and alias analysis. Reaching definitions and live variables are well-known analyses and their descriptions can be found in textbooks such as [2]. Alias analysis is also well-known, but there can be many different variations. We have implemented a fast variation of alias analysis and it is described in Section 5.4. The standard analysis phases are followed by a loop information collection pass described in Section 5.5. Next, Velociraptor performs a preliminary bounds-check elimination pass described in Section 5.6.

This is followed by region detection analysis, region outlining and region specialization. Region detection analysis finds potentially interesting regions (PIRs) that include regions of code and these regions are outlined into separate functions. The original functions (i.e. the non-outlined functions) are immediately passed to the code generator. The outlined functions are not immediately compiled and code generation for these functions is delayed until they are called. These outlined functions are dynamically specialized based upon properties such as array shapes of some of the parameters to these functions. The process of specializing regions at runtime is called region specialization and facilitates optimizations such as bounds-check elimination. Region detection and region specialization are novel technical contributions of this thesis and are described in *Chapter 6*.

Finally, VRcompiler performs CPU and GPU optimization and code-generation. For CPUs, we generate LLVM IR. Our CPU code generation strategy, including optimizations such as memory reuse optimizations, are described in Section 5.7. GPU code generation and optimization as well as the supporting GPU runtime VRruntime is covered in *Chapter*

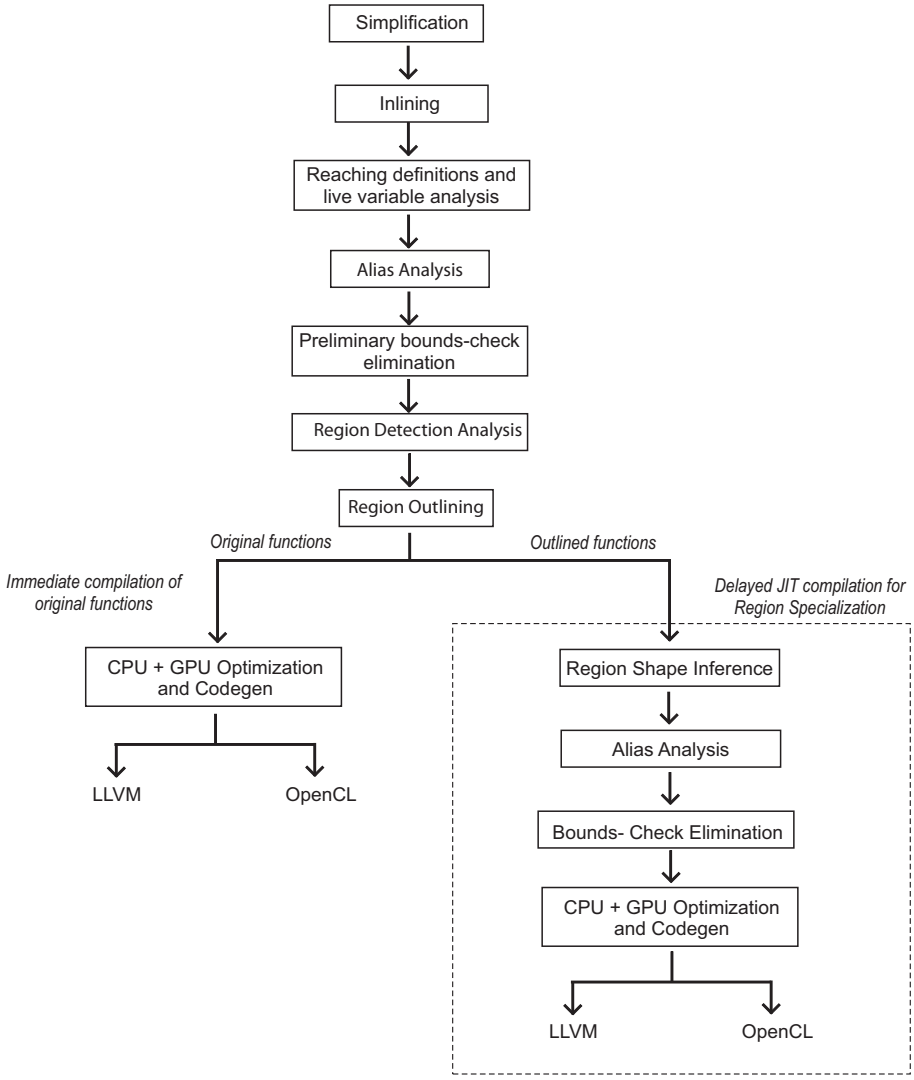


Figure 5.1 Compiler analysis, transformation and code generation infrastructure provided by VRcompiler

5.8.

5.2 Simplification

Before any analysis or optimization, VRcompiler first breaks down complex expressions so that the resulting IR is friendlier for subsequent analysis and transformation phases. Simplification pass has the following rules:

Making temporary arrays explicit: One of the objectives of simplification is to break down expressions involving arrays to make allocation of temporary arrays explicit. For example, consider the statement $D = A * B + C$ where A, B, C and D are arrays. In this expression, $A * B$ will create a temporary array is created. After simplification, we want to rewrite the statement as $t1 = A * B; D = t1 + C$. This objective is achieved by following the rule that if the child expression C of any expression E may allocate an array object, then C is assigned to a temporary variable t, and use of C is replaced with t.

Library calls: We want to make the use of library functions explicit in the IR. Therefore, if the child expression C of any expression E is a call to a library function, then C is taken out, assigned to a temporary and use of C is replaced by t.

Simplifying control flow constructs: Consider the condition expression of if/else statement or while statement, as well as the return expressions in a return statement. If these are not name expressions, then we assign the expression to the temporary, and replace the original use of the expression with the temporary.

Array index expressions: We want to make array read accesses explicit in the IR. Therefore, if the child expression C of any expression E is an array index read expression, then C is taken out, assigned to a temporary and use of C is replaced by t.

5.3 Inlining

We had two motivations for performing function inlining. The first consideration was accuracy of analysis. All analyses in Velociraptor are intra-procedural analysis and the primary goal of function inlining is to allow more accurate analysis, particularly shape inference. Without function inlining, function calls are often opaque barriers reducing the accuracy of the analysis.

The second motivation for function inlining was that we did not want function calls

inside loops and GPU regions. Function calls inside loops can reduce performance. For GPU regions, function calls inside GPU kernels are inlined by OpenCL driver whether we inline them or not. Therefore, we chose to inline functions in GPU sections ourselves as it gives us the ability to do more accurate analysis and better optimization.

Given these two motivations, we attempted to inline quite aggressively and chose to inline function calls wherever possible. We give priority to inlining function calls occurring inside loops and to function calls occurring inside GPU regions. If the called function itself includes function calls, those are also inlined. However, recursive functions or more generally functions that are part of a cycle in the call graph are not inlined. However, indiscriminately applying function inlining can result in an explosion of the size of the caller. Therefore, we only perform function inlining if the number of statements in the body of the caller function does not exceed a fixed constant. In the current implementation, this upper limit is fixed at 300 statements.

5.4 Alias analysis

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

5.5 Loop information collection

Loop information collection pass, as the name suggests, collects various types of information about loops occurring in a function. For each loop, it collects the the child and parent loops (if any). Array references occurring inside a loop are generally of interest to loop optimizations, and thus we collect the list of array index expressions occurring inside the loop. We also collect information about which variables are used or defined inside the loop. Some expressions, such as function calls, are potentially very costly. Thus, we collect information about which functions may be called inside the loop, whether there are any calls for array allocation and whether any library functions operating on arrays (such as matrix multiplication) are called inside the loop. Finally, some loops have a more complex control flow than others and we collect information whether the loop includes return, break or continue statements. The loop information collection pass does not require any iterative fixed point computations, and the runtime is linear in the number of statements and expressions occurring inside a function body.

5.6 Preliminary bounds-check elimination

Many array languages have array bounds-checks which can be a significant overhead, particularly on GPUs. Consider an array reference $A[i]$ where A is a one-dimensional array and i is an integer. There are potentially three checks that need to be performed. Let us assume the language uses 0-based indexing and let the array A have N elements. If the language supports negative indices, such as Python, then the compiler needs to check $i < 0$. If i is less than zero, then the actual index will be $i + N$. Once the effective index is established, we need to check if the index is both within the lower bound (i.e. greater than 0) and upper bounds (i.e. less than N).

VRIR encodes these three checks (negative indexing, lower bounds and upper bounds) directly in the IR as binary attributes of each dimension in the index expression. Thus, if the host compiler knows that some checks are not necessary due to language semantics or optimizations, then the host can simply encode this information in the VRIR given to Velociraptor. Similarly, if the bounds-checking optimization phases in Velociraptor can

prove that a check is not required, then the corresponding attribute is simply set to false without requiring any other change in the tree.

There are many approaches of minimizing the number of bounds-checks required. However, Velociraptor is a JIT compiler and thus compilation time is an issue, thus we have chosen to implement several fast, but effective, intra-procedural techniques that are similar to previous work [28] and are aimed to array references occuring inside loops. Our bounds-check elimination is performed in two phases. The first phase, which we call as preliminary bounds-check elimination is described here while the second phase is performed after shape inference and described in Section 6.6.

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

The second approach is based on elimination of redundant checks. 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. The two instances of $expr_0$ might have different values because the scalar variables involved in the expression might have been reassigned between the two occurences of $expr_0$. If the compiler can prove that 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. Velociraptor uses the following sufficient condition for proving value equality for the two instances of $expr_0$. If $expr_0$ is only a function of loop indices, loop invariant variables and known constants, then the value of the two instances of $expr_0$ in the same loop iteration will be equal. Velociraptor has already collected information about which variable are loop invariant, and which are loop indices, in the loop information collection pass and uses this information to determine if two instances of an expression are equal.

5.7 CPU Code generation

The compiler compiles CPU code to LLVM IR, and uses the LLVM 3.2 framework to generate native CPU code. LLVM detects the underlying CPU on a user's machine, and automatically performs lower-level optimizations such as register allocation and instruction selection based on properties of the user's CPU.

LLVM IR is based on SSA. LLVM documentation recommends that compilers using LLVM should not generate phi instructions because LLVM itself includes optimized algorithms for generating phi instructions. Instead, we generate "alloca" instructions for each variable at the beginning of the function. Any use and definition of a variable is treated as a load and store to the memory location associated with the variable. LLVM automatically eliminates such alloca instructions where possible and converts to SSA internally.

We illustrate the code generation scheme with an example. Consider an example $a = b + c$ where a, b and c are 64-bit floating-point scalars. We will generate an alloca instruction for each variable. We will generate two loads, one add and one store instruction for the statement and the generated LLVM IR is shown in *Figure 5.2*.

```
1  //alloca instructions are generated at the start of the function
2  // a,b,c are 64-bit floating-point variables in the original code
3  %a = alloca f64;
4  %b = alloca f64;
5  %c = alloca f64;
6
7  //Code for the statement a=b+c
8  //load the two operands into temporary variables
9  %1 = load f64* %a;
10 %2 = load f64* %b;
11 //perform the floating-point addition. result is a temporary value
12 %3 = fadd f64 %1 f64 %2;
13 //store the temporary value into c
14 store f64 %3, f64* %c
```

Figure 5.2 Example of LLVM IR code generation

We describe the code generation strategies for constructs of particular interest in VRIR: array index expressions, library operations on arrays and parallel loops.

5.7.1 Code generation for index expressions

Index expressions in systems such as NumPy and McVM are more complicated than index expressions in languages such as C. We already described the bounds-checking and negative indexing and our solutions for dealing with such issues. In this section, we focus on code generation once bounds checking has been done.

Consider a array index expression such $A[i, j]$ where A is a two-dimensional row-major array and i and j are integers. The compiler needs to generate the memory address corresponding to the array index, and then perform the load of the computed memory address. In NumPy, the address computation itself requires multiple steps. Array A is in fact a pointer to a structure. The structure contains fields such as the actual data pointer, an array containing the dimensions of A and other data. Let the data pointer field be called $dataptr$ and dimensions field be called $dims$. In such a case, the address computation will be $A->dataptr + sizeof(double) * (i * A->dims[1] + j)$. The actual pointer to data and the dimensions of the array need to be dynamically looked up. If the array index expression occurs inside a loop, then in our code generator, we move the dynamic lookups of data pointer and dimensions of the array outside the loop. For safety, we move lookups outside the loop only if the array is not redefined or growing inside the loop.

In addition to simple integer expressions, the compiler also needs to deal with expressions indexed using ranges. These are compiled into function calls to library functions.

5.7.2 Code generation and memory reuse optimization for library functions

VRIR has many library functions operating on arrays, such as binary element-wise operations on arrays, and we provide implementations of these library functions. Our implementation calls the BLAS where possible.

One of the issues in library implementation is that functions such as that many of the library functions allocate new arrays for the result. Consider the statement: $C = A + B$ where A, B and C are arrays. Language semantics of most array languages are that the expression $A + B$ will allocate a new array. A simple library function implementation will

take A and B as inputs, allocate a new array of the right size, perform the computation and return the new array.

However, the library function call may be inside a loop. Thus, the result array C may have been allocated in a previous iteration and may already be of the required size. In this case, we may be be able to avoid reallocating C. Thus, we have implemented a modified library function that takes C as input in addition to A and B, and checks if C is of the right size. The compiler only calls this optimized library function if C has a previous definition and if the array C is not aliased with other arrays. This optimization does not require any separate optimization pass and is implemented as part of our code generator. Overall, this optimization saves unnecessary memory allocations and reduces pressure on the memory allocator and garbage collector.

5.7.3 Parallel CPU code generation

Parallel CPU loops are defined over parallel domains of one or more dimensions. In the parallel CPU implementation, we parallelize the outermost dimension of the parallel domain. We divide the outermost dimension of the parallel domain into blocks of equal size, where the number of blocks is a constant multiple (two in the current implementation) of the number of hardware threads available on the user's machine. The number of blocks is kept greater than the number of threads because it allows some load-balancing at runtime. Different work blocks may require different amount of computation. If one thread finishes one work block early, it has the opportunity to start working on other work blocks without waiting for other threads to finish.

The program execution thread is called the master thread. The master thread divides the work into blocks, and places all blocks into a work queue. We maintain a shared counter to count the number of blocks of work completed so far. The master thread sets the counter to zero and goes to sleep. We maintain a pool of worker threads which are spawned only once and kept alive throughout the execution of the program. Worker threads are woken up whenever some work is available. Each worker thread checks the queue to see if work is available, and if a block of work is available, then removes it from the work-queue and starts executing it. When the worker thread finishes the work, it increments the shared

counter and checks to see if all work items are finished. If all work-items are finished, then the worker signals the master thread to wake up and goes to sleep. If all work-items are not finished, it checks the queue again to see more work is available. If the worker thread has no more work, then the worker thread goes to sleep.

5.8 GPU section code generation

Velociraptor supports GPU code generation and execution through the concept of GPU sections in VRIR. Any list of statements in VRIR can be marked as a GPU section. Inside GPU sections, library functions operating on arrays as well as parallel loops are offloaded to the GPU. VRcompiler is responsible for compiling parallel loops to OpenCL kernels. The GPU task management and data transfers are handled by VRruntime. For library calls such as matrix multiplication, VRcompiler generates high-level calls to VRruntime which then generates calls to RaijinCL. VRruntime is discussed in *Chapter 7* and RaijinCL is discussed in *Chapter 8*. In this section, we focus on code generation.

VRcompiler attempts to generate an OpenCL kernel for each parallel-for loop present inside the GPU section. VRcompiler generates the OpenCL source code for the kernel as well as the call to VRruntime to actually invoke the kernel. Details about kernel invocation in VRruntime are discussed in *Chapter 7*. A GPU section may also contain other constructs other than parallel-for loops and array library operations such as matrix multiplication. Inside GPU sections, array allocations and operations such as array slicing are handled by VRruntime and are performed on the GPU. VRcompiler generates CPU for all other constructs. For example, all scalar computations are handled on the CPU. Control flow dependences, such as an if-conditional choosing between two parallel loops, also need to be evaluated on the CPU.

We now describe the code generation for parallel-for loops. We translate the loop body of a parallel-for loop into the body of an OpenCL kernel. In other words, the body of the parallel-for loop is mapped to a single work-item in OpenCL. Each parallel-for loop in VRIR also has a corresponding list of shared variables. Shared variable in VRIR are converted to kernel parameters in OpenCL. VRcompiler creates a scalar parameter for each shared scalar variable. Array variables are more complicated. The data of the array will

be stored in an OpenCL buffer, but compiler also needs to pass shape information. Therefore, shared array variables are converted into multiple OpenCL parameters including the OpenCL buffer, an offset into the buffer where the array data begins, dimensions of the array, and the strides of the array for strided arrays. VRcompiler analyzes the loop-body and automatically generates declarations for local variables.

Velociraptor supports array bounds-checks inside GPU kernels as described in *Chapter 4*. In order to report errors, we maintain an OpenCL buffer with a single integer element which is initialized to zero to indicate no errors. This buffer, which we call as error buffer, is added as an argument to all generated OpenCL kernels. If a work-item encounters an out-of-bounds array access, the work-item writes the error buffer and exits. The error buffer is checked at the end of the GPU section and the error is reported.

The GPU code generation benefits from the analysis and optimizations implemented in Velociraptor including bounds check elimination passes and region specialization described in previous chapters. In order to get good performance, it is important to divide the computation into work-groups of specified size. Currently we use a fixed size of 64 work-items in a work-group which appears to work well across a range of currently available desktop GPU architectures. If the parallel-for loop is over a two-dimensional domain, then it is divided into two-dimensional work-groups of size 8×8.

In addition to generating the OpenCL kernels, VRcompiler also needs to generate metadata to be used by VRruntime. Velociraptor generates a task for VRruntime for each computation to be offloaded to the GPU. Each VRruntime task needs to specify the array variables which may potentially be read or written by an operation. VRcompiler generates this information by a simple traversal of the loop body.

Chapter 6
Region specialization

In array-based languages, accurate information about sizes and shapes of arrays and loop-bounds can enable optimizations such as bounds-check elimination. Information about array aliasing can also be important for some compiler optimizations. However, obtaining accurate shape and alias information in a just-in-time compiler are challenging data-flow analysis problems. Intra-procedural flow analyses do not know the properties of function parameters thus reducing their effectiveness. Inter-procedural approaches can be more accurate but can be too expensive to implement in a just-in-time compiler. Further, even inter-procedural static analysis are ineffective in programs driven by user input, file I/O or random number generation for input parameters to the core computation.

We introduce a new analysis called *region detection analysis* which identifies interesting regions of code and also identifies the key variables whose runtime properties affect the properties of the region. At runtime, Velociraptor examines the properties of identified key variables and uses this information to generate optimized code for the region that is specialized for the actual runtime values of the properties of the identified key variables. We call the process of generating optimized code for regions using runtime information as *region specialization*.

The main focus of region specialization is obtaining accurate shape information for better compiler optimizations, but we use the same infrastructure to also collect more accurate alias information. Compared to previous methods for shape inference in just-in-time compilers, which performed runtime shape inference only at the function level, our method can

be more fine-grained and thus potentially more accurate.

Before presenting the details, we first define some terms that are used in this chapter.

Region shape information: Region shape information for a given program segment consists of numeric values of the sizes and strides of arrays at every program point in the program segment. If the program segment contains one or more loop-nests such that the loop-bounds are affine expressions of outer loop indices, then the value of the expression defining the co-efficients of such loop-bound expressions at the program point just before the loop-nest is also included in the region shape information.

Region shape inference: Region shape inference is any method by which the compiler infers region shape information without any additional annotations from the programmer. In this work, we define a particular algorithm for region shape inference in Section 6.5 that runs on regions identified by region detection analysis, and utilizes information obtained at runtime.

Type of an array: We use the array type introduced in *Chapter 4*, but describe it here for reference. The type of an array consists of the element type, the number of dimensions and the layout (row-major, column-major or strided). The exact size array are not included in the type. For strided arrays, the strides are also not included in the type. Thus, two arrays of different sizes may be of the same type if they have the same element type, layout and number of dimensions.

The chapter is structured as follows. We present a motivating example in Section 6.1. A high-level overview of the approach is provided in Section 6.2. Our region detection analysis algorithm is presented in Section 6.3. Combining function inlining with region detection analysis and region outlining produces a powerful code transformation system. This is discussed in Section 6.4. Region shape inference algorithm is presented in Section 6.5. The optimizations enabled by region specialization, such as bounds-check elimination, are presented in Section 6.7.

6.1 Motivating examples

We consider an example of how region shape inference may work in a just-in-time compiler for a MATLAB-like language. We assume that the compiler has already performed type

inference.

```
1 function C = fn(A,B)
2 D = A+B;
3 C = 2*D;
4 end
```

Figure 6.1 Sample program written in pseudo-MATLAB syntax

First, consider the example in *Figure 6.1*. If the shapes of the function parameters A and B are known, then the shapes of the arrays C and D can be inferred by forward propagation of shape information starting from the function parameters. However, the shape of the function parameters may be different in each call to the function fn. Thus, the function may need to be specialized multiple times depending upon the shapes of the function parameters.

However, even knowing the shape of function parameters may not be enough. Consider an example code listing in *Figure 6.2*. The functions $opaque\_fn1$ and $opaque\_fn2$ are opaque functions whose return value is not known ahead-of-time. For example, they may be user-input functions. The compiler may want to specialize the program according to the shapes of matrices A and B, as well as the bounds of the loop-nest L in lines $12-17$. The shapes of the matrix B and the loop L are dependent upon the value of variables m and n at program point P just before line 10. The JIT compiler needs to wait until the program has executed to the program point P to perform shape inference and shape specialization. At program point P, we can accurately determine the shape of B and loop-nest L. The shape of A is also known at program-point P.

Thus, we conclude that performing shape inference at the entry of the function body may not be the best approach. Instead, we need to identify the program points where we have sufficient information, and the regions of code where shape inference and specialization can be performed.

```
function B = gn(A)
m = opaque_fn1();
c = opaque_fn2();
if(c):
  n = expr1;
else:
  n = expr2;
end
#We define a program-point P here.
B = zeros(m,n);
#Loop-nest L
for i=1:m
  for j=i:n
    t = A(j,j)*2;
    B(i,j) = A(i,j)*2 + t;
  end
end
end
```

Figure 6.2 Sample program written in pseudo-MATLAB syntax

6.2 High-level overview

6.2.1 Potentially Interesting Region (PIR)

Before we delve into the mechanics of shape inference, we need to first understand why we want to do shape inference. Shape inference is particularly important for compiler optimizations for loops operating on arrays, or for vector operations involving arrays. Not all vector operations are interesting. For example, if there is a single vector expression, then it can simply be compiled to a library call and the library call can examine the shape at runtime and call the appropriate implementation of the call. However, if there are multiple vector operations, or if there is an interesting loop-nest, then there may be some potential optimization. Thus, the compiler needs to identify regions of code where shape inference needs to be performed.

In addition to identifying interesting computations which may be specialized, the compiler also needs to identify the program point at which we have sufficient information to infer the shape information required to specialize the region. Keeping in mind these re-

quirements, we introduce the concept *Potentially Interesting Region* (PIR).

A block or region of code is called a Potentially Interesting Region (PIR) if it satisfies the following properties:

1. The region must consist of at least one of the following three computations: one or more loop-nests operating on arrays, two or more vector operations anywhere in the region or one or more vector operation inside a loop.

2. The region must have exactly one entry point.

3. The region must have exactly one exit point during the normal execution of the program, where a normal execution is defined as a case where no exceptions are thrown.

4. Shapes of all array variables occurring in the region, and at least the loop-bounds of at least the top-level loops, are inferable at the entry point of the region.

The definition of PIRs is purposefully left somewhat flexible. In particular, the requirement for shapes being inferable depends upon the shape inference algorithm used and may require some optimistic assumptions. Our compiler finds and specializes one particular class of PIRs. The exact algorithmic details of our system are given in the next section.

6.2.2 Critical variables and inferred variables

Given a potentially interesting region, inferring the shape information of the region may require knowing the shape or value of some variables at runtime at the entry point of the region. Such variables are called *critical variables*. We divide critical variables into two (potentially overlapping) sets. Critical value variables are variables whose value must be known at the entry point of the region for inferring shape information. Critical shape variables are arrays whose shapes must be known at the entry point of the region for inferring shape information.

Consider a statement such as $A = B + C$ where A, B and C are arrays. In this example, if we wanted to infer the shape of A, then we need to know the shape of variables B and C and therefore B and C are called critical shape variables. Consider an array allocation

example $D = zeros(m,n)$ [1] where D is a matrix and m and n are scalar integers. In this case, if we know the value of m and n, then we know the shape of D. Therefore, m and n will be critical value variables.

While critical value variables in the previous example were all scalars, sometimes array variables may also be critical value variables. For example, consider the case of indexing an array with another array such as $A(B) = C$ where A and B are arrays. In this example, if the language implements array-growth, then the shape of A after the statement may also depend upon the value of B. In this case, B will be a critical value variable. Marking array variables as critical value variables needs to be done with caution because propagating or storing array values for compiler analysis and optimizations can be very expensive.

6.2.3 Data-flow analysis

The examples described in the previous section were for single statements, but similar ideas apply to regions of code. However, for multiple statements, we also need to take data-flow across statements into account. We show an example of an identified PIR and calculated critical variables in *Figure 6.3*. The identified PIR consists of the array allocation and the for-loop. The first statement assigns n using some non-analyzable expression and therefore is not included in the region.

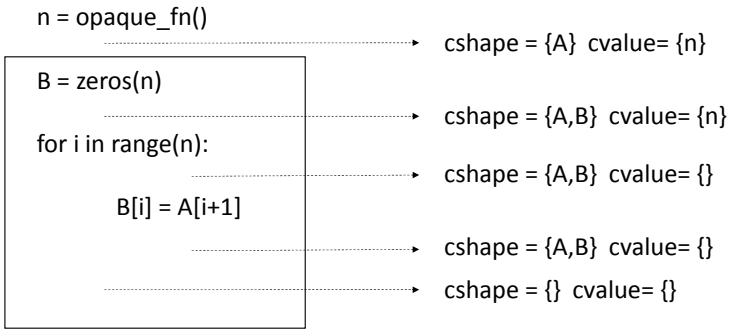


Figure 6.3 An example in pseudo-Python code showing critical shape and value variables calculated by region detection analysis

We show the critical value and shape variables at each program point in the region. While at most program points, the calculated sets are simple to understand, the reader may

[1] $zeros(m,n)$ will allocate a two-dimensional array of size (m,n)

be curious about the critical shape variables at the bottom of the loop body. Intuitively, the variables A and B may be used in the next iteration of the loop and therefore are still critical shape variables after the last statement of the loop body. The exact rules for data-flow analysis are presented in Section 6.3 which discusses data-flow across various types of statements such as assignment statements, conditionals and loops.

6.2.4 Multi-phase design

Region specialization infrastructure is implemented in multiple phases.

The first phase is called *Region detection analysis*. In this phase, Velociraptor identifies PIRs and outlines them into separate functions. This region detection and oulining happens once for each VRIR function. Consider a function F. Velociraptor will run region detection analysis and will outline the PIRs into new functions. For each PIR, Velociraptor also generates dispatcher functions that take the same arguments as the outlined function and replaces the PIRs in F with calls to their respective dispatcher functions.

The second phase occurs at runtime when the dispatcher function is called. The dispatcher functions are tightly integrated with the JIT compiler. The dispatcher functions maintain a code cache of specialized versions of the outlined functions. The code cache is a simple map where the keys consist of a tuple containing the shape and strides of critical shape arrays, the values of critical value variables, and the alias groups of the function parameters of outlined function. The value corresponding to each key is a pointer to the previously generated specialized code. The dispatcher examines the arguments received at runtime, and looks for a suitable specialized version in the cache. If a suitable specialized version of the code exists, then that version is executed. Otherwise the specialized version is generated as follows.

Velociraptor performs shape and alias analysis for the outlined function. Shape inference is described in Section 6.5. For alias analysis, Velociraptor reruns the alias analysis described in *Chapter 5* over the outlined function. The difference is that the previously described alias analysis is run before region detection analysis and region outlining, and makes conservative assumptions about function parameters. In this case, the alias analysis is run over the outlined function and starts with accurate aliasing information about the

input variables to the outlined region. Finally, Velociraptor generates specialized code for the outlined region using the information obtained by shape and alias analysis to perform better code optimizations. More details are given in Section 6.7.

In our current implementation, the code cache for an outlined PIR is restricted to storing last five generated versions. This ensures that the code cache does not grow arbitrarily if the PIR is called many times with arguments of different shape properties. Further, if the critical value variables for a function includes arrays, then a specialized version is stored in the cache if and only if the array is of length five or less. This ensures that the keys of the code-cache also remain of small size. Our region detection analysis attempts to avoid marking arrays as critical value variables, so this case is not very common.

6.3 Region detection analysis

Region detection analysis constructs PIRs from a given function body. Region detection analysis first collects a list of legal regions within a function body through a flow analysis described below. Legal regions are regions that satisfy all the conditions of a PIR outlined in the previous section except that they may or may not be interesting. Once the legal regions are collected, we do a simple inspection of each region and cull non-interesting regions from this list. The simplification and loop information collection passes detailed in *Chapter 5* must be run before running region detection analysis.

We define a routine called *FindPIR* that takes a statement list and initialization parameters for the flow analysis as input. FindPIR returns the list of legal regions found in this statement list. The returned list may be empty if no PIRs are found in the input statement list. Let us consider a list L of statements. L may have zero or more PIRs. For example, in *Figure 6.4*, we show a statement list with two PIRs. Each PIR is defined by the first and last statement of the PIR, the set of critical shape variables and the set of critical value variables. The first and last statement of the PIR can be stored simply as the index of the statements in L. To find this list of legal regions in a statement list, FindPIR performs a backwards data flow analysis.

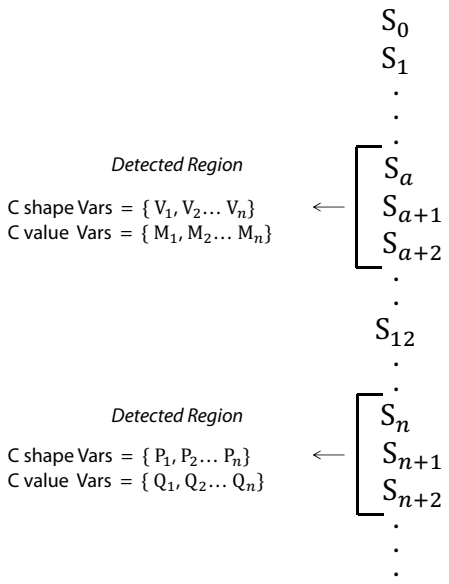


Figure 6.4 A statement list may contain zero or more PIRs. Here showing a list with two PIRs

6.3.1 Flow-facts and notation

As FindPIR is a data-flow analysis, we need to define the flow facts computed by the analysis. Consider a statement list L with N statements $S[0,L], S[1,L] \ldots S[N-1,L]$. We define $N+1$ program points $P[0,L], P[1,L], \ldots P[n,L]$ where $P[N,L]$ is the terminal program. Let us consider a program point $P[i,L]$ where $i \neq N$. $P[i,L]$ is the program point just before statement $S[i,L]$. At each such program point $P[i,L]$, FindPIR computes three properties:

1. The integer index $idx[i,L]$ of the last statement of the PIR that $S[i,L]$ belongs to. If $S[i,L]$ does not belong to any PIR, then this value is set to -1.

2. The set $cshapeVars[i,L]$ of critical shape variables at program point $P[i,L]$. If $S[i,L]$ does not belong to any PIR, then this set is empty.

3. The set $cvalueVars[i,L]$ of critical value variables at program point $P[i,L]$. If $S[i,L]$ does not belong to any PIR, then this set is empty.

The value of properties at $P[N,L]$ are supplied as function arguments to FindPIR and FindPIR propagates the information up the list L. At the function body level, FindPIR initializes the value of the flow facts $P[N,L]$ by setting the sets $cshapeVars[N,L]$ and $cvalueVars[N,L]$ as empty and $idx[N,L]$ as -1. The statement list L may contain compound statements. FindPIR may get called recursively on children of compound statements, and the recursive calls compute properties at program points internal to the compound statements. The initial properties parameter of these recursive calls depends upon the statement type and is discussed further below.

We have defined rules for how information is propagated from $P[i+1,L]$ to $P[i,L]$ based upon the statement type and properties of $S[i,L]$. The idea is to find a statement that may be the last statement of a region, and then keep building the region as long as possible. Let us assume the flow analysis reaches $S[i,L]$ and has so far determined that statement $S[i+1,L]$ is part of a region. Now, when analyzing $S[i,L]$, we can take one of three decisions:

1. We can decide that $S[i,L]$ belongs in the same region as $S[i+1,L]$. In this case, we will set $idx[i,L] = idx[i+1,L]$.

2. We may decide that $S[i,L]$ belongs to a new region. In this case, $S[i,L]$ will be the last statement of this new region and set $idx[i,L] = i$.

3. We may decide that $S[i,L]$ does not belong to any region. Then we set $idx[i,L] = -1$.

We now describe the rules used for each statement type. In the following discussion, we use the notation that array variables are indicated by capital letters and scalar variables are indicated using lower-case letters.

6.3.2 Assignment statements

Array library operation on RHS: Consider the case where RHS is an array library operation such as element-wise array arithmetic, matrix multiplication or math library functions

such as square root called element-wise on arrays. Simplification pass in Velociraptor enforces that the LHS will be a name expression in this case. The following equations are obeyed

$$
\begin{aligned}
cvalueVars[i,L] &= cvalueVars[i+1,L] \\
cshapeVars[i,L] &= cshapeVars[i+1,L] - name(LHS) + operands(RHS) \\
idx[i,L] &= \begin{cases} i & if\ idx[i+1,L] == -1 \\ idx[i,L & if\ idx[i+1,L] \neq -1 \end{cases}
\end{aligned}
$$

For example, consider $A = B + C$. In this case, $cshapeVars[i,L] = cshapeVars[i+1,L] - A + \{B,C\}$.

Array allocation on RHS: Consider an array allocation expression on the RHS that calls functions such as *zeros*, *ones* or *empty* that are defined in VRIR. LHS will be a name expression. The following rules are obeyed:

$$
\begin{aligned}
cvalueVars[i,L] &= cvalueVars[i+1,L] + operands(RHS) \\
cshapeVars[i,L] &= cshapeVars[i+1,L] - name(LHS) \\
idx[i,L] &= \begin{cases} i & if\ idx[i+1,L] == -1 \\ idx[i,L] & if\ idx[i+1,L \neq -1 \end{cases}
\end{aligned}
$$

Array element read on RHS: The LHS will be name expression. Let the array read in the RHS be *arr*.

We first consider the case where the LHS is not included in $cvalueVars[i+1,L]$, i.e. it is not a critical value variable. If the statement is within a for-loop with affine bounds and the subscript expressions are affine expressions of loop invariants and loop indices. In this case, the compiler may want to analyze the expression and may want to know the properties of the expression. We define the set $subscriptUses$ as the set of all variables used in any of the subscript expressions except for loop indices. We follow the following rule:

$$cvalueVars[i,L] = cvalueVars[i+1,L] + subscriptUses$$
$$cshapeVars[i,L] = cshapeVars[i+1,L] + arr$$
$$idx[i,L] = \begin{cases} i & if\ idx[i+1,L] == -1 \\ idx[i,L] & if\ idx[i+1,L] \neq -1 \end{cases}$$

If LHS is not a critical value variable, but the previous condition is not satisfied, then we simply include this statement in the region as follows:

$$cshapeVars[i,L] = cshapeVars[i+1,L]$$
$$cvalueVars[i,L] = cvalueVars[i+1,L]$$
$$idx[i,L] = \begin{cases} i & if\ idx[i+1,L] == -1 \\ idx[i,L] & if\ idx[i+1,L] \neq -1 \end{cases}$$

Let us now consider the case where the LHS is in $cvalueVars[i+1,L]$. Such cases may happen when the value on LHS is used to index another array, i.e. in cases where we have expressions such as $A[B[i]]$ in the original source code. In this case, we want to declare B to be a critical value variable. Let B be the array variable on the RHS. The following rules are applied:

$$cshapeVars[i,L] = cshapeVars[i+1,L] + B$$
$$cvalueVars[i,L] = cvalueVars[i+1,L] + B$$
$$idx[i,L] = \begin{cases} i & if\ idx[i+1,L] == -1 \\ idx[i,L] & if\ idx[i+1,L] \neq -1 \end{cases}$$

Array slicing on RHS: Consider an array-slicing operation such as $A = B(m:n)$.

$$cvalueVars[i,L] = cvalueVars[i+1,L] + \{m,n\}$$
$$cshapeVars[i,L] = cshapeVars[i+1,L] - A + B$$
$$idx[i,L] = \begin{cases} i & if\ idx[i+1,L] == -1 \\ idx[i,L] & if\ idx[i+1,L] \neq -1 \end{cases}$$

Array as subscript expression on RHS: Sometimes an array may be used to subscript another array. For example, consider the expression $A[B]$ in Python where A and B are arrays. In this case, the result is an array the shape of the result depends upon the shape of the array subscript. Let us assume that the array *subArr* is the array used to subscript the array *arr*. In this case, we follow the rule:

$$cshapeVars[i,L] = cshapeVars[i+1,L] + subArr + arr - name(LHS)$$
$$cvalueVars[i,L] = cvalueVars[i+1,L] + subArr$$
$$idx[i,L] = \begin{cases} i & if\ idx[i+1,L] == -1 \\ idx[i,L] & if\ idx[i+1,L] \neq -1 \end{cases}$$

Array element write on LHS: In a language such as Python, writing to an array element will not change the shape information and thus such statements will not cause any modifications in the three sets. However, in languages such as MATLAB, writing to an array index can grow the array. First, we consider the case without array growth. In such a case, the rules followed are identical to array element read.

Next, we consider the case where the array can grow. We include the statement in the region if and only If the array element write is within a for-loop with affine bounds, and the subscript expressions are affine expressions of loop invariants and loop indices. If the condition is satisfied, we follow the rules:

$$cvalueVars[i,L] = cvalueVars[i+1,L] + subscriptUses$$
$$cshapeVars[i,L] = cshapeVars[i+1,L] + arr$$
$$idx[i,L] = \begin{cases} i & if\ idx[i+1,L] == -1 \\ idx[i,L] & if\ idx[i+1,L] \neq -1 \end{cases}$$

Array as subscript expression on LHS: Let the array being subscripted be *arr* and the array used as subscript be *subArr* and the array on RHS be $name(RHS)$. Let us first consider the case without array growth. In this case, we follow rules similar to the array as subscript on RHS.

$$cvalueVars[i,L] = cvalueVars[i+1,L] + subArr$$
$$cshapeVars[i,L] = cshapeVars[i+1,L] + subArr + arr + name(RHS)$$
$$idx[i,L] = \begin{cases} i & if\ idx[i+1,l] == -1 \\ idx[i,l] & if\ idx[i+1,l] \neq -1 \end{cases}$$

In cases where the language semantics implies potential array growth, we do not include this statement in the region.

Scalar operation on RHS, scalar variable on LHS Consider a scalar operation on the LHS where the LHS does not involve any array operations. We distinguish three cases:

1. The LHS is not a critical value variable. We follow the equations:

$$cvalueVars[i,l] = cvalueVars[i+1,L]$$
$$cshapeVars[i,l] = cshapeVars[i+1,L]$$
$$idx[i,L] = \begin{cases} i & if\ idx[i+1,L] == -1 \\ idx[i,L] & if\ idx[i+1,L] \neq -1 \end{cases}$$

2. The LHS is a critical value variable, and the RHS is an expression involving only

addition or multiplication of scalars, or a scalar value comparison such as an equals to comparison.

$$cvalueVars[i,L] = cvalueVars[i+1,L] - name(LHS) + operands(RHS)$$
$$cshapeVars[i,L] = cshapeVars[i+1,L]$$
$$idx[i,L] = \begin{cases} i & if\ idx[i+1,L] == -1 \\ idx[i,L] & if\ idx[i+1,L] \neq -1 \end{cases}$$

3. If the LHS is a critical value variable, but the conditions for the previous case are not satisfied, we terminate the region by setting $idx[i+1,l] = -1$ and setting the corresponding critical sets to empty.

6.3.3 If/else conditionals

If/else conditionals need to be handled carefully. Let L_{if} and L_{else} be the statement lists corresponding to the if-branch and else-branch respectively. Let the length of these lists be N_{if} and N_{else} respectively. Using our notation, $P[0,L_{if}]$ and $P[N_{if},L_{if}]$ are the program points post and prior to the body of the if-branch. Similarly, $P[0,L_{else}]$ and $P[N_{else},L_{else}]$ as the program points post and prior to the else-branch. The program points prior to and after the entire branch statement are $P[i,L]$ and $P[i+1,L]$ respectively.

Let us assume that statement $S[i+1,L]$ is part of a region. We first attempt to determine if the branch statement can be included in this region. We set the critical sets and *idx* at $P[N_{if},L_{if}]$ and $P[N_{else},L_{else}]$ with the value at $P[i+1,L]$. We then recursively call the Find-PIR on the if and else bodies, which will determine the values at $P[0,L_{if}]$ and $P[0,L_{else}]$. We verify if the *ifBodyList* contains a single region covering the entire if-body and the *elseBodyList* contains a single region covering the entire else-body.

If the verification passes, we distinguish between two cases. If neither the if-body nor the else-body contain any definition of any of the variables in $cshapeVars[i+1,L]$ and $cvalueVars[i+1,L]$, then we merge as follows:

$$cvalueVar[i,L] = union(cvalueVars[0,L_{if}], cvalueVars[0,L_{else}])$$
$$cshapeVars[i,L] = union(cshapeVars[0,L_{if}], cshapeVars[0,L_{else}])$$
$$idx[i,L] = \begin{cases} i & if\ idx[i+1,L] == -1 \\ idx[i,L] & if\ idx[i+1,L] \neq -1 \end{cases}$$

If the verification passes, and if either the if-body or the else-body contains a potential definition of any of the variables in $cshapeVars[i+1,L]$ and $cvalueVars[i+1,L]$, then we need to consider which branch is potentially taken as well. Therefore, we need mark the boolean condition variable (let's call it V) of the branch statement as a critical value variable. We merge as follows:

$$cvalueVars[i,L] = union(cvalueVars[0,L_{if}], cvalueVars[0,L_{else}]) + V$$
$$cshapeVars[i,L] = union(cshapeVars[0,L_{if}], cshapeVars[0,L_{else}])$$
$$idx[i,L] = \begin{cases} i & if\ idx[i+1,L] == -1 \\ idx[i,L] & if\ idx[i+1,L] \neq -1 \end{cases}$$

If the verification fails, then the branch statement $S[i,L]$ cannot be included in the region that $S[i+1,L]$ belongs to. However, the if/else branches may still contain feasible regions. To find these, we now reset the flow facts at at $P[N_{if},L_{if}]$ and $P[N_{else},L_{else}]$ by setting idx at both points to -1 and the critical sets to empty, and run FindPIR again on these branches to identify any feasible regions.

6.3.4 Loops

Before we describe the rules, we observe some desirable properties. We either include the loop as a whole in a region, or exclude it entirely. We do not want regions that reside inside a loop-body because whenever a region is encountered, a recompilation may be triggered. If a region is inside a loop body, then recompilation may be triggered many times. Further,

we want that the critical value variables at any program point inside the loop should be loop-invariant. If a critical value variable is changing in every iteration of the loop, then such values cannot be propagated by region shape information and will lead to unknown shapes.

We now define some notation. From the convention in this section, the program points just before and after the loop are $P[i,L]$ and $P[i+1,L]$ respectively. Let L_b be the statement list corresponding to the loop body and N_b be the number of statements in the loop body. Let the number of statements in the loop body be m and the statements within the body are numbered 0 to $m-1$. Using our notation, the program points just before and after the loop body are $P[0,L_b]$ and $P[m,L_b]$ respectively. If the loop is an for-loop with loop bounds that are affine functions of outer loop variables, then we define the set of variables used in the loop-bound expressions excluding the loop indices as *loopBoundParams*. If the loop does not have affine loop bounds, then *loopBoundParams* is defined to be empty.

We define a helper function $TestLoop(stmt, cshape, cvalue, idx)$ that tests whether the loop body of the loop *stmt* is a single region if the program properties at $P[post,L]$ are initialized with the values *cshape*, *cvalue* and *idx*. The function returns if the loop body is a single region, and as a side effect computes the program properties within the loop body. The function is shown in *Algorithm 1*.

Let us consider the case where the statement $S[i+1,L]$ is in a region (i.e. $idx[i+1,L] \neq -1$). We first attempt to see if the loop can be included in this region. This test is done by calling $TestLoop(S[i,L], cshape[i+1,L], cvalue[i+1,L])$. If the test returns true, then we propagate the values from $S[i+1,L]$ as well properties computed within the loop. So we apply the following rule:

$$cshapeVar[i,L] = setUnion(cshapeVars[i+1,L], cshapeVars[0,L_b], loopBoundParams)$$
$$cvalueVars[i,L] = setUnion(cvalueVars[i+1,L], cvalueVars[0,L_b], loopBoundParams)$$
$$idx[i,L] = idx[i+1,L]$$

If the previous test returns false, then we check if the loop can be last statement of a new region by calling $TestLoop(stmt, \{\}, \{\})$. If this test returns true, then we follow the

Algorithm 1 Algorithm for testing whether the loop body is a single region, given flow analysis initialization settings

```
1:  procedure TESTLOOP(stmt, cshape, cvalue)
2:      L_b ← getLoopBodyStmtList(stmt)
3:      m ← length(L_b)
4:      cshapeVars[m, L_b] = cshape
5:      cvalueVars[m, L_b] = cvalue
6:      idx[m, L_b] = −1
7:      while notConverged(P[0, L_b]) do
8:          FindPIR(stmt, cshapeVars[m, L_b], cvalueVars[m, L_b], −1)
9:          if idx[0, L_b] ≠ m − 1 then
10:             break
11:         end if
12:         cshapeVars[m, L_b] = setUnion(cshapeVars[m, L_b], cshapeVars[0, L_b])
13:         cvalueVars[m, L_b] = setUnion(cvalueVars[m, L_b], cvalueVars[0, L_b])
14:     end while
15:     return idx[0, L_b] == (m − 1)
16: end procedure
```

following rules:

$$cshapeVars[i,L] = setUnion(cshapeVars[0,L_b], loopBoundParams)$$
$$cvalueVar[i,L] = setUnion(cvalueVars[0,L_b], loopBoundParams)$$
$$idx[i,L] = i$$

Finally, if neither test returns true, then we do not include $S[i,L]$ in any region and set $idx[i,L] = -1$ and set the critical variable sets to empty.

So far, we have not discussed how to handle break and continue statements. Typically data-flow analysis are defined on somewhat lower level IRs such as control-flow graphs but we are dealing with VRIR ASTs which may contain break and continue statements. Our handling of these statements is done in a way similar to McSAF [12] analysis framework.

Algorithm 2 Outline of function FindPIR used to find PIRs

```
 1: procedure FINDPIR(L, cshape, cvalue, idx)
 2:     N ← length(L)
 3:     cshapeVars[m,L] = cshape
 4:     cvalueVars[m,L] = cvalue
 5:     idx[m,L] = idx
 6:     for i ← m−1...0 do
 7:         S[i,L] ← L[i]
 8:         stype ← getStmtType(S[i,L])
 9:         if stype == assignment then
10:             ...
11:         else if stype == ifelse then
12:             ...
13:         else if stype == loop then
14:             if idx[i+1,L] ≠ −1 then
15:                 result ← TestLoop(S[i,L], cshape[i+1,L], cvalue[i+1,L])
16:                 if ¬result then
17:                     result ← TestLoop(stmt, {}, {})
18:                 end if
19:             end if
20:         end if
21:     end for
22:     return reconstructRegionListFromFlow()
23: end procedure
```

6.4 Effect of combining inlining, region detection and region outlining

The combination of function inlining, region detection analysis and region outlining can be a powerful combination of code transformations. The net effect of these transformations can be to rewrite code to be more amenable to code specialization based upon properties such as region shape information and aliasing properties. This is particularly true when there is a function call in the body of a loop. Consider an example written in Python-like language. The original code, shown in *Figure 6.5a*, contained a function call to *stencil* inside a loop. After inlining, the function *stencil* will be inlined in the function body.

Region detection may decide that the loop is an interesting loop, and the region will then get outlined. The resulting code is shown in *Figure 6.5b*.

```
1
2  def stencil(A,B):
3    n = A.shape[0]
4    for i in range(1,n):
5      A[i] = (B[i]+B[i-1]+B[i+1])/3
6
7  def foo(A,B)
8    ⋮
9    for i in range(m):
10     stencil(A,B)
11   ⋮
```

(a) Original code calling a function inside a loop

```
1  #Critical shape vars = {A,B}. Critical value vars = {m}
2  def outlinedPIR(A,B,m)
3    for i in range(m):
4      n = A.shape[0]
5      for i in range(1,n):
6        A[i] = (B[i]+B[i-1]+B[i+1])/3
7
8  def foo(A,B)
9    ⋮
10   outlinedPIR(A,B,m)
11   ⋮
```

(b) Refactored code produced by inlining, region detection and region outlining

Figure 6.5 Example showing how combining inlining, region detection and region outlining performs an interesting refactoring

In the final code, three properties hold true making it a good target for specialization:

1. The loop-body is not spread out over multiple functions. The nested-loop structure is made explicit and thus easier for the compiler to analyze.

2. Only interesting computations, such as reasonably analyzable loops, are outlined. Thus, the other compiler optimization passes know that the computation in this function is potentially interesting.

3. Shape properties of variables and functions occuring in the function body should be computable simply from information about the function parameters. This is not guaranteed but is likely due to heuristics used by region detection analysis.

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

6.5 Region shape inference

Region shape inference computes shape information at each program point in a region. We have implemented a flow-sensitive, intra-region, forward propagating, iterative data-flow analysis for region shape inference. Region shape inference is called upon the outlined PIRs. The core of region shape inference consists of a flow analysis, which is similar to constant propagation, and is discussed in Section 6.5.1. However, in order to handle potential array growth, results of the flow analysis may get invalidated and another iteration of the flow analysis may be required. Handling array growth is discussed in Section 6.5.2.

6.5.1 Flow analysis

Consider a program point $P[i, L]$ inside the region. Region detection analysis must have computed the critical shape variables and critical value variables at $P[i, L]$. At each program point $P[i, L]$, region shape inference computes the sizes and strides of critical shape variables and the value of critical value variables. The required variable values, array shapes and strides shapes can either have the value *unknown* or have a constant known numeric value.

Region shape analysis can be thought of as a variant of classical constant propagation analysis. In constant propagation, the compiler typically forward propagates the values of

scalar variables. In the best case, constant propagation aims to compute the value of all live scalar variables at a given program point. In region shape inference, we instead aim to compute the values and shapes of variables in critical value and critical shape sets at any given program point. Region shape inference does not propagate the values of variables that are not critical value variables.

Further, region shape inference does not propagate a single value and instead propagates a value range. For critical value scalars, the range is the current estimate of the lowest possible value and the highest possible value of the variable at the given program point. For critical value arrays, the range is the current estimate of the lowest and highest possible value of any array element in the array.

The propagation of flow facts across conditional branches and the fixed-point computation required for loops is similar to constant propagation. One of the key points in defining a data-flow analysis is to specify the merging rule given two flow-facts. The merge rule in region shape inference is also similar to the rules in constant propagation. If we want to merge the value of array shape, stride or value of critcal value variable from two different branches, and if the values are different, then the result is marked as unknown. If the value in the two branches is the same, then the merge result also retains the same value.

Constant propagation often requires computing the values of simple arithmetic expressions involving either constants, or symbols whose value is known. Similarly, VRcompiler performs such scalar arithmetic if required for propagating critical value variables. Region shape inference also requires evaluating the shape of three types of expressions: library functions involving arrays, slicing operation involving arrays and and array allocation functions. The shape of the result of most library functions can be computed if the shape of the operands is known. For example, given a matrix multiplication expression $A \times B$, VRcompiler can compute the shape of the result if the shapes of A and B are known. The shape of the result of an array allocation functions can be computed if the value of the operands is known. Similarly, the shape of the result of an array slicing operation can be trivially computed if the value of all subscript expressions is known.

6.5.2 Dealing with array growth

In *Chapter 4*, we introduced the concept of array growth in VRIR inspired by languages such as MATLAB. If an array element write, such as $A(i, j) = v$ is such that the array subscript expressions are out of current bounds, then the array size is increased such that the index expression is now within bounds.

In the flow analysis described in the previous section, usually we make the conservative assumption that if we encounter a potentially growing array index write expression, then we do not know the shape of the array after the expression. However, there is one exception to this rule. If we encounter affine array index write expressions in loops with affine loop bounds, then we optimistically assume that the index expression is not growing.

After region shape inference is done, Velociraptor has more accurate information about loop bounds and affine co-efficients of the index expression. Using this information, Velociraptor verifies whether the optimistic assumption that affine array index expressions do not lead to array growth is true using algorithm specified in Section 6.6. If Velociraptor finds that one or more affine array index expression might lead to array growth, then Velociraptor discards the results from the previous flow analysis. Instead, Velociraptor repeats the flow analysis but no longer makes optimistic assumptions about the cases which were found to be potentially growing. The process of flow analysis and verification of optimistic assumptions about array growth are repeated until we find a fixed point.

6.6 Array bounds-check elimination

We already described a preliminary bounds-check elimination phase in Section 5.6. We now describe another approach for bounds-check elimination using region shape information. This approach is applicable for loop-nests with affine bounds, i.e. loop-nests where the bounds of the inner loop variables are affine functions of outer loop variables. Let us consider a m-dimensional loop-nest with indices $i_1, i_2, \ldots, i_m$. The loop bounds of a m-dimensional affine loop-nest can be defined by m affine inequalities for lower-bound and m inequalities for upper bounds. The lower-bound and upper-bound inequalities are given in (6.1) and (6.2) respectively.

$$
\begin{aligned}
& i_1 >= l_{01} \\
& i_2 >= l_{02} + l_{12} * i_1 \\
& i_3 >= l_{03} + l_{13} * i_1 + l_{23} * i_2 \\
& \vdots \\
& i_m >= l_0 m + l_{1m} * i_1 + l_{2m} * i_2 \ldots + l_{(m-1)m} * i_{m-1}
\end{aligned} \tag{6.1}
$$

$$
\begin{aligned}
& i_1 <= u_{01} \\
& i_2 <= u_{02} + u_{12} * i_1 \\
& i_3 <= u_{03} + u_{13} * i_1 + u_{23} * i_2 \\
& \vdots \\
& i_m <= u_0 m + u_{1m} * i_1 + u_{2m} * i_2 \ldots + u_{(m-1)m} * i_{m-1}
\end{aligned} \tag{6.2}
$$

The co-efficients u_{jk} and l_{jk} occurring in these affine expressions might be symbolic expressions involving loop-invariant variables. However, during region detection analysis any such loop-invariant variables required for computing the loop-bounds will be marked as critical value variables and the region shape inference will then propagate the value of these variables. Thus, if the region shape inference is successful, we can assume that the values of these co-efficients is a known integer constant.

Consider an array index expression occurring inside a loop with affine bounds where the array size is known to be a loop-invariant. The array index expression may have one more than one dimension. However, for bounds-check elimination we can consider subscript in each dimension independently. We focus on subscript expressions that are affine functions of loop-index variables. Let the affine subscript expression have the value v in a given loop-iteration given by (6.3). We can assume that the co-efficients c_k are known after

region shape inference.

$$v = c_0 + c_1 * i_1 + c_2 * i_2 + \ldots + c_m * i_m \tag{6.3}$$

Given the above equations, and the assumption that the array size is loop-invariant and known before code generation, we can verify whether the subscript expression is always within bounds. We discuss the possible cases.

6.6.1 Lower bounds checks

Let us define an integer constant *minIndex* that represents the lowest acceptable value for a given subscript expression that does not throw out-of-bounds error. The value of *minIndex* depends upon the language semantics. For example, for MATLAB *minIndex* is 1 while for Python, the value is $-D+1$ where D is the size of the array in the subscript dimension. We can verify that an array subscript expression is always greater than or equal to *minIndex* by contradiction. Let us assume that there is a value of v defined in (6.3) such that the inequality in (6.4) is true.

$$v < minIndex \tag{6.4}$$

We construct a system of affine integer inequalities by combining (6.1), (6.1), (6.3) and (6.4). We then use an integer linear programming (ILP) solver to find a feasible solution to this system of equations. If there does not exist a feasible solution to this set of equations, that implies that there does not exist any valid value of v such that $v < minIndex$. Therefore, if no feasible solution is found, the subscript is always greater than or equal to *minIndex* and the compiler can eliminate the bounds-check for this subscript.

6.6.2 Negative indexing checks

The technique used for lower-bounds checks can also be used to verify whether an index may be negative by constructing another equation set with *minIndex* set to zero.

6.6.3 Upper bounds checks

The technique used for eliminating upper bounds-checks is a slight variation of the technique for lower bounds-check. Let the maximum acceptable index for the subscript expression be *maxIndex*. The value of *maxIndex* depends upon the language. For languages with zero-based indexing, this value will be $D-1$ where D is the size of the array in the subscript dimension and for one-based languages *maxIndex* will be D. For proof by contradiction, we build the inequality in (6.5) and then try to find a feasible solution for the system of affine inequalities formed by (6.1), (6.1), (6.3) and (6.4). If no feasible solution is found, then compiler eliminates the upper bounds-check.

$$v > maxIndex \tag{6.5}$$

6.6.4 Array growth

For languages that permit array growth, such as MATLAB, array writes may grow the array if and only if the subscript expression is greater than current array size in the subscripts dimension. Thus, if the subscript expression is always less than or equal to *maxIndex*, then the subscript expression does not grow the array.

6.7 Other optimizations enabled by region specialization

In this section, we discuss optimizations enabled by region specialization, except for bounds-check elimination which was discussed in the previous section.

6.7.1 Replacing dynamic lookup of array shapes and symbolic loop-bounds with constants

Many expressions, such as array index expressions, require dynamic lookup of array shapes. However, after region shape inference, Velociraptor knows the shape of many arrays before

code generation and therefore these dynamic lookups are replaced with constant values. For example, consider an array index operation $A[i, j]$ on a two-dimensional row-major array. Without region shape inference, the access needs to be compiled to $baseAddress(A) + i * (A->dims[1]) + j$. However, if we know the shape of A, then the $A->dims[1]$ can be replaced by a constant. Similarly, consider a loop with symbolic loop-bounds. After region shape inference, many such symbolic bounds can be replaced with constants and can aid lower-level compilers such as LLVM or OpenCL driver to perform better loop unrolling or other optimizations.

6.7.2 Alias information passed to OpenCL compiler

Alias analysis information is used during GPU code generation. If Velociraptor has accurate aliasing information about aliasing of arrays, it can often be conveyed to the OpenCL compiler using keywords such as *restrict* which declares that the pointer the keyword is applied to is not aliased with other pointers occuring in the OpenCL kernel. Further, accurate alias information can allow Velociraptor to prove that certain arrays are only read but not written inside a given OpenCL kernel and this information can be passed to the OpenCL compiler using the *const* keyword. OpenCL compiler uses such annotations to perform optimizations such as enabling use of GPU's read-only texture cache for reads of arrays marked as *const restrict*. Consider a case where an array A is written inside a loop and array B is only read inside the loop. The compiler may be able to prove that A and B are not aliased by using alias information obtained after region specialization and then declare array B as *const elemtype* restrict* where *elemtype* is the element type of the array B.

Chapter 7
GPU runtime

We use the OpenCL API for performing computations on GPUs. OpenCL is a low-level API and exposing the full complexity of the API to the code generator was not very desirable. We therefore implemented VRruntime, which provides a simple high-level interface to the compiler and abstracts over both the OpenCL API as well as RaijinCL API. VRruntime manages the actual execution of the GPU kernels, GPU memory allocations as well as data-transfers. We briefly describe the API exposed by VRruntime in Section 7.1 and the internal implementation details in Section 7.2.

VRruntime is a smart runtime and includes several optimizations. Some of the optimizations, such as asynchronous dispatch and avoiding unneeded data transfers between CPU and GPU, are generally beneficial for all GPUs. Such optimizations are described in Section 7.3. We also implemented a novel optimization of using all available GPU queues when each individual computation kernel is too small to fully utilize the GPU. The technique is well known amongst expert GPU programmers but VRruntime is the first runtime to perform it automatically. This optimization depends upon the size of the computation, number of available GPU queues and the computational capacity of the GPU. We have implemented a command-line utility that performs micro-benchmarks to automatically determine the number of queues, and the peak floating point performance of the GPU. The optimization heuristic for using multiple queues, and the micro-benchmarks performed to automatically determine parameters, are described in Section 7.4.

7.1 VRruntime API

VRruntime API is designed as a high-level API that exposes four concepts: array variables, tasks, array management and synchronization primitive.

7.1.1 Array variables

Each array variable has a unique integer ID, similar to VRIR. During the lifetime of a program, an array variable may refer to different GPU buffers. VRruntime does not have a concept of scalar variables, because purely scalar computations will be handled on the CPU. In order to use an array variable allocated on the CPU in a computation, the array variable must first be *submitted* to VRruntime. Submission of a variable is done through an API call that takes the array integer identifier, and the pointer to the array object. Once the variable has been submitted to VRruntime, VRruntime will automatically make a copy the array data on the GPU if required. The CPU and GPU copies are not guaranteed to be synchronized until a synchronization operation is explicitly invoked. Thus, once a variable has been submitted to VRruntime, it should not be accessed on the CPU without performing a synchronization operation.

7.1.2 Tasks

VRruntime exposes a single implicit task queue, and VRcompiler enqueues tasks to this queue to perform any computation on the GPU. Two types of tasks can be enqueued in VRruntime: Kernel calls and array library function calls such as matrix multiplication. Kernel calls and array library function calls return an event ID that may be used later on by the client of VRruntime. Tasks map to either a single OpenCL kernel call, or to a sequence of OpenCL kernel calls. Whenever a task enqueue operation is called on VRruntime, the corresponding OpenCL calls are immediately enqueued in the appropriate OpenCL queue by VRruntime.

Library function calls such as matrix multiplication specify the list of array variables used as input and output, and the value of any scalar variable inputs. For kernel calls, whenever VRcompiler generates OpenCL code for a kernel, it registers the kernel with

VRruntime. Registration of a kernel involves specifying the source code of a kernel, the list of array variables used as input arguments to the kernel, as well as the array variables which may be read or written by the kernel. The registration function returns a unique kernel ID, which can be used to invoke the kernel. Registration is done during code generation, and once the kernel is registered it may be invoked any number of times. Each kernel call enqueue operation specifies the kernel ID to enqueue, parallel iteration domain over which to execute the kernel, as well as the value of any scalar variables. The array variables to be used are already provided during registration of the kernel.

7.1.3 Array management

Array management functions include operations such as array allocation, array copy and array slicing. These operations may lead to allocation or deallocation of buffers on the GPU. Unfortunately OpenCL does not allow queuing these operations in an OpenCL queue and any calls to allocate GPU buffers may be performed immediately. However, if insufficient memory is available, buffer allocation may fail. Determining whether a memory allocation may fail is also not feasible. OpenCL buffers are managed by the OpenCL runtime, and there is no guarantee whether a buffer is actually placed in GPU memory at any give time.

Given the OpenCL API limitations, a fully asynchronous API for array management functions was not feasible. VRruntime maintains it's own information about the memory allocated so far. VRruntime queries the OpenCL runtime about the maximum amount of memory available on the device. If it determines that an array memory operation may lead to OpenCL buffer allocation failure, it first finishes all pending tasks and then determines if any buffers can be deallocated. If sufficient memory becomes available, then it is allocated and program execution can proceed. Otherwise, the computation is moved to the CPU automatically by simply executing the OpenCL kernel on the CPU.

Finally, VRruntime also supports marking array variables as dead. If an array variable is marked dead, then it implies that the memory associated with the array variable may be deallocated once all tasks related to the variable enqueued before the mark-dead operation are finished.

7.1.4 Synchronization

VRruntime exposes three types of synchronization operations. The first operation, called *waitAll*, finishes execution of all pending VRruntime tasks and synchronizes the data back to the CPU. The second operation, called *waitForVariable*, takes an array ID as input and finishes all pending tasks that may affect the value of the specified variable. The third operation, called *waitForEvent*, takes an event ID as input and waits until the specified event is finished.

7.2 VRruntime data structures

Internally, VRruntime maintains several tables which are updated whenever an API function is called.

7.2.1 Array variable table

The most important data-structure in VRruntime is an array variable table which maintains information about the state of array variables. The table is indexed with the array ID and includes many types of information. For each array ID, we store information about the size, shape and layout of the array that the array ID is currently pointing to. We also store the CPU address range pointed to by the array variable. The address range information is used for data transfers between CPU and GPU, and for determining the aliasing properties of arrays. If the array has been allocated on the GPU, then we also store information about the GPU buffer which holds the data and the offset within the GPU buffer where the data starts. The offset is necessary to handle multiple array slices all pointing to sub-ranges within another larger array. We store information about whether the CPU copy, GPU copy or both contain the freshest data. This determines if a data transfer may be necessary. We also store a copy-on-write flag for each variable to implement the copy-on-write optimization described in the next section. Finally, we store the list of tasks (identified using event ID) which may read or write the array variable. This list is necessary to implement the *waitForVariable* synchronization operator discussed above.

7.2.2 GPU buffer table

Apart from the array variable table, VRruntime maintains a GPU buffer table for holding information about GPU buffers allocated by VRruntime. This is complementary to the array access table. For each GPU buffer, we maintain a list of variables whose data is contained in the buffer and we also maintain the list of tasks which may read the buffer, and the list of tasks which may write the buffer.

7.2.3 Event table

Finally, we maintain an event table where we maintain the mapping between event IDs generated by VRruntime to OpenCL events. This table is used when we need to wait upon an event through any of the synchronization operators in VRruntime. This table is updated whenever a new task is enqueued.

7.3 GPU-agnostic VRruntime optimizations

VRruntime has several important optimizations that allow for good performance:

7.3.1 Asynchronous dispatch

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

7.3.2 Copy-on-write optimizations

Consider the case where variable *A* is explicitly cloned and assigned to variable *B*. VRruntime does not perform the cloning operation until required. For each computation submitted to VRruntime after the copy, VRruntime examines the operands read and potentially written in the task using meta-information submitted to VRruntime by the compiler. If VRruntime does not encounter any computation that may potentially write *A* or *B* after the deep copy, then the deep copy operation is not performed. This is a variation of copy-on-write technique, where we can characterize our technique as copy-on-potential-write. Copy-on-write has been implemented for CPU arrays in systems such as MATLAB [40], and for various CPU data-structures in libraries such as Qt, but we implement it for GPU arrays.

7.3.3 Data transfers in parallel with GPU computation

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

7.4 Increasing GPU utilization by using multiple GPU queues

GPUs are massively parallel processors that are most efficiently utilized in workloads with many work-items. For example, currently available high performance GPUs may have tens of SIMD processors, and the GPU vendors recommend having hundreds of active work-items per SIMD processor. Thus, top-end GPUs require thousands of work-items active at the same time for full utilization of the hardware.

However, some workloads may involve a number of calls to kernels with small number of work-items instead of a single kernel with many work-items. Consider a computation

consisting of two kernels S_1 and S_2 with small number of work-items but where S_1 and S_2 do not have any control or data dependencies. Traditionally, GPUs only had a single command queue where kernels were enqueued, and the GPU executed these kernels in sequence. For example, if S_2 was enqueued after S_1, the single command queue did not begin execution of S_2 until S_1 had finished. Thus, a single GPU queue creates implicit task serialization.

Newer GPUs, such as GPUs based on AMD's GCN architecture, have multiple command queues and kernels enqueued in different command queues can execute in parallel. For example, if S_1 and S_2 were enqueued in different command queues, then they can execute in parallel. Using multiple command queues can lead to more efficient utilization of the GPU hardware.

There are three problems with applying this optimization technique automatically in a runtime. First, OpenCL does not provide a way to query the number of hardware queues available. Second, using multiple command queues is only beneficial when the computations are small. The definition of small depends upon the computational capacity of the GPU. A matrix multiplication of matrices of size 100×100 may be small for one GPU but large for another GPU. One indirect way to measure the computational capacity of the GPU is to measure the peak floating point performance. Third, the runtime will need to know the dependence relationship between kernels. This is easily solved in VRruntime because each task also specifies the variables potentially read or written by the task.

In order to use multiple GPU queues, we wanted to determine two parameters for each GPU: the number of GPU queues available and the peak floating-point performance of the GPU. We perform a standalone command line tool for estimating these parameters on a given machine. The command line tool performs some micro-benchmarks and writes the guessed values of the parameters in a configuration file, which is later loaded by VRruntime to drive optimizations. These micro-benchmarks for peak floating-point performance and number of queues are discussed in Section 7.4.1 and Section 7.4.2 respectively. The optimization heuristic is described in Section 7.4.3.

7.4.1 Determining peak floating-point performance

We estimate the peak floating point performance of the GPU for a variety of situations: only add operations, only multiply operations, an equal mix of add and multiply operations, and finally purely FMA operations defined in OpenCL. VRcompiler does not use the results of this test currently but RaijinCL uses this test to determine whether or not to generate FMA instructions.

The idea is to execute a kernel that performs a large number of floating point operations and measure the time. We created a template OpenCL kernel where the kernel body has a loop, and the loop-body has multiple independent operations of the required type. The computation is performed purely on scalar variables which can be held in register file. Finally, in order for the OpenCL compiler to not discard the computation, the results are written to an output buffer. However, the number of iterations of the loop-body is kept big enough so that writing to the output buffer has minimal impact on kernel performance. This template is used to generate kernels for testing the floating-point operations listed above, and for both 32-bit and 64-bit floating-point datatypes.

7.4.2 Determining number of hardware queues

GPUs typically consist of more than one SIMD cores. For example, the Radeon HD 8750M GPU used in some of our experiments consists of 6 SIMD cores. Each OpenCL work-group will usually map to a single SIMD core in the GPU though each SIMD core may be able to run more than one work-group. For example, if a kernel is launched with 12 work-groups on the 8750M GPU, then the GPU may distribute 2 work-groups each to each SIMD core. However, if a single kernel execution does not have enough work-groups, then an alternate strategy might be to enqueue multiple kernels in different queues. OpenCL allows creating arbitrary number of command queues for a given GPU device. However, the number of queues implemented in the hardware or the driver may be less than the number of queues created by the application and the driver may map more than one application queue to the same hardware queue. Consider the case where the hardware and driver maintains m hardware queues. If we execute a kernel with a single workgroup, then this kernel execution will only utilize a single SIMD core in the GPU. If we launched n such kernels

where $n \leq m$, each launched on a different queue, then these kernels can run concurrently. Thus, n different SIMD cores on the GPU will be utilized assuming load is balanced fairly between SIMD cores.

With this background, we now consider the problem of determining the number of queues. We consider the kernel previously used for measuring peak throughput of add operations. This kernel is effectively ALU-bound. We execute a single work-group of this kernel and measure the time taken to finish execution of this kernel. Let this time be t_1. We then measure the time taken for n instances of the kernel launched on n different queues, starting with $n = 2$. Let this time be t_n. If $n \leq m$, then the n kernels will execute concurrently and t_n should approximately be the same as t_1. However, if $n > m$, then more time will be taken. Therefore, we start $n = 2$ and keep increasing n until we find that t_n is higher than t_1. Specifically, we use the heuristic $t_n \geq 1.1 \times t_1$ to stop the test. The test reports $n - 1$ as the estimated number of queues.

We used this test on a variety of hardware and found that the test results matched the specification published by AMD and Intel for their GPUs. For Nvidia GPUs, our test determined that only one queue is available even though hardware guides from Nvidia state that more than one queue is available. Our hypothesis is that Nvidia OpenCL driver only exposes or utilizes a single hardware queue.

7.4.3 Automatically using multiple hardware queues

The test for determining the number of queues is run separately (i.e. offline) and the result is written in a configuration file. VRruntime loads this configuration file and if m hardware queues are supported, then VRruntime maintains m OpenCL command queues. OpenCL standard includes both in-order and out-of-order queues but an OpenCL driver is not required to support out-of-order queues and most OpenCL drivers only support in-order queues for GPUs. We therefore only consider in-order queues.

Consider a task given to VRruntime. Each task provided to VRruntime, such as execution of an OpenCL kernel generated by VRcompiler, also carries metadata about the variables that are read and written by the task. VRruntime examines the task and determines if there is a potential data-flow dependence between the given task and the tasks

that are currently already enqueued. If there is no data-flow dependence, then the compiler simply enqueues the task into the queue where the least number of tasks have been enqueued. If there is a data-dependence, then these dependences may actually be originating from multiple kernels enqueued in different queues. We can still enqueue the dependent kernel in any queue because OpenCL allows specification of dependences from any kernel execution in any queue. While OpenCL allows cross-queue dependences, but typically cross-queue dependences create additional overhead in the OpenCL driver and are not very efficient. We use the heuristic that the kernel is enqueued in the queue responsible for the largest dependence. If there is only dependence, then this reduces to enqueing into the queue where the dependence is originating from. In this case, we don't need to explicitly specify any dependences to OpenCL because we are using an in-order queue and thus the dependence is implicitly satisfied by the property of the queue.

Chapter 8
RaijinCL

8.1 Introduction

Matrix operations such as matrix multiply, transpose and reduction operations are the fundamental building block of many important scientific computations. Array-based languages such as MATLAB and Python/NumPy offer built-in operators for these operations. We needed a fast and portable OpenCL implementation of these building block operations in order to support these operators on the GPU. However, prior to this work, no portable and high-performance matrix operations libraries were available for GPUs. Therefore, we decided to design and implement our own GPU matrix operations library called RaijinCL to portably and efficiently support the various operations such as matrix multiplication. While the main motivation for creating RaijinCL was to support Velociraptor, the library is completely standalone and generic, and can be used by any C++ and OpenCL project that needs fast matrix operations on the GPU.

For CPUs, many high-performance libraries for matrix operations are available. For example, various implementations of the BLAS API [27] such as ATLAS[4] or OpenBLAS[31] or the C++ library Eigen are popular choices for CPUs. However, there is no standardized BLAS API for GPUs and creating such a library is also non-trivial. Optimizing OpenCL kernels for matrix operations for a single GPU family is itself a hard problem. Writing a portable library is even harder because kernels tuned for one architecture usually perform

poorly on other architectures. Further, the user may want to distribute the computation over both the CPU and GPU to maximize the system throughput. This is a particularly challenging task on systems that integrate CPU and GPU on the same chip. In such systems, CPU and GPU may share and contend for resources such as caches, memory bandwidth and power budget.

We provide a solution to the problem of portable high-performance GPU matrix operations library. Our OpenCL library RaijinCL automatically generates high-performance implementations of various matrix operations tuned for a particular architecture. The library achieves performance portability through auto-tuning, i.e. it tests various types of kernels and tuning parameters on each compute device in a user's machine, and caches the best found kernels for each compute device. RaijinCL provides auto-tuned implementations for general matrix multiply (GEMM), matrix-vector multiply and transpose. RaijinCL also provides some utility functions such as reductions and element-wise array operations such as array addition and transcendental math operations. These functions are not currently auto-tuned.

The library delivers performance competitive with vendor BLAS on GPUs from multiple vendors. In addition to providing optimized GPU kernels, we have also extended our library to investigate system level throughput for GEMM on single chip CPU/GPU systems. On single chip CPU/GPU systems, we show that our solution significantly outperforms both CPU-only and GPU-only solutions, as well as naive CPU+GPU solutions.

This section is organized as follows. First we outline the contributions of this work in Section 8.2. We then present the general design of the library in Section 8.3. The kernel design and search spaces for GEMM, matrix-vector multiply and transpose are presented in Section 8.4, Section 8.5 and Section 8.6 respectively. We present comprehensive performance results and analysis of kernels generated by RaijinCL for various GPU architectures in Section 8.7. The tuning process for single-chip CPU/GPU systems is presented in Section 8.8 and experimental results are presented in Section 8.9.

8.2 Contributions

We make five contributions in this work.

Kernel design and search-space for auto-tuning: We describe the design and implementation of our auto-tuning library including the set of kernels and parameter space explored by the library for matrix multiply (GEMM), matrix-vector multiply and transpose.

Comprehensive performance evaluation showing good performance on all tested GPUs: We present performance results from a range of architectures from multiple vendors for GEMM. Four variations of GEMM are defined: SGEMM for single-precision GEMM, DGEMM for double-precision, CGEMM for single-precision complex and ZGEMM for double-precision complex GEMM. We present results for all four datatypes.

We show that GEMM routines selected by our library achieve good performance in every case, and are competitive with the vendor's proprietary BLAS on various GPUs. Our study is the most comprehensive cross-vendor empirical comparison of GPU GEMM implementation that we are aware of in terms of vendor coverage, GEMM variation (SGEMM, DGEMM, CGEMM and ZGEMM) coverage and coverage of other matrix operations.

Analysis of GPU kernels on various architectures and compiler issues: In addition to results, we present analysis of several aspects of the performance of the explored kernels on the tested GPUs. Our analysis includes both architectural aspects as well as several compiler issues found on various OpenCL implementations.

Hybrid CPU/GPU GEMM tuning and analysis: We extended our auto-tuner to single-chip CPU/GPU systems. For the hybrid case, our auto-tuner tunes the GPU kernel and the load distribution between CPU and GPU at the same time. We tested single-chip CPU/GPU solutions from Intel and AMD and show that our hybrid approach delivers up to 50% better performance than CPU-only or GPU-only policies. We also show that the kernel tuned for a GPU-only policy is not necessarily the best kernel for the hybrid CPU/GPU case and thus it is not merely a scheduling problem. We also provide some guidelines that are critical to obtaining good performance on single-chip hybrid systems.

Intel Ivy Bridge platform results and analysis: We are the first ones to cover Intel Ivy Bridge GPU architecture from the perspective of OpenCL implementation of matrix operations. We present a brief overview of the architecture and an analysis of our kernels on the architecture. In addition to performance results for both GPU-only and CPU+GPU performance, we also present data about power consumption of various strategies.

8.3 Library design

Designing a library such as RaijinCL requires consideration about performance as well as ease of use. Auto-tuning, discussed in Section 8.3.1, is required for maximizing performance of library operations. However, performance considerations go beyond just the performance of individual operations in the library. Instead, a high-performance library such as RaijinCL has to expose the right application programming interface (API) so that an application programmer can have control over global considerations such as CPU-GPU synchronizations and resource allocation. These considerations are discussed in Section 8.3.2 and Section 8.3.3. Finally, a library should be easy to install and deploy on user's machines without annoying the user and these considerations are discussed in Section 8.3.4.

8.3.1 Auto-tuning

RaijinCL performs auto-tuning, i.e. it generates and tests many different kernels and selects the best one for a given device. Auto-tuning is performed for GEMM, matrix-vector multiply and transpose. For each of these problems, RaijinCL implements a number of parameterized OpenCL kernel code generators called *codelets*. Each codelet represents a particular algorithmic variation of the solution. For example, RaijinCL implements six codelets each for the four data-type variations of GEMM: SGEMM, DGEMM, CGEMM and ZGEMM. For each parameter, we have a predefined list of possible values. An important part of the design of RaijinCL was determining the correct codelets and parameters, so that the right design space is exposed for each kernel.

The auto-tuner searches over all possible combination of parameters for each codelet. The best performing kernel and metadata (such as work-group size) required for execution of the kernel is cached. The tuning process is of the order of one hour to several hours depending upon the machine but only needs to be performed once for each OpenCL device and can be reused many times by any application using RaijinCL.

8.3.2 Asynchronous API

Following OpenCL's design principles, RaijinCL's API is also asynchronous. Computationally heavy API calls in RaijinCL, such as GEMM, perform minimal setup and simply enqueue relevant kernel calls to the OpenCL device. Thus, RaijinCL API calls finish very fast without blocking the CPU and return an OpenCL event object.

8.3.3 Control over resource allocation

Efficient implementation of GEMM routines can require copying the arguments into a more efficient layout into a temporary buffer. Usually the GEMM library implementation will allocate, use and destroy the temporary buffers automatically without exposing them to the application programmer. RaijinCL offers both a high-level API, that is similar to other GPU BLAS APIs such as AMD's OpenCL BLAS, and a low-level API. The high-level API implementation automatically allocates the temporary buffers, performs the appropriate copy or transpose operation and registers a callback with OpenCL to destroy the buffers when the kernel call is complete.

However, there are two performance related concerns with this high-level approach. First, consider a sequence of three matrix multiplication calls $A \cdot B$, $A \cdot C$, and $A \cdot D$. Here, the GEMM library will perform the allocate-copy-destroy operation for A three times which is inefficient. Second, memory available to discrete GPU devices is often very limited and thus the application may want to know the exact memory usage of library routines and may want to explicitly manage the life-cycle of the memory objects allocated by the library. This is difficult to achieve in a high-level API. In the case of our high-level API, OpenCL runtime does not provide a guarantee of the amount of delay between the finishing of the kernel call and execution of the callback to destroy the buffers. Thus, in addition to the high-level API, we offer a four-step low-level API. First, a routine in the low-level API determines the size and type of temporary buffers required for a given input size and layout, and allocates them. The buffers may be reused for multiple problems of the same size and layout. Second, the programmer calls the copy routine. Third, the computation kernels are called. Finally, the temporary buffers can be deallocated. Thus, RaijinCL offers a choice between the convenience of a high-level API and control of a low-level API.

8.3.4 Easy deployment

One issue with auto-tuning libraries is that deployment of the library requires running the auto-tuner on each machine it is installed to. This can lead to a long installation time on each machine and may require many manual steps. Building auto-tuning libraries often requires building from source, when the library tests various alternatives, and doing this on each machine can be a headache for system administrators who prefer distributing binary packages.

We have tried to simplify the installation procedure for RaijinCL. RaijinCL itself can be distributed in binary form and comes with a small command-line utility for performing auto-tuning. The user specifies the device to tune for from the list of OpenCL capable devices installed on his/her machine. The tuning application only requires the OpenCL driver and does not require the user to install a C/C++ compiler. The tuning application generates and tests many different OpenCL kernels, and creates a device profile for the specified device. The device profile contains the best found OpenCL kernels as well as some metadata. The device profile is a simple self-contained text file and can be easily redistributed for deployment by other users of the same device. For example, if many users within the same lab have similar GPUs, the device profile can be generated just once and copied to other machines.

Device profiles for some popular devices like AMD Tahiti (which powers GPUs such as Radeon 7970) are available on our website[1]. If a device profile is already available for your device, then you can simply download the profile and skip the tuning process completely. We are hoping that the community and hardware vendors will contribute many device profiles which will further simplify the deployment process.

8.4 GEMM kernels

We first give some background about GEMM and then describe the kernels and parameters explored by our auto-tuner.

[1] http://www.raijincl.org

8.4.1 GEMM Background

General matrix-multiply (GEMM) computes the operations $C = \alpha op(A)op(B) + \beta C$ where A, B and C are matrices, α and β are scalars. $op(A)$ is either A or A^T, and $op(B)$ is either B or B^T depending on input flags specified by the caller. Our GEMM API supports both row-major and column-major orientations. However, for this thesis, we only consider row-major layouts. We support all four standard datatypes for the elements of A, B and C: single-precision floating point, double-precision floating point, single-precision complex and double-precision complex which correspond to SGEMM, DGEMM, CGEMM and ZGEMM in the BLAS terminology.

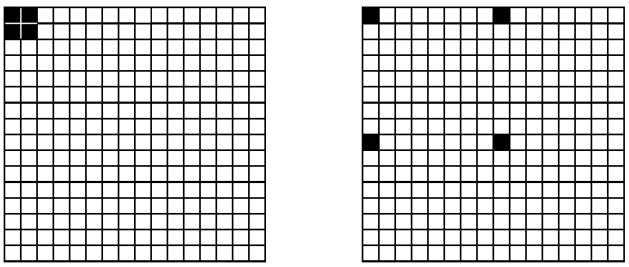
Four variations of GEMM can be considered based on whether the inputs are transposed : $C = \alpha AB + \beta C$, $C = \alpha A^T B + \beta C$, $C = \alpha AB^T + \beta C$ and $C = \alpha A^T B^T + \beta C$. These are called the NN, TN, NT and TT kernels respectively where N corresponds to no transpose and T corresponds to transpose. For square matrices the memory read pattern in TT kernel is very similar to the NN kernel, and we focus on NT, TN and NN layouts only for this paper. Let us assume we have an efficient kernel for any one of the three cases, and we have efficient transpose and copy routine. Then, one need not find efficient routines for the remaining layouts. One can simply transpose or copy the inputs appropriately, and then call the most efficient kernel. However, the memory access pattern for the TN, NT and NN cases can be quite different from each other and the layout found to perform the best on one architecture may not be the best layout on another architecture. Thus, we have implemented three variations of matrix multiplication: TN, NT and NN.

A naive row-major NN matrix-multiply kernel is shown in Listing 8.1. A naive OpenCL implementation will assign computation of one element of C to one work-item. However, such an implementation will make poor use of the memory hierarchy of current compute devices. Thus, typically matrix multiplication is tiled. Each of the three loop directions i, j and k, can be tiled with tiling parameters T_i, T_j and T_k and each work-item is assigned to compute a tile $T_i \times T_j$ of C. This tile is computed as a sum of a series of matrix multiplications of $T_i \times T_k$ and $T_k \times T_j$ tiles of A and B respectively.

```
int i,j,k;
for(i=0;i<M;i++){
  for(j=0;j<N;j++){
     for(k=0;k<K;k++){
    C[i][j] += A[i][k]*B[k][j];
      }
  }
}
```

Listing 8.1 Row-major NN matrix-multiply

Consider a work-group of size W_x, W_y, where each work-item computes a tile of size (T_i, T_j). The work-group will compute a $(W_x \times T_i, W_y \times T_j)$ tile of C. While we have specified the tile size, we have not specified how the tile-elements are assigned to work-items. We have implemented two possible assignments. The first assignment is that each work-item computes a tile consisting of T_i consecutive rows and T_j consecutive columns. The second possibility is that the T_i rows are offset by W_x from each other, and T_j tiles are offset by W_y from each other. We give a visual example. Consider a work-group of size $(8,8)$ where each work-group is assigned $(2,2)$ tile. Then, two possibilities for elements computed by the $(0,0)$ work-item are shown in Figure 8.1.



(a) consecutive elements (b) offset elements

Figure 8.1 Tiles computed by a work-item, consecutive or offset elements

8.4.2 Codelets and search parameters

Our auto-tuner has six codelets for GEMM. Each codelet implements a particular layout (NT, TN or NN) and a particular element assignment scheme (consecutive or offset), thus

giving six combinations. Each codelet has the following parameters:

Local memory usage

Each input matrix can be brought into a workgroup's local memory, which is shared across work-items, or fetched directly from global memory. Thus, there are two possible options for each input, and four possible values for this parameter.

Use of OpenCL images

Storing the input matrices as OpenCL images may be beneficial on some GPU architectures. Again, each input may or may not be stored as an image, and thus there are four possible values for this parameter.

Tile sizes

We explore tile sizes of 2, 4 and 8 for the number of rows and columns of output matrix. However, larger tile sizes increase register pressure. If register usage is more than the number of registers available, the compiler may spill registers which degrades performance substantially. Exploring such slow kernels will unnecessarily increase tuning time. Modern GPUs seem to have about 64 or 128 32-bit registers per thread. Thus, for each codelet, we estimate the register usage and prune out kernels that use more than 128 32-bit registers per work-item. For example, RaijinCL does not explore 8x8 tiles for DGEMM but does explore it for SGEMM. We also tile the inner loop and explore tile sizes of 1,2,4,8 and 16.

SIMD width

All loads and multiply-accumulate operations are done according to the value of this parameter. Our code generator explores SIMD widths of 1, 2 and 4.

Work-group size

The number of work-items inside the group. We currently search for two-dimensional work-groups with 32, 64, 128 and 256 work-items. We chose this set because most current GPU optimization manuals recommend work-group size be 32 or a multiple of 32.

The search space explored by the library for some of the parameters (such as tile sizes, register usage limit or work-group sizes) can be trivially changed by modifying constants in the code. Thus, other researchers are free to extend our experiments.

We have implemented a tool in OpenCL to estimate key hardware parameters. The tool is described in Section 7.4.1. We utilize the estimated number of registers and speed of FMA instructions compared to separate add and multiply to prune the search space for GEMM kernels.

8.5 Matrix-vector multiply kernels

Consider a row-major matrix A of size $m \times n$ and a vector B of size $n \times 1$. We consider the matrix-vector product $C = A \times B$ where C is a $m \times 1$ vector. The matrix-vector multiplication $A \times B$ consists of m vector dot product computations. Each of these m computations is independent.

We have defined one codelet for matrix-vector multiplication which parallelizes the computation as follows. Each dot product computation is assigned to a single one-dimensional OpenCL work-group. Therefore, the computation is divided into m work-groups. Let the size of each work-group be W. The dot product computation is a parallel reduction and we perform it in two phases. In the first phase, we assign the n elements of the vector in a cyclic fashion to the W work-items. So the k-th work item is assigned elements k, $k+W$, $k+2W$ and so on. Each work-item computes the dot product of the elements it is assigned. In the second phase, all the work-items store their results to a local memory array, and the 0-th work item of the work group computes the sum of all the work items. The work-group size W is a parameter of the codelet and it is auto-tuned. The search space for W is currently set to a set of four values: $16, 32, 64, 128$.

8.6 Transpose kernels

RaijinCL implements an out-of-place transpose. We have defined two codelets for transpose, one which uses local memory and one that doesn't. Consider transposing a matrix of size $m \times n$. Both codelets share the characteristic that we create $m \times n$ work-items and each work-item with global ID (i, j) writes the index (i, j) of the output matrix. In both codelets, we create two-dimensional work-groups of work-items and the work-group size in both dimensions is a parameter of both codelets.

The codelet that does not use local memory is a direct translation of the definition of the transpose. Each work-item (i, j) reads element (j, i) of the input matrix and writes it to the index (i, j) of the output matrix. However, in this case two consecutive work-items do not read two consecutive memory locations from the input matrix and this read pattern can be inefficient on some GPUs.

The codelet that uses local memory solves this issue by making consecutive work-items read consecutive elements from memory. Consider a work-group of size $W \times W$. In the codelet that uses local memory, we create a local matrix of size $W \times W$ shared between the work-items of a work-group. The work-group has to read a total of $W \times W$ values from the input matrix in global memory and and write $W \times W$ values to the output matrix in global memory. Instead of reading from global memory in a transposed pattern, the work-group reads the block of $W \times W$ values in a work-item order pattern and writes it to local memory matrix of size $W \times W$ in a transposed pattern. Consider the work-item with local ID (l_i, l_j) in the work-group. This work-item reads the (l_i, l_j) element from within the $W \times W$ block of the input matrix to be processed by the work-group and writes it to the index (l_j, l_i) of the local matrix. Then, after all work-items have finished writing to the local matrix, the work-item writes the index (l_i, l_j) of the local matrix to the (l_i, l_j) index of the $W \times W$ block of the output matrix assigned to the work-group.

8.7 Experimental results and analysis of GPU kernels

8.7.1 Results

To evaluate our auto-tuning library we measured the performance of the kernels across 5 GPUs belonging to AMD GCN, AMD Northern Islands, Intel Ivy Bridge, Nvidia Fermi and Nvidia Kepler architectures. We report percentage of peak attained by RaijinCL and vendor BLAS (where available) on each architecture in the best case for each library in Table 8.1.

Architecture Vendor Type	GCN AMD GPU	N.Islands AMD GPU	Fermi Nvidia GPU	Kepler Nvidia GPU	IB GPU Intel GPU
SGEMM (RaijinCL)	71.8	71.7	69.5	44.1	52.7
SGEMM (vendor)	64.9	66.2	69.0	45.1	N/A
DGEMM (RaijinCL)	83.5	85.3	57.6	91.9	N/A
DGEMM (vendor)	65.5	84.7	60.7	93.0	N/A
CGEMM (RaijinCL)	82.2	65.8	77.9	48.5	60.7
CGEMM (vendor)	65.6	64.5	80.8	51.9	N/A
ZGEMM (RaijinCL)	89.7	85.5	67.5	94.5	N/A
ZGEMM (vendor)	65.7	85.1	66.6	88.3	N/A

Table 8.1 Percentage of peak obtained on each device in best case

The highlights of are our results are:

- We have spanned all recent GPGPU architectures from all three major desktop vendors (AMD, Intel and Nvidia). We have also covered all four GEMM variations (SGEMM, DGEMM, CGEMM and ZGEMM). To the best of our knowledge, this

work presents the most comprehensive study of GEMM covering all major recent desktop GPU architectures.

- The GEMM routine generated by our library outperformed AMD's OpenCL BLAS on both AMD GCN and AMD Northern Islands GPUs.
- Our library achieves about 97% of CUBLAS performance on real variants on both Nvidia Fermi and Kepler, and about 94% of CUBLAS performance on complex variants.
- On Intel's Ivy Bridge GPU architecture, we reached about 52% of peak on SGEMM and 60% of peak on CGEMM, which is higher than some architectures such as Nvidia Kepler. Intel does not provide a BLAS for Ivy Bridge GPU and thus our solution will be particularly important for users.
- Our auto-tuner found different parameter settings for all architectures, but found that TN was the best suited layout for all GPUs.

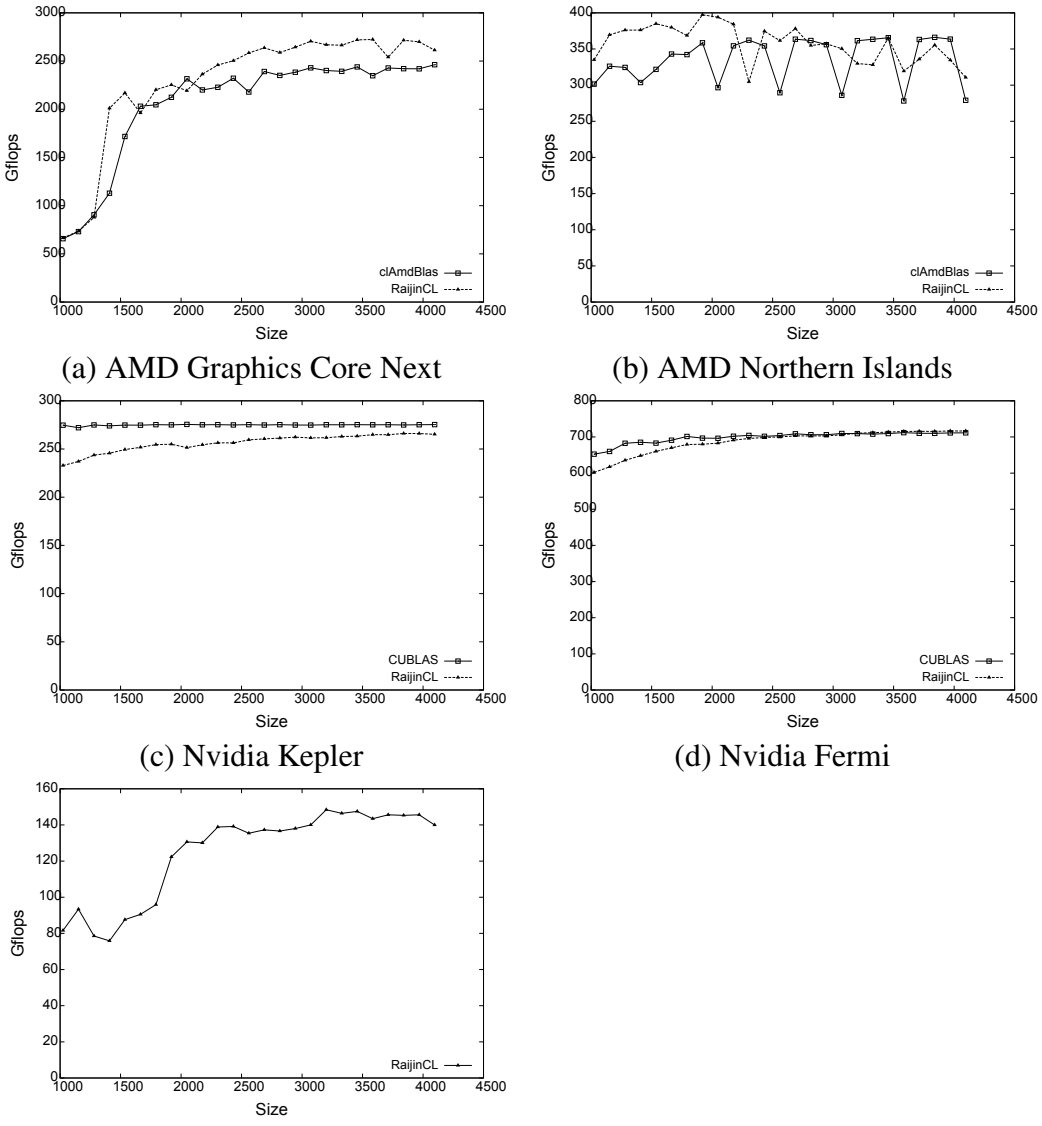
More detailed performance results for SGEMM, showing performance vs problem size, are shown in *Figure 8.2*. The machine configurations are detailed in Table 8.2 and optimal parameters found for SGEMM, DGEMM, CGEMM and ZGEMM are summarized in Table 8.3, Table 8.4, Table 8.5 and Table 8.6 respectively. [2]

8.7.2 Observations and analysis

Best kernels on AMD architectures

On AMD's GCN and Northern Islands GPUs, each work-item in the fastest SGEMM kernels computed an 8x8 tile of the output matrix. Our kernel requires more than 64 32-bit registers for such a tile. Such kernels are not feasible on Nvidia's architectures discussed in this work because they have a limit of 63 registers per work-item. However, AMD's architectures do not have such limitations and thus are able to store such tiles entirely within

[2] We could not include plots of performance vs size for DGEMM, CGEMM and ZGEMM due to space constraints but these are available from our website or upon request.



(a) AMD Graphics Core Next

(b) AMD Northern Islands

(c) Nvidia Kepler

(d) Nvidia Fermi

(e) Intel Ivy Bridge GPU

Figure 8.2 GPU throughput versus problem size for SGEMM

	GCN	N. Islands	Fermi	Kepler	Ivy Bridge
Vendor	AMD	AMD	Nvidia	Nvidia	Intel
GPU	Radeon HD 7970	Radeon HD 8650G	Tesla C2050	Geforce GT 650M	HD 4000
OS	Kubuntu 12.04	Windows 8.1	Ubuntu 12.04	Windows 7	Windows 7
CPU	Intel Core i7 3820	AMD A10-5750	Intel Core i7 920	Intel Core i7 3610QM	Intel Core i7 3610QM
RAM	6GB	8GB	12GB	6GB	6GB
Driver	Catalyst 13.4	Catalyst 13.8b1	304.88	319.20	9.18.10.3071

Table 8.2 Machine configurations. All machines are 64-bit and use DDR3 memory

Architecture Vendor	GCN AMD	N.Islands AMD	Fermi Nvidia	Kepler Nvidia	IB GPU Intel
Layout	TN	TN	TN	TN	TN
Elements are consecutive	No	No	No	No	No
Tile size	(8,8,1)	(8,8,2)	(8,4,16)	(8,4,16)	(8,8,4)
Work-group size	(8,8)	(8,8)	(8,8)	(16,16)	(4,8)
SIMD width	4	4	4	2	4
Bring A to local memory	No	No	Yes	Yes	Yes
Bring B to local memory	No	No	Yes	Yes	Yes
Store A in image	No	No	Yes	Yes	No
Store B in image	No	No	Yes	Yes	Yes

Table 8.3 Optimal parameters for SGEMM

registers. We found that the best performing kernels on GCN do not utilize local memory and do not copy data into images. Previously, some researchers, such as Du et al. [13], had noted that copying data into images is required for a high-performance GEMM implementation on some AMD architectures. The discrepancy between our result and previous research can be explained. Some AMD GPUs (such as Northern islands and previous-generation Evergreen) have a L1 data cache for read-only data which is quite important for performance on GEMM. Previously, this L1 cache was only used for image data but

Architecture Vendor Type	GCN AMD DGEMM	N.Islands AMD DGEMM	Fermi Nvidia DGEMM	Kepler Nvidia DGEMM
Layout	TN	TN	TN	TN
Elements are consecutive	No	No	No	No
Tile size	(4,4,2)	(4,8,4)	(4,4,8)	(4,4,8)
Work-group size	(16,8)	(8,8)	(8,8)	(8,8)
SIMD width	2	2	2	2
Bring A to local memory	No	No	Yes	Yes
Bring B to local memory	No	No	Yes	Yes
Store A in image	No	No	Yes	Yes
Store B in image	No	No	Yes	Yes

Table 8.4 Optimal parameters for DGEMM

Architecture Vendor Type	GCN AMD CGEMM	N.Islands AMD CGEMM	Fermi Nvidia CGEMM	Kepler Nvidia CGEMM	IB GPU Intel CGEMM
Layout	TN	TN	TN	TN	TN
Elements are consecutive	No	No	No	No	No
Tile size	(4,4,2)	(4,4,2)	(4,4,8)	(4,4,16)	(4,4,8)
Work-group size	(8,8)	(8,8)	(8,8)	(8,16)	(8,8)
Complex vector size	2	2	2	2	1
Bring A to local memory	No	No	Yes	Yes	Yes
Bring B to local memory	No	No	Yes	No	Yes
Store A in image	No	No	Yes	No	No
Store B in image	No	No	Yes	Yes	No

Table 8.5 Optimal Parameters for CGEMM

Architecture	GCN	N.Islands	Fermi	Kepler
Vendor	AMD	AMD	Nvidia	Nvidia
Type	ZGEMM	ZGEMM	ZGEMM	ZGEMM
Layout	TN	TN	TN	TN
Elements are consecutive	No	No	No	No
Tile size	(4,4,1)	(4,4,2)	(2,4,8)	(4,2,1)
Work-group size	(8,8)	(8,8)	(8,8)	(16,4)
Complex vector size	1	1	1	1
Bring A to local memory	No	No	Yes	No
Bring B to local memory	No	No	Yes	No
Store A in image	No	No	Yes	No
Store B in image	No	No	No	No

Table 8.6 Optimal Parameters for ZGEMM

improvements in AMD's drivers now enable use of this cache for OpenCL buffers as well.

AMD OpenCL compiler issues

We looked at the assembly code generated by AMD's OpenCL compiler for various kernels, and have identified two issues that may be preventing further performance improvements. First, register allocation is somewhat fragile. Even small changes in the code, such as commenting out unused variables, sometimes produced very different register allocations. Second, we also noticed that AMD's compiler reorders memory loads and does not follow the load order as written in the program. This prevents us from experimenting with the effect of various memory load orders. We are filing a request with AMD to provide a compiler option to disable this feature.

Nvidia Kepler local memory limitations

We found that generally loading both operands into local memory was desirable for both Fermi and Kepler architectures on most GEMM variations. However, on CGEMM, we found that the best kernel for Nvidia Kepler used local memory for only one of the

inputs. Attempting to load tiles of both input matrices into local memory increased local memory requirements per work-group, decreasing the number of active work groups on each Kepler core. Decreased occupancy led to worse performance. Compared to Nvidia's prior generation Fermi, Nvidia increased the number of ALUs from 32 to 192 per core, while keeping the amount of local memory per core (48kB) the same. The limited size of the local memory on Kepler appears to be holding back the ALUs in this case.

FMA instructions

OpenCL supports FMA as a built-in function. On supported architectures, FMA built-in maps to a hardware instruction while some others implement it in software. OpenCL provides a preprocessor macro FP_FAST_FMAF for OpenCL kernels that is defined when the implementation supports a fast FMA operation. However, we discovered that some implementations do define the macro, but performance sometimes varies with datatype (float vs double) or even SIMD width. We now use our estimate of the speed of FMA OpenCL built-in on a given architecture.

Intel Ivy Bridge GPU architecture analysis

The Intel Ivy Bridge GPU (HD 4000) architecture is not well known in high performance computing related literature, and thus we provide a brief description. The HD 4000 GPU consists of two *sub-slices*. Each sub-slice has eight ALUs called execution units (EUs), each of which can perform 16 flops/cycle. Thus, the whole GPU can perform 256 flops/cycle. Each sub-slice has access to 64kB of local memory which is divided into 16 banks and provides 64 bytes/cycle of bandwidth.

OpenCL workgroups are mapped to a sub-slice and each sub-slice runs more than one workgroup. OpenCL workgroups are broken into EU threads, which are analogous to warps on Nvidia architectures and wavefronts on AMD architectures. EU threads are then distributed across EUs in the sub-slice. Each EU can run up to 8 EU threads to hide latencies and increase utilization similar to AMD and Nvidia architectures. However, unlike say Nvidia architectures with a fixed warp size, the number of work-items in a EU thread is kernel-dependent. Each EU thread has access to a 4kB register file. The compiler maps

either 8, 16 or 32 work-items to one EU thread depending upon register usage of each work-item. We estimate that the best found kernel by RaijinCL for SGEMM on Ivy Bridge is using more than 64 registers, making it impossible to map 16 or 32 work-items to one EU thread. Thus, likely 8 work-items of our kernel are mapped to each EU thread and our work-group (32 work-items) is divided into 4 EU threads. For full occupancy, we need 64 EU threads (or 8 work-groups in our case) per sub-slice. Our kernel is using less than 2kB of local memory per workgroup, and thus there is enough local memory to run more than 16 work-groups and there should be enough EU threads to fully occupy all EUs.

RaijinCL achieved about 52% of peak on SGEMM (Figure 8.2(e)) and 62% on CGEMM. Our estimate is that there is enough local memory bandwidth to achieve about 85% of peak on our SGEMM kernel, compared to 52% achieved, and the bottleneck is likely elsewhere. We performed an additional experiment to see if global memory access is one of the bottlenecks. We removed all global memory loads from the auto-generated kernel and replaced the load value with just the address computation. The SGEMM performance improved to about 60% of peak, which supports the hypothesis that global memory access is indeed one of the bottlenecks.

8.8 Tuning for single-chip CPU+GPU systems

In the previous section, we looked at performance of GPUs in isolation to the rest of the system. In this section, we extend the discussion to system-level performance. System level performance considerations include performance of both the CPU and GPU, as well as data transfer and synchronization overheads between the CPU and the GPU. In single-chip CPU+GPU solutions such as Intel's Ivy Bridge and AMD's Richland APUs, the CPU and the GPU may share resources such as the memory controller and caches. Such chips also integrate sophisticated power management techniques where the CPU may boost beyond its boost clocks on CPU-heavy workloads when the GPU is idle, and the GPU may boost on GPU-heavy workloads when the CPU is idle.

We want to utilize both the CPU and the GPU where possible. Consider an output matrix C of $M \times N$ elements. We logically partition the M rows into P parts of M/P rows each. We assign the computation of p_g parts out of P to GPU, and p_c parts out of P to CPU.

Here $p_g + p_c = P$ and thus determining p_g automatically assigns p_c for a fixed value of P. The value of p_g chosen affects the net system performance. For a given value of p_g and P, and a given auto-generated GPU kernel, we follow the following algorithm for computing an output matrix C of size $M \times N$.

1. Copy the relevant input data to GPUs if required. The creation of GPU input buffers can be optimized on some single-chip systems that allow creation of GPU buffers without data copy.

2. Enqueue a call to the provided GPU kernel to compute $M * p_g/P$ rows of C.

3. Flush the GPU command queue to force the OpenCL runtime to begin execution of the kernel.

4. Call the CPU BLAS to compute $M * (P - p_g)/P$ remaining rows of C.

5. Enqueue a copy of data from the GPU to the final output matrix C. We are using in-order queues, so this call will only execute once the GPU computation is finished thus ensuring correctness.

6. Wait on the GPU command queue to finish execution of all GPU operations.

7. Delete all GPU buffers and free any other temporary resources.

In such a system, the net system performance depends on the selected GPU kernel as well as the parameter p_g. We have fixed P to eight in our system as it offers a good balance of search space granularity and tuning time. We wanted to determine the best possible GPU kernel and p_g parameters in a hybrid setting. The GPU kernel that provides the best performance when considering GPU performance in isolation may not be the best kernel when considering whole system performance. Thus, the tuning for hybrid CPU/GPU GEMM is done separately from the GPU-only case. The tuner for the hybrid CPU/GPU case has two differences from the GPU-only auto-tuner we have discussed so far. The first difference is that the hybrid tuner extends the search space with one more parameter: the number of parts p_g assigned to the GPU out of P. The second difference is that while the

single-GPU auto-tuner optimizes for execution time of the kernel on the GPU, the hybrid tuner optimizes for net system throughput.

There are two assumptions in our current algorithm that allow us to eliminate data copies from the CPU to the GPU on the Ivy Bridge platform. First, we assume that the input matrices are in TN-layout. Second, we assume that the input matrices are aligned to 128-byte boundaries. The alignment restriction allows the creation of an OpenCL buffer from host memory without data copy on Ivy Bridge platform. We expect that such alignment restrictions will not be required on future platforms. For example, the upcoming AMD Kaveri platform will allow completely unified addressing of memory and will not require any data copies irrespective of the alignment.

8.9 Results for single-chip CPU+GPU systems

We experimented with Intelś Ivy Bridge platform and AMDś Richland platform (A10-5750M) with Northern Islands GPU. The machine configurations are detailed in Table 8.2. We compared four strategies:

1. CPU-only using vendor-provided CPU BLAS
2. GPU-only using previously obtained kernel tuned for GPU-only scenario
3. CPU+GPU using previously obtained kernel tuned for GPU-only scenario and tuning the load distribution
4. Full CPU+GPU tuner where both GPU kernel and load distribution are tuned together

We first discuss the performance results on both platforms, and then provide power analysis on the Ivy Bridge platform.

8.9.1 Performance

Performance results are shown in Table 8.7. On the Ivy Bridge platform, our full tuner outperformed the CPU-only, GPU-only as well as the simple CPU+GPU tuner. We found

that the GPU kernel optimized for the GPU-only case was completely different than the kernel optimized for the CPU+GPU case on Intel Ivy Bridge showing that the full co-tuning of GPU kernels and load distribution is indeed required. Optimal GPU kernels for the hybrid case and the load distribution is summarized in Table 8.8.

On the AMD Richland platform, we found that SGEMM was best performed on the GPU alone instead of using both the CPU and GPU. Thus, resource constraints and other overheads outweigh the benefits of a hybrid strategy in this case. This is unsurprising given the imbalanced ratio of CPU/GPU performance on SGEMM on this platform. DGEMM, where the CPU/GPU ratio is more balanced, shows that hybrid strategy outperforms CPU-only or GPU-only solutions.

Platform	Ivy Bridge	Richland	
Vendor	Intel	AMD	
Type	SGEMM	SGEMM	DGEMM
CPU-only	170	80	40
GPU-only	140	274	27.3
CPU+GPU (simple tuned)	175	274	57.4
CPU+GPU (fully tuned)	235	274	57.4

Table 8.7 Hybrid CPU+GPU tuning results (Gflops)

8.9.2 Power analysis on Ivy Bridge

In single-chip systems such as Intel Ivy Bridge, the chip specifies a maximum power draw. For example, the chip used in this test has a 45W thermal design power (TDP) though it can go slightly above its TDP for short bursts. The sum of the CPU power-draw and the GPU power-draw is not allowed to exceed the specified maximum power draw. In order to maximize performance, platforms such as Ivy Bridge implement dynamic power distribution schemes where the power is distributed between the components depending on the workload. In CPU-heavy workloads, more power is diverted to the CPU while in GPU-heavy workloads more power is diverted to the GPU.

We wanted to understand the relationship of power distribution between components and performance. Intel provides a tool called system analyzer that records power measure-

	Intel Ivy Bridge	AMD Richland	
Type	SGEMM	SGEMM	DGEMM
Layout	TN	TN	TN
Elements are consecutive	No	No	No
Tile size	(8,4,8)	(8,8,2)	(4,8,4)
Work-group size	(8,16)	(8,8)	(8,8)
SIMD width	4	4	2
Bring A to local memory	Yes	No	No
Bring B to local memory	Yes	No	No
Store A in image	No	No	No
Store B in image	No	No	No
GPU load (fraction)	10/16	16/16	8/16

Table 8.8 Tuned parameters for GPU kernel and load distribution for hybrid CPU+GPU GEMM

CPU load (fraction)	GPU kernel	Perf. Gflops	Socket power Watts	CPU Power Watts	GPU Power Watts	Perf. /socket power Gflops/Watt
1	CPU-only	170	40	35.9	0.7	4.25
0	GPU-optimized	140	29	2.4	22.6	4.83
10/16	GPU-optimized	185	50.5	28.3	17.5	3.66
0	Hybrid-optimized	110	24.3	2.3	18.6	4.52
10/16	Hybrid-optimized	235	51.4	32.6	14.3	4.57

Table 8.9 Measured peak power for various GPU kernels and CPU/GPU load distributions

ment reported by the chip. We used this tool to measure the socket, GPU and CPU power consumption. The socket power consumption is the sum of power consumption of the CPU, the GPU as well as some common parts of the chip such as the memory controller. The measured peak power consumption of each strategy are provided in Table 8.9. The data confirms that Ivy Bridge has a very dynamic power distribution scheme. Under CPU-only GEMM, the CPU drew up to 36W of power while the GPU was nearly idle. Under GPU-only GEMM, the GPU drew up to 23W of power while the CPU was nearly idle. However, when subjected to the hybrid load, our results show that the chip's power controller decided to throttle both the CPU and the GPU back. Under hybrid load, the power draw of each component is capped by the hardware to about 70-80% of the peak power draw of each component in order to remain under the TDP. Thus, we conclude that in hybrid solutions like Ivy Bridge, the hybrid peak performance cannot be expected to reach the sum of peaks of individual components because under hybrid workloads each component is operating under stronger power constraints compared to when the component is acting alone.

8.9.3 Observations and performance guidelines

We have observed two important considerations when optimizing performance on single-chip systems.

Explicit flushing of GPU queue

Explicitly flushing the GPU queue before the CPU kernel is called is extremely important. Without manual flushing, the OpenCL runtime from both Intel and AMD appears to not immediately begin GPU execution and this can reduce performance by as much as 50%.

Effect of callbacks

It is also important to not register any callbacks with the OpenCL runtime. In the initial version of the library, we had registered callbacks with OpenCL to automatically delete GPU buffers when the GPU finishes execution. However, we noticed that upon registering callbacks, the OpenCL runtime was waiting in a loop and consumed significant CPU time. The increased CPU load affected the CPU BLAS performance. The increased CPU load

also led to increased CPU power consumption, and the chip often reduced the GPU clocks to remain within limits. Thus, registering callbacks with the OpenCL runtime reduced performance significantly and now we do not register any callbacks with OpenCL.

8.10 Conclusions

Developing portable and fast matrix operations library for GPUs is an important and challenging problem to solve. Even though OpenCL kernels are theoretically portable, different devices require different OpenCL kernels in order to achieve high performance. In this chapter we presented a solution to this problem via the design, implementation and evaluation of RaijinCL, a portable and high-performance auto-tuning OpenCL library for GEMM and other matrix operations. In addition to performance, we have also considered many practical issues such as easy deployment and API design issues around asynchronous dispatch and GPU memory usage.

Our auto-tuning approach was designed by identifying a collection of codelets for each kernel. Different versions of the codelets expose important algorithmic variations. For example, our GEMM codelets expose the argument layout (transposed or not) and the element assignment scheme (consecutive or offset). Within each codelet we identified important parameters such as tile sizes, SIMD width, work-group size, how to handle local memory and whether or not to use OpenCL images. Given the group of codelets and the parameter space, the auto-tuner evaluates all points in the search space and identifies the best codelet, and the best parameters for that codelet.

We experimented on a wide variety of GPU architectures from multiple vendors including AMD, Nvidia and Intel. For each device we compared our auto-tuned library to the vendor's specialized library (when one was available). We found that we sometimes outperformed the vendor's library, and at other times we were close to vendor libraries performance. Thus, we did achieve the goal of having a portable and high-performance solution across a range of GPUs.

In addition to our performance results, we discussed a number of issues about architecture and OpenCL implementations. For example, we provided some analysis of kernels on AMD and Nvidia architectures. We also discussed compiler issues on various platforms

such as register allocation and use of FMA instructions that can have significant impact on GEMM performance.

We also extended our auto-tuner to optimize system-level performance on single-chip CPU/GPU systems where the workload is distributed over both the CPU and GPU. We showed that the GPU kernels optimized for GPU-only GEMM are sometimes different than the GPU kernels optimized for hybrid CPU/GPU GEMM implementation. We found that our auto-tuning approach successfully generated kernels and load distribution policies that can outperform CPU-only, GPU-only and even naive CPU+GPU tuning policies. We also provided guidelines for carefully managing CPU-GPU synchronization based on our experience in developing RaijinCL.

We provided an overview and analysis of the Intel Ivy Bridge GPU architecture, which has not previously been described in the academic literature. We also presented results on power usage of various CPU and GPU solutions on Ivy Bridge and discussed how power management schemes affect the performance achieved in hybrid workloads compared to CPU-only or GPU-only workloads.

To summarize, RaijinCL is a fast, portable and practical matrix operations libraries for GPUs. This work has several novel contributions. We described the search space and kernel design for auto-tuning kernels for various matrix operations. We presented comprehensive experimental results and analysis on various GPUs. This thesis is also the first work to consider the issue of, and propose and evaluate solutions for, maximizing system throughput on GEMM on single-chip CPU/GPU system.

In the larger picture, RaijinCL plays an important role in the thesis. Library operations such as matrix multiplication are an important part of array-based languages and RaijinCL was written to enable Velociraptor to support these operations on the GPU. RaijinCL also fits well with the theme of portability and performance that we see in the rest of the thesis.

Chapter 9
Two case-studies

We have claimed that Velociraptor is a reusable and embeddable toolkit that is not tied to a single toolchain and can be used by compilers for different languages. In order to prove that Velociraptor is reusable, and that it can be embedded with little effort in existing language implementations, we have built two case studies of integrating Velociraptor. The first case-study is McVM, an existing JIT-compiled implementation of the MATLAB language. The second case-study is a prototype JIT compiler extension to the reference Python interpreter CPython. The two case-studies are quite different. In the first case study, we are extending an existing JIT compiler while in the second case study we are adding a JIT compiler to an interpreter. The case-studies also target two different languages and two different memory management semantics.

We first describe the McVM and Python case studies in Section 9.1 and Section 9.2 respectively. We conclude the chapter with some remarks about the development experience of using Velociraptor in Section 9.3. We describe our development experience in building these two case-studies and also briefly describe a project by another researcher who is using Velociraptor.

9.1 McVM integration

McVM is a virtual machine for the MATLAB language. McVM is part of the McLAB project, which is a modular toolkit for analysis and transformation of MATLAB and ex-

tensions. 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. We added support for parallel loops and GPUs by introducing new language constructs detailed in Section 9.1.1. In *Chapter 3*, we describe that a compiler writer must provide a pass to identify which sections to compile using Velociraptor, and also needs to provide glue code. We provide both of these pieces for McVM. The compilation strategy of integrating Velociraptor into McVM is detailed in Section 9.1.2.

9.1.1 McVM language extensions

We added one new keyword and three new *magic functions* to McVM for parallel loops and GPU sections. Magic functions are functions that look like empty function calls to other MATLAB implementations but are recognized as special by McVM. Magic functions allow us to add functionality to the language without extending the grammar, and without breaking compatibility with other implementations.

First, we added the *parfor* keyword for specifying parallel loops. *parfor*-loop is similar to MATLAB *for*-loop except that each iteration is declared to be independent. MathWorks MATLAB also provides a *parfor*-loop but their specification and implementation of the construct is different than ours. We first describe our implementation, and then describe their implementation and differences.

We require that the user declares the list of variables that are shared by all iterations of the parallel-loop. These variables are declared by calling a magic function provided by us, called *mcvm_shared* that takes as input variables that are shared across all threads. The call to *mcvm_shared* should be the first statement of the parallel-loop. For example, if variable *A* and *B* are shared across iterations, then the user will call *mcvm_shared(A,B)* as the first statement of the loop-body. All other variables occurring inside the loop are automatically assumed to be private to each thread.

In MathWorks MATLAB, they spawn multiple worker processes and dispatch different iterations of the loop amongst the workers. They place a number of restrictions on the language constructs usable inside the loop, presumably to simplify the work of identifying the

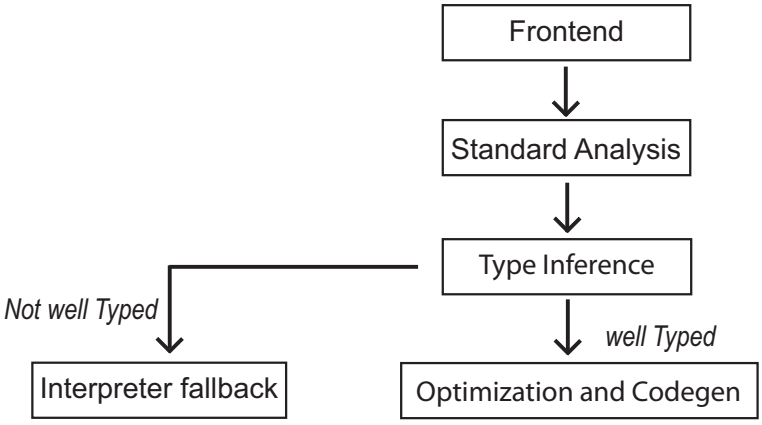
data to be copied to/from worker processes. They require that array indexing expressions occuring inside the parfor-loop should either consist of expressions involving variables not set within a loop (i.e. loop invariant variables), or only be of the form $i+k$ where i is the loop-index and k is a loop-invariant expression or constant. Unlike our implementation, MathWorks MATLAB automatically infers shared variables and private variables based on variable classification rules specified in their documentation. For example, any variable that is the target of a non-indexed assignment expression is a private variable. We have made the design decision to make shared variable declaration explicit because explicit declarations should make user code simpler to understand both for users and compilers.

We provide support for specifying GPU sections. We have provided *gpu_begin()* and *gpu_end()* magic functions that act as section markers and indicate to the compiler that the section should be offloaded to the GPU.

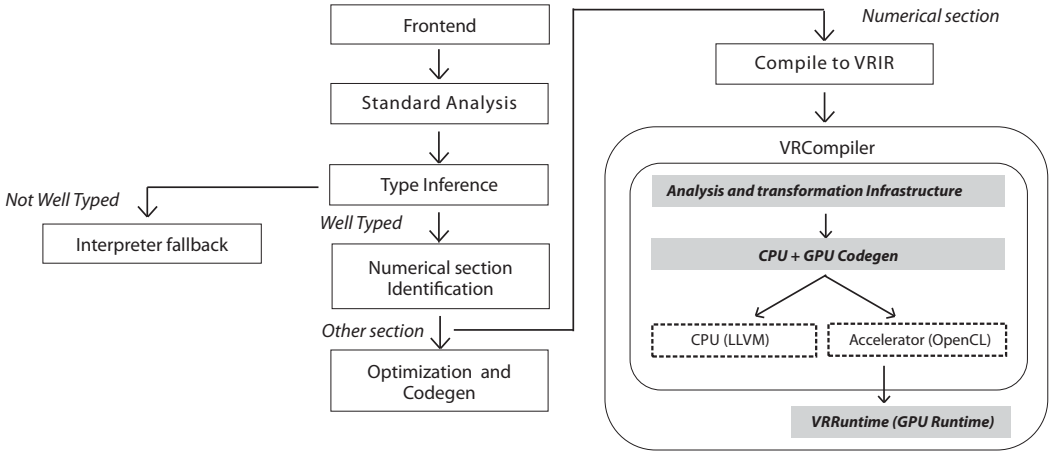
9.1.2 Integrating Velociraptor into McVM

We have extended McVM to integrate Velociraptor. A simplified overview of McVM before and after integrating Velociraptor is shown in *Figure 9.1*. The general condition for compiling any code to VRIR is that the types of all variables and expressions inside the function should be known and mappable to VRIR types and any library calls occurring inside the code should be mappable to VRIR library functions. Also, if the code calls a user-defined function, then that function should also be compilable to VRIR.

We have implemented a number of simple heuristics to automatically identify sections of code to be compiled to VRIR and handed over to Velociraptor. These heuristics are called after McVM has finished type inference. The first heuristic is to identify functions that can be completely converted to VRIR. If the body of a function satisfies all the constraints given above, then the function is compiled to VRIR. If the first heuristic is not applicable to a given function, then McVM looks for parfor-loops and GPU sections in the code. If the body of the parfor-loop or the GPU section satisfies the constraints above, then these are outlined into a separate function and compiled to VRIR. Otherwise, we remove the GPU annotations and also convert any parallel-loop to serial for-loops in any code that cannot be compiled to VRIR. The final rule is that if there is a loop that calls a function



(a) Original McVM



(b) McVM with Velociraptor integrated

Figure 9.1 McVM before and after Velociraptor integration

foo, and if foo will be compiled VRIR, and if the body of the loop satisfies the typing constraints above, then the loop is outlined and compiled to VRIR. It is important to capture such loops and compile to VRIR, because it allows us to communicate to Velociraptor and it's specialization infrastructure that the function foo may be called inside a loop.

Overall, integrating Velociraptor required us to implement the numerical section identification heuristics, code generator from McVM's IR to VRIR and outlining infrastructure. We have also provided glue code for exposing McVM's internal data-structures to Velociraptor. This required providing a McVM-specific implementation of Velociraptor's abstract

memory allocation APIs, telling Velociraptor about MATLAB language semantics (such as using one-based indexing), exposing LLVM representation of various array classes and providing macros and functions for field access of array class members.

9.2 Python integration: PyRaptor

We have written a proof-of-concept compiler called PyRaptor for a numeric subset of Python, including the NumPy library, that integrates with CPython, the standard Python [33] interpreter. PyRaptor provides a number of language extensions to Python language and these are described in Section 9.2.1. The design of PyRaptor and how it glues together CPython and Velociraptor are described in Section 9.2.2.

9.2.1 Language extensions

We have provided several extensions to the Python language. However, we did not want to modify CPython's inbuilt lexer and parser. Therefore, all syntax extensions are done via language's inbuilt code annotation mechanisms or via magic functions. As an example of our language extensions, we show a pure Python version of a stencil operation over 2D matrices in *Figure 9.2a* and the same example with our extensions is shown in *Figure 9.2b*. We discuss the provided extensions in the next subsections.

Compilation extensions

As described in *Chapter 3*, any implementation needs to identify and outline numerical sections. The idea is that only a few numerical functions in a Python program are compiled, while the rest of the program is interpreted by the Python interpreter. In our prototype, we require that the programmers outline suitable numerical sections into functions manually.

Once suitable numerical functions are identified, the programmer adds an annotation to these functions that tells PyRaptor to compile these functions. This is done conveniently via a Python construct called *decorators*. A decorator is a function or method that takes the function being decorated as input, performs some processing, and returns another function as output. The function being decorated is replaced by the output of the decorator. Thus, if

```
#An example showing a simple stencil operation over a 2D array
   with periodic boundary
def stencil(A,B)
  m,n = A.shape
  for i in range(m):
    for j in range(n):
      B[i,j] = (A[i,j] + A[i-1,j] + A[i+1,j] + A[i,j+1] +
          A[i,j-1])/5
```

(a) Stencil example in plain Python

```
#Create a VRIR module
stenmodule = PyRaptorModule('stencil')

#add the compile decorator and type annotations
@stenmodule.compile
def stencil(A: PyArray(PyFloat64(),2,'RowMajor'),
  B: PyArray(PyFloat64(),2,'RowMajor'))
  m,n = A.shape

  #declare a block of code to be run on GPU
  gpu_begin()

  #parallel loop over a 2D domain instead of serial nested loops
  #A,B declared as shared variables in the loop
  for i,j in PAR(Domain(m,n),shared=[A,B]):
    B[i,j] = (A[i,j] + A[i-1,j] + A[i+1,j] + A[i,j+1] + A[i,j-1])/5

  gpu_end()

#code will now be compiled, and stencil will get replaced by
   compiled version automatically
stenmodule.compile()
```

(b) Stencil example with our language extensions

Figure 9.2 Stencil example in plain Python and with our extensions

a function f is decorated with a function g, then the function f is replaced by the Python interpreter by the result of the call $g(f)$ at module loading time. Decorators provided by PyRaptor allow dynamic replacement of functions being compiled with the compiled function dynamically generated by PyRaptor.

Type annotations for functions

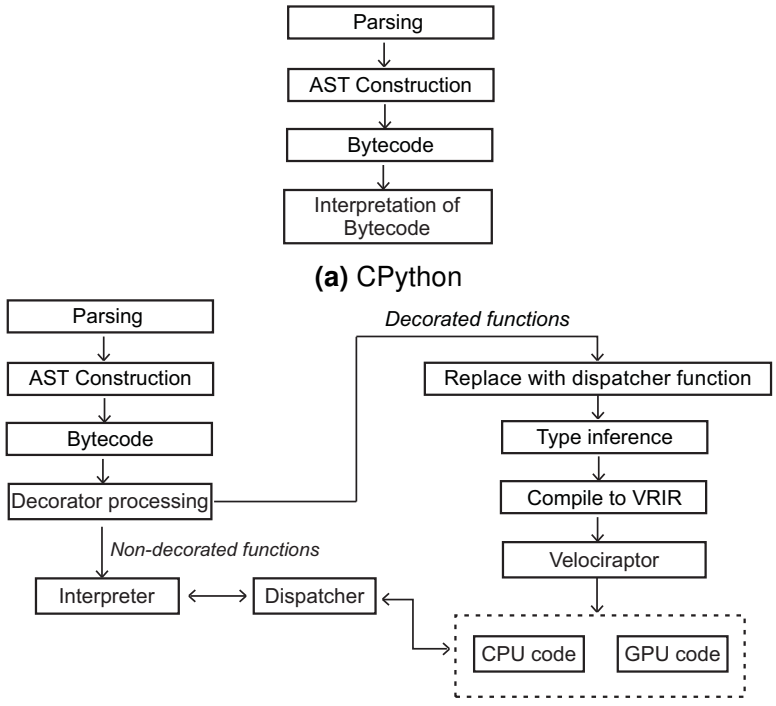
The programmer then adds type signatures to the function. Python 3 provides convenient function annotation syntax to add meta-data to function parameters and return values. We provide some special classes that represent the type of the parameters in VRIR and programmer adds the type information via the function annotation syntax. We have shown the example of the syntax in *Figure 9.2b*.

Domains

We have provided a Domain class to conveniently represent iteration over multi-dimensional domains. Python provides for-loops that iterate over arbitrary iterator objects. A common idiom in Python is to iterate over range objects, which are simple one-dimensional ranges with a start, a stop and a step value. We provide a class called Domain, which represents multi-dimensional ranges.

Parallelism and GPUs

PyRaptor defines a special function *PAR* that can be used to declare parallel for-loops. It takes as input a domain to iterate over, and a list of shared variables. Each iteration of the parallel for-loop can execute independently. The semantics is similar to those defined for VRIR with some additional restrictions imposed due to limitations of CPython. CPython APIs were not designed to be called from multiple threads, and therefore we do not allow any operations that require CPython API operations. This leads to the restriction that operations that allocate new objects are not allowed because memory allocation for Python objects is done via CPython API. Finally, we also provide constructs (*gpu_begin()* and *gpu_end()*) that act as markers to annotate blocks of code to be executed on the GPU.



(a) CPython

(b) CPython extended using PyRaptor

Figure 9.3 Integration of PyRaptor with CPython

9.2.2 PyRaptor: Design, components and glue code

PyRaptor is implemented as an *extension module* for CPython version 3.2 and above, and does not require the user to modify the installed CPython or NumPy. An extension module is a Python library written in as C or other compatible languages that utilizes the CPython C API. Extension modules are compiled to dynamically loaded libraries (DLLs) which can be loaded by the CPython interpreter using the specified APIs.

PyRaptor is a two-phase compiler consisting of a type inference phase and a code generation phase. In the type inference phase, PyRaptor first calls a Python standard library module called *ast* which provides functionality to construct an AST from Python source code. PyRaptor uses this module to construct untyped ASTs from annotated functions. Then PyRaptor performs type inference on this AST by propagating the types of the function parameters provided by the user into the body of the function. We require that the type of a variable not change within a function. PyRaptor can handle scalar numerical datatypes,

NumPy arrays and tuples. Many of the commonly used constructs are handled including for-loops, while-loops, conditionals, various array indexing and slicing operators as well as many math library functions for arrays.

In the second phase, PyRaptor compiles annotated functions to VRIR and then calls Velociraptor to generate the CPU and GPU code. Once a Python function is compiled, PyRaptor replaces the original Python function with a dispatcher function. The dispatcher takes as arguments the arguments of the Python function it is replacing, as well as the function ID of the generated function. The dispatcher converts the Python objects received as arguments of the function into their C counterparts and calls the compiled function.

PyRaptor also provides all the glue code necessary to interface Velociraptor with the necessary data structures in CPython and the NumPy library. This includes glue code 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.

9.3 Development Experience

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

Overall, we found that using Velociraptor enabled some saving in the development time of both compilers. As an indirect estimate, we can look at the number of lines of code required. For McVM, the integration required about 3000 lines of code. Our Python compiler (including type-inference) required only about 4000 lines of code. The tools provided by us, Velociraptor and RaijinCL, together are more than 20,000 lines of code. Further, some of the implementation details of Velociraptor are quite subtle, and require effort much beyond what the lines of code metric suggests. Thus, we can reasonably claim

that the effort of integrating Velociraptor is far less than the amount of functionality and code implemented in our tools.

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

Other researchers have also started using the toolkit. A student in our lab, Sameer Jagdale, is working on a project to statically generate C++ from MATLAB and Python functions where the programmer has added type annotations. The generated C++ can then be compiled into a dynamically loadable library (DLL) that can be used from MathWorks MATLAB and CPython respectively. In order to support two languages, he is using parts of Velociraptor and has added a static C++ backend to VRcompiler. He first statically compiles MATLAB and Python to VRIR and then uses his new static back-end to generate C++ from VRIR. He is reusing many of the passes in VRcompiler, including simplification, loop information collection and preliminary bounds-check elimination. However, given that he is writing a static compiler generating C++, he is not using our LLVM and OpenCL backends or dynamic techniques such as region specialization. Apart from VRcompiler, he is also reusing most parts of the Python tool-chain presented in this chapter. The tools described in this thesis have saved some of his development effort in two ways. First, he has been able to reuse some of our code. Second, he has saved some development effort by compiling both languages to a common IR (VRIR) because he only had to implement one back-end that can be used to generate code for both languages.

Thus, to conclude, our McVM and Python case studies demonstrate that Velociraptor can indeed be used for different languages and different compilers. Our toolkit is also starting to be used and extended by other users in new ways. A common IR for array-based languages and a reusable compiler framework may enable more researchers to quickly build new compilers for such languages.

Chapter 10
Compiler and runtime evaluation

We evaluated the performance of Velociraptor on a variety of benchmarks for MATLAB and Python representing a variety of numerical computations. We evaluated both CPU and GPU performance on multiple machines, and also evaluated the effect of various compiler and runtime optimizations on the performance. This performance evaluation was done against currently available state-of-the-art compilers for these languages. The benchmarks used in the study are described in Section 10.1 and the system setup is given in Section 10.2.

We answer the following questions in the experiments:

1. How does Velociraptor generated serial CPU code compare with existing compilers for array-based languages? We answer this question in Section 10.3.

2. Do parallel-loop constructs supported by Velociraptor lead to any speedup? We answer this question in Section 10.4.

3. How does Velociraptor generated GPU code compare with parallel CPU code generated by Velociraptor? We answer this question in Section 10.5.

4. What is the effect of region specialization on performance? Does region specialization require frequent recompiles? What is the accuracy of region shape inference? These region specialization related questions are discussed in Section 10.6.

5. Is automatic use of multiple hardware queues by Velociraptor beneficial? We answer this question in Section 10.7.

We present a summary of the results in Section 10.8.

10.1 Benchmarks

In order to comprehensively evaluate the performance of Velociraptor, we wanted a representative set of benchmarks covering many different array computation patterns. We looked at benchmarks used by previous researchers such as those used by McVM project and those contributed by the Python community. However, in the high performance computing community, researchers have been looking at classification of important algorithmic patterns in numerical computing. These patterns have been called *dwarves of parallel computing*. We also cover some benchmarks derived from benchmark suites that supply representative samples of each dwarf.

10.1.1 McVM benchmarks

The main use-case for Velociraptor is in numerical computations where the containing compiler has good type information. Therefore, we looked at benchmarks used by previous researchers for evaluating McVM performance and selected the benchmarks where McVM was very successful in inferring the types. We then added parallel-for and GPU annotations to the benchmarks where applicable. No other change was made to the benchmarks. The benchmarks are as follows:

nb1d: nb1d is a n-body computation that simulates the gravitational movement of a set of objects.

clos: Closure, abbreviated as clos, computes the transitive closure of a directed graph. The performance critical part of the computation consists of a a sequence of dense matrix multiplications.

fftm: This consists of a fast fourier transform algorithm written in MATLAB.

fiff: This benchmark solves a 2D partial differential equation using finite-difference method.

capr: This benchmark computes capacitance of a transmission line using finite-difference method and Gauss Seidel iteration.

fdtd: This benchmark solves a partial difference equation using Finite Difference Time Domain (FDTD) technique.

10.1.2 Dwarf based Python benchmarks

We have ported some dwarves from other benchmark suites to Python.

lud: LUD represents the dense linear algebra dwarf. It performs LU decomposition and it is derived from OpenDwarves suite.

spmv: SpMV represents the sparse linear algebra dwarf. It performs sparse matrix-vector multiplication and it is derived from the OpenDwarves suite.

srad: SRAD represents structured grid computations dwarf. However, somewhat unusually for a structured grid, it includes some indirect accesses through index arrays.

pagerank: Pagerank represents the map-reduce dwarf and is inspired from Google's pagerank algorithm.

fft: FFT represents a fast fourier transform dwarf. It is written as a recursive function and cannot be offloaded to GPUs targeting OpenCL 1.2 due to the recursive function calls.

10.1.3 Python NumFocus benchmarks

These benchmarks come from a benchmark suite put together by the open-source community around Python compilers. The benchmark suite can be found online[1]. The benchmarks are hosted under a repository run by NumFocus community and hence we will refer to them as Python NumFocus benchmarks.

growcut: Performs image-segmentation using a 2D window function.

julia: Computes the Julia fractal.

pairwise: Computes the pairwise distance between a set of points in 3D space.

[1] https://github.com/numfocus/python-benchmarks

10.2 Experimental setup

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

1. Machine M1: Core i7 920, Tesla C2050, 6GB DDR3, Ubuntu 12.04, Nvidia driver version 280.13, CPython 3.2, MathWorks MATLAB R2013a 64-bit, Cython 0.21.

2. Machine M2: AMD A10-5750M with integrated GPU AMD Radeon HD 8650G, discrete GPU Radeon HD 8750M, 8GB DDDR3, OpenSUSE 13.1, Catalyst 14.2, CPython 3.2, Cython 0.21.

The first machine has a single discrete GPU (Tesla C2050), while the second machine has an integrated GPU (Radeon 8650G) as well as a discrete GPU (Radeon 8750M). These three GPUs belong to three different architectural families. The CPUs in both machines also belong to very different CPU architecture families. We used CPython v3.2 and Cython 0.21 for Python benchmarks. For MATLAB, we used MATLAB R2013a 64-bit in tests. We did not have a suitable MATLAB license for machine M2 and therefore performance was only compared to McVM in this case.

Each test was performed ten times and we report the mean. All reported execution times for Velociraptor and McVM include the JIT compilation time.

10.3 Serial CPU performance against baseline compilers

In this section, we look at performance against baseline compilers with all Velociraptor optimizations turned on. For investigating CPU performance, we configured Velociraptor to disable GPU code generation and everything was executed on the CPU. We also disabled parallel-for support in Velociraptor. In this mode, Velociraptor simply converts all parallel-for loops to serial for-loops. We present results for machine M1. Results for serial CPU execution on machine M2 were qualitatively similar and not presented here. Even when

parallel-for loops are disabled, some benchmarks may still use multiple cores because they might call multi-threaded library operations. All the array-based language implementations considered here (McVM, MathWorks MATLAB, Python, Velociraptor) call multi-threaded libraries.

For MATLAB, we evaluated the performance against MathWorks MATLAB and against McVM without integrated Velociraptor. For Python, we evaluated the performance against Cython, an optimizing static JIT compiler for Python that generates C++ from a type-annotated Python dialect. For our Python dwarf benchmarks, we also had C code that implemented the same computation. In these cases, we also evaluated the performance serial C code compiled with gcc. Results are presented in *Table 10.1*.

	Benchmark	Velociraptor	C	MATLAB
Python dwarves	lud	1.04	1.04	
	srad	5.09	8.19	
	pagerank	1.23	1.21	
	spmv	1.02	1.62	
	fft	1.33	23.81	
Python NumFocus	growcut	2.90		
	julia	2.46		
	pairwise	1.47		
MATLAB benchmarks	nb1d	1.03		0.66
	fiff	1.09		1.34
	fftm	0.97		0.37
	clos	1.0		1.01
	capr	1.83		0.21
	fdtd	0.87		0.23

Table 10.1 Speedup of CPU code generated over baseline compilers Cython and McVM on machine M1

Results show that the serial code generated by Velociraptor is generally competitive with the baseline compilers Cython and McVM. With the exception of the FFT Python benchmark, the code generated by Velociraptor is not far from C performance. Both Cython and Velociraptor perform poorly on FFT Python benchmark compared to C because the benchmark is written in a recursive way and generates and destroys millions of objects.

10.4 Parallel-loop performance

Velociraptor supports parallel-for loops in VRIR. Some of the benchmarks can utilize this construct. We present the speedup of enabling parallel-for loops compared to serial code generated by Velociraptor in *Table 10.2*. Machine M1 has a quad-core CPU with hyper-threading. Machine M2 has a quad-core CPU but each pair of cores shares the FPU, instruction decoders and the L2 cache. Thus, we expect lower overall speedup on machine M2 compared to M1 and this is reflected in our results. The geometric mean of speedup achieved on M1 was 3.5 while geometric mean of speedup achieved on M2 was 2.13.

	Benchmark	Machine M1	Machine M2
Python dwarves	lud	2.94	2.12
	srad	4.76	1.8
	pagerank	3.84	1.82
	spmv	3.67	1.68
Python NumFocus	growcut	2.8	3.05
	julia	5.23	2.91
	pairwise	4.03	2.75
MATLAB benchmarks	fiff	2.43	1.87
	capr	2.83	1.73
Geometric mean		3.5	2.13

Table 10.2 Speedup of parallel code generated by Velociraptor compared to serial code generated by Velociraptor

10.5 GPU performance

In this section, we evaluated the speedup of hybrid CPU+GPU code generated by Velociraptor with the parallel CPU code generated by Velociraptor. In both cases, all compiler and runtime optimizations are turned on. We performed experiments on all three GPUs and the results in presented in *Table 10.3*.

Overall, the results of using GPUs were mixed. Compared to CPUs, GPUs have several overheads such as data transfer making them unsuitable for problems where a large

	Benchmark GPU	Machine M1 Nvidia C2050	Machine M2 AMD 8650G	 AMD 8750M
Python dwarves	lud	1.17	0.37	0.5
	srad	0.63	1.72	2.55
	pagerank	0.75	0.61	0.98
	spmv	0.12	0.12	0.13
Python NumFocus	growcut	0.34	0.54	1.43
	julia	0.44	0.88	1.34
	pairwise	1.03	0.98	2.81
MATLAB benchmarks	nb1d	1.7	1.03	1.83
	fiff	0.7	0.65	0.34
	clos	2.34	0.84	0.84
	capr	0.69	0.61	0.78
	fdtd	0.87	1.12	2.03
Geometric mean		0.71	0.67	0.97

Table 10.3 Speedup of hybrid CPU + GPU code generated by Velociraptor over parallel CPU code generated by Velociraptor

amount of data transfer and only a small amount of actual computation is required. SpMV benchmark is one such example.

Another issue is that the JIT compilation for GPUs may be bigger than the JIT compilation overhead for CPUs. On the Nvidia-based machine M1, the installed GPU is a server class GPU with high floating-point throughput and thus one may expect to achieve good speedups over CPUs. However, we found that kernel compilation time was quite significant and in many cases was more than the kernel execution time. For example, in the growcut benchmark, the GPU executed the kernel 10 times faster than the CPU execution time but the OpenCL kernel compilation time was many times longer than the GPU execution time.

The machine M2 is a laptop and the GPUs are budget chips designed for low-power operation. For example, the fp64 throughput of these GPUs is cut down to $1/16$ the speed of fp32 operations compared to $1/3$ or $1/2$ speed that workstation or server GPUs are designed for. Further, the discrete GPU 8750M is connected to the system via a very low speed interconnect (PCIe 2.0 x8) whereas server or workstation GPUs typically connect at much higher bandwidth. Given these constraints, big speedups were not expected but the GPUs still performed well in many benchmarks. AMD's OpenCL driver was much faster

at compiling OpenCL kernels and thus kernel compilation time was not the bottleneck.

10.6 Region specialization

We first discuss the behavior of region specialization and then look at the performance impact. Region detection analysis identified interesting regions in 12 out of 14 benchmarks being considered in this section. The only benchmarks where region detection analysis did not identify any regions were the FFT dwarf and the FFT MATLAB benchmark. The FFT dwarf is a written as a recursive function, which are not handled by region detection analysis. The FFT MATLAB benchmark is written as a complex for-loop making it hard to analyze.

One of the concerns with program specialization techniques is that they may generate many different versions at runtime, which increases JIT compilation time and memory consumption. We found that each of the regions identified in the benchmarks was specialized only once in a given execution of the program.

The main objective of region specialization is to gather accurate shape information. We found that in 10 out of 12 benchmarks where regions were identified, we were able to infer the shapes of all variables occuring inside the region. The two cases where we were only able to gather partial shape information were spmv and growcut. The former has some indirect memory accesses that the compiler was not able to analyze while the latter had a non-affine inner loop that made a few array index expressions unanalyzable.

We investigated the performance impact of region specialization on both CPU code as well as CPU+GPU code. We present the speedups of enabling region specialization compared to disabling region specialization in *Table 10.4*. We only consider machine M2 with AMD GPUs in this section because the large OpenCL kernel compilation time for machine M1 overshadows any performance impact of compiler optimizations.

10.7 Using multiple hardware queues

Using multiple hardware queues requires a GPU that supports multiple hardware queues. Out of the three GPUs under consideration in this section, only Radeon 8750M in machine

	Benchmark	CPU	AMD 8650G	AMD 8750M
Python dwarves	lud	1.04	1.2	1.05
	srad	1.29	1.13	1.08
	pagerank	1.05	1.42	1.02
	spmv	1.13	1.0	1.0
Python NumFocus	growcut	1.51	1.84	1.35
	julia	1.0	1.0	1.0
	pairwise	1.28	2.14	1.15
MATLAB benchmarks	nb1d	0.99	1.	1.
	fiff	1.16	1.12	1.07
	clos	1	1	1
	capr	1.05	1.14	1.09
	fdtd	0.99	0.98	1.
Geometric mean		1.11	1.21	1.06

Table 10.4 Speedup obtained by enabling region specialization over not enabling region specialization on machine M1

M2 supports multiple queues. Further, out of the benchmarks we tested, only the nb1d benchmark consists of multiple kernels that can be queued independently. On the nb1d benchmark, we obtained a speedup of 7% by enabling the optimization of automatically using multiple hardware queues.

10.8 Conclusions

We demonstrated that Velociraptor generated serial CPU code is competitve with state-of-the-art compilers and VMs such as Cython, McVM and MATLAB. We also demonstrated that parallel-loops supported in Velociraptor can provide good speedups in many benchmarks on multi-core CPUs. For GPUs, we found good speedups over multicore in several benchmarks using AMD GPUs, particularly AMD GCN architecture based Radeon 8750M. For Nvidia GPUs, we found that OpenCL kernel compilation time was a significant issue and overshadowed any gains in execution time.

We also studied the effect of optimizations implemented in VRcompiler. We showed that region specialization was quite successful in identifying regions in benchmarks, obtaining shape information at runtime and finally led to significant improvement in performance

for both CPUs and GPUs.

Chapter 11
Related Work

Velociraptor is a reusable toolkit that can be used for building compilers for different languages. There exists a large body of work related to compilation infrastructure for multiple languages and we discuss the relation of Velociraptor to prior work in Section 11.1. We discuss prior work on compilers for array-based languages that target GPUs in Section 11.2 and compiler infrastructure for GPUs intended for multiple languages in Section 11.3. Region specialization, which collects shape information at runtime, is one of the contributions of this work and we discuss related work about shape inference in Section 11.4. Finally, we discuss related work on matrix operations libraries for GPUs in Section 11.5.

11.1 Building compilers for multiple languages

The idea that multiple languages can be compiled to the same intermediate representation (IR) is an established principle. For example, GCC compiler suite has front-ends of multiple languages (C, C++, Objective C, D, Fortran) that all compile to the same IR and thus much of the code generation and optimization infrastructure is shared between languages. Similarly, many languages use LLVM IR as a compilation target. For example, Clang project is a frontend that compiles C, C++ and Objective-C to LLVM IR.

Another related idea is to compile languages to instruction set of mature virtual machines such as various Java Virtual Machine (JVM) implementations and Microsoft's Com-

mon Language Runtime (CLR). Substantial investment has been made into optimizing these virtual machines. Thus, these VMs have good JIT compilers and garbage collectors. Multiple languages can be compiled to their instruction sets and all languages benefit from the JIT compiler infrastructure built for the VM. Languages that run on the JVM include [46] Java, Scala and Kotlin and languages that run on CLR [45] include C#, F#, VB.net and Boo.

Velociraptor and VRIR take inspiration from these ideas of a common IR for multiple languages. Our solution applies the idea in a different context and with several key differences.

The first difference is that most of these toolkits (such as GCC's backend infrastructure, JVM or CLR) are monolithic platforms that were not designed to be embedded or integrated with other systems. If a language implementation wants to use these platforms, then that choice generally needs to be made upfront before starting to write the implementation. However, evolving existing systems is also an important design goal for our work. Substantial rewrites of language implementations may break compatibility with existing user code, especially code that may depend upon the C/C++ API. In contrast, Velociraptor is explictly designed as an embedded compiler toolkit and is a better choice to evolve existing language implementations.

The second difference is that the common IR (such as LLVM IR, GCC's IR, JVM or CLR bytecode) are quite low-level from the point of view of array-based languages. For example, these IRs do not provide any rich array data-types or high-level matrix operations. VRIR is a high-level solution that has been designed specifically for array-based numerical computing languages. There has been no previous effort to create a unified IR for array-based languages.

Finally, Velociraptor is also designed specifically for array-based languages. Velociraptor offers optimizations such as shape specialization and memory reuse optimization that are specific to array-based languages. These optimizations are not found in general-purpose toolkits. Velociraptor also offers both CPU (including multi-core CPU) and GPU code generation whereas the mentioned solutions only offer CPU code generation.

11.2 Compilers for array-based languages for GPUs

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 [20] and PyOpenCL [21]. 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 [9] 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 [16] was a dynamic compiler that compiled annotated Python loops to AMD's deprecated CAL API. Theano [7] is a Python library that compiles expression trees to CUDA. In addition to GPU code generation, it also includes features like symbolic differentiation. Parakeet [36] is a compiler that takes as input annotated Python code and generates CUDA for GPUs and LLVM for CPUs. MEGHA[30] 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 [32] 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 [29] is a NumPy-aware JIT compiler for compiling Python to LLVM. The work has some similarities to our work on the CPU side, including support for various array layouts, but it is tied to Python and therefore does not support the indexing schemes not supported by Python. Numba also provides an initial prototype of generating CUDA kernels, given the body of the kernel (equivalent to the body of a parallel loop) in Python. However, it assumes the programmer has some knowledge of CUDA, and exposes some CUDA specific variables (such as thread-block index) to the programmer. Furthermore, unlike our work, it is not a general facility for annotating entire regions of code as GPU-executable and will mostly be useful for converting individual loop-nests to CUDA.

11.3 Multi-language compilers targeting GPUs

One possible approach to building a multi-language compiler for GPUs is to compile bytecode of high-level VMs to GPUs. This is complementary to our approach and there are two recent examples of this approach. AMD's Aparapi [1] provides a compiler that compiles Java bytecode of a method to an OpenCL kernel body, and the generated kernel can then be applied over a domain. Unlike VRIR, which contains multi-dimensional arrays and high-level array operators, Aparapi is a low-level model where only one-dimensional arrays are allowed with Java's simple indexing operators. Dandelion [35] is a LINQ style extension to .net that cross-compiles .net bytecode to CUDA. The programming style for LINQ is quite different from the loop and array-operator based approach in our work.

To our knowledge, there has not been prior work on embeddable or reusable compilers for GPUs. The only work in this area that we are aware of is NOVA [11]. NOVA is a static compiler for a domain-specific compiler for a new functional language and it generates CUDA code. Velociraptor and NOVA provide different programming styles (imperative loop and array-based vs functional respectively) and different compilation models (dynamic vs static respectively). Collins et al. claim that NOVA can be used as an embedded domain-specific language but did not show any such embedded uses whereas we integrated our toolkit in two compilers.

11.4 Shape inference and region specialization

Various research project have attempted shape inference using different techniques. The MAGICA system [19] was a static compiler that inferred shapes statically. MAGICA can reason about symbolic shapes and in some cases could prove that two arrays have the same shape without knowing the actual value of the shape. Our approach is quite different from MAGICA because we perform shape inference and specialization at runtime, and take into account program information obtained at runtime. This allows our system to succeed in cases where problem sizes are determined at runtime, for example through user-input. Our system also reasons about loops and can deal with constructs such as array-growth which MAGICA could not deal with.

MaJIC [3], a dynamic compiler for MATLAB, and Parakeet [36] for Python, also performed some shape inference by propagating shape information at function level . MaJIC maintained array shape information as part of its type system and specialized functions at runtime based upon types of the arguments. The type information, which included array shape information, was propagated inside the function. Some limited constant propagation was also performed.

Our system can be thought of as an extension of the technique in MaJIC but with two major differences. First, we introduced region detection analysis and we specialize regions instead of entire functions. Thus, our system specializes at a more granular level and specialization is done at a program point where we possibly have more accurate information required for shape inference. For example, both MaJIC and MAGICA will be unable to infer shapes in the code shown in *Figure 6.2*. Second, region detection analysis identifies the critical variables whose value or shape should be used in specialization. For example, consider a function with two parameters with multiple parameters. Some of the parameters may simply be numerical values used in the computation and not related to shape information. In this case, one may not need to specialize the function based upon value of the numerical parameter. Region detection analysis identifies the parameters which should be used for generating specialized versions of the functions. MaJIC did not distinguish between useful and non-useful function parameters and performed function specialization based on value of all the parameters.

Grelck et al. [17] investigated runtime specialization of SAC functions including runtime specialization based on shape. They have a two-step compilation process. An ahead-of-time compiler generates a generic version of the function. When the program calls the generic version of the function, it begins executing but in a background thread another compiler is invoked that generates a specialized version of the function. Once the specialized version is ready, the execution is switched to the specialized version. Our compiler specializes code at a more granular level where we identify and specialize regions of code instead of entire functions.

McFLAT [5] introduced a profile-guided static compilation system that observed shape information from several executions of the program and then specialized the program based upon frequently encountered shapes. The present work is a different approach that performs

shape specialization at runtime and is not a profile-guided methodology. The author of this work has been involved in a related work called Jit4gpu. Jit4gpu [16] is a compiler that generated GPU code from loop-nests written in Python, could perform some limited shape-specialization. However, jit4gpu did not attempt any shape information propagation, relying instead on simple runtime lookup of values which only worked in very simple cases. The approach was also limited to single loop-nests and not entire regions of code.

Some other lines of research are related to our work. Shape specialization has been investigated in the context of compilation of expression-trees of vector operations. For example, some C++ libraries, such as Eigen, use template metaprogramming to achieve static compile-time shape specialization of vector operations. Dynamic specialization of code has been investigated in the context of inspector-executor methods but these are typically limited to a single loop-nest instead of regions of code, and often require at least partial execution of code. Finally, our concept of regions of code is similar but not equivalent to static control portions (SCoP) [6] in the polyhedral literature.

11.5 Matrix libraries for GPUs

Auto-tuning is a well-established technique on CPUs. ATLAS [4] is a well-known example of an auto-tuning BLAS. Auto-tuning has also been used for some FFT libraries such as FFTW [15]. However, FFTW uses an online tuner where the tuning happens at application runtime. In contrast, libraries like ATLAS perform offline tuning, where auto-tuning is performed at install time and hence only done once per machine. Our approach is similar to ATLAS in this regard.

Implementing a fast GEMM routine on GPUs and other accelerators has been of considerable interest in the past few years. Several researchers have written hand-tuned implementations for particular GPU architectures Volkov et al. [43] described a fast GEMM prototype in CUDA for Nvidia's G80 architecture. Their ideas have now been included in Nvidia's CUBLAS library. Nakasato [25] described a fast GEMM prototype for AMD's Cypress GPU (which powered products such as Radeon 5870). Their implementation was written in AMD's CAL API, which was a low-level pseudo-assembler exposed by AMD for their previous-gen GPUs. CAL API has now been deprecated. Nath et al. [26] report

a fast CUDA GEMM implementation for Nvidia Fermi GPUs. Tan et al. [41] describe a fast CUDA GEMM implementation for Nvidia's Fermi architecture. They wrote their implementation in PTX, which is a pseudo-assembler for CUDA. This allowed tighter control over instruction scheduling compared to high-level languages like CUDA-C and OpenCL. They report better performance than CUBLAS. Matsumoto et al. [24] reported several high-performance OpenCL GEMM kernels for AMD's Tahiti GPU. Their implementation uses an auto-tuner to search for parameters, though they limit their experimental evaluation to only one architecture so it is not clear how well it will translate to other architectures. Schneider et al. [37] implemented a fast ZGEMM routine on Cell Broadband Engine that ran on Cellś Synergistic Processing Unit (SPU). They optimized their implementation for the vector instruction set of the Cell processor.

Several researchers have looked at portable OpenCL GEMM implementations for multiple architectures. Du et al. [13] present a study of a portable GEMM implementation. They had two codepaths. One codepath was an OpenCL translation of Fermi GEMM routines from MAGMA's CUDA kernels. This was essentially a handwritten implementation with only a few parameters. The second codepath was an auto-tuning implementation for AMD GPUs. In comparison, our library offers a unified and completely auto-tuning implementation for all architectures. In terms of performance, our results show about 15% higher performance on SGEMM on Nvidia GPUs and we get comparable performance on AMD architectures. Weber et al. [44] discussed an auto-tuning implementation on AMD's GCN GPUs. However, compared to Weber et al., our results show about 60% higher peak performance on Radeon 7970 on SGEMM and about 20% higher performance on DGEMM. Tillet et al. [42] also presented an auto-tuning framework and showed results for SGEMM and DGEMM on Fermi and GCN architectures. However, our results are almost 40% faster on GCN and 20% faster on Fermi. The reason for our much faster performance is that Tillet et al. are only using one codelet, NN layout with consecutive elements, while our tuner tests 6 different codelets and we show that the TN codelet with offset elements is optimal for these GPUs. Further, their codelet accepted fewer parameters. For example, their codelet does not support copying inputs into OpenCL images.

There is no previous research on tuning GEMM for single-chip CPU+GPU systems. There is also no previous coverage of the Intel Ivy Bridge platform.

11.6 Conclusion

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

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

Chapter 12
Conclusions and Future Work

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

We presented an embedded toolkit called Velociraptor that can be used by compiler writers inside their own compiler to compile numerical array-based code to both CPU as well as GPU code. The toolkit is designed to be embedded inside existing language implementations. In order to simplify the evolution of existing codebases, our toolkit does not require rewriting of data-structures such as array representations inside the language runtime.

Velociraptor toolkit is designed to be reusable across languages. We discussed a new high-level IR called VRIR to achieve this language portability. VRIR can be used to express a wide variety of core array-based computations and can contain code sections for both CPUs and GPUs. 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.

We presented a novel program specialization technique called region specialization that is particularly useful for array-based languages. The technique first identifies interesting regions of code, and then specializes these regions based on information collected at run-

time. We showed that it can be used to collect accurate shape information at runtime. We demonstrated that region specialization and the optimizations it enables, such as bounds-check elimination, can be profitable on both CPUs and GPUs.

In addition to the compilation infrastructure, Velociraptor also includes a smart GPU runtime that manages GPU resources and task dispatch to the GPU. VRruntime efficiently implements some standard techniques such as avoiding redundant data transfers and asynchronous dispatch. VRruntime also includes a new optimization of automatically dispatching work to more than one GPU queue to keep the GPU fully utilized and we showed that it can lead to some performance improvements in benchmarks where it is applicable.

The compilation infrastructure provided by Velociraptor is complemented by a matrix operations library called RaijinCL and it is a new contribution of this thesis. RaijinCL is a portable and high-performance matrix operations library that generates optimized OpenCL kernel for any given GPU architecture using auto-tuning technique. We demonstrated that RaijinCL is competitive with vendor-tuned libraries from multiple vendors. We also demonstrated a hybrid matrix-multiply solution that utilizes both the CPU and GPU. Our experimental results show that the GPU kernel best suited for GPU-only workloads is not necessarily the best kernel for hybrid workloads.

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

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

community about the need and potential designs of such toolkits.

Bibliography

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

[2] Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, 2006.

[3] George Almási and David Padua. MaJIC: compiling MATLAB for speed and responsiveness. In *PLDI '02*, pages 294–303, 2002.

[4] Clint Whaley Antoine, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27:3–35, 2000.

[5] Amina Aslam and Laurie Hendren. McFLAT : A profile-based framework for MATLAB loop analysis and transformations. In *Proceedings of the 23rd International Workshop on Languages and Compilers for Parallel Computing (LCPC 2010)*, pages 1–15. Springer Berlin / Heidelberg, October 2010.

[6] Cedric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 7–16. IEEE Computer Society, 2004.

[7] 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.

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

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

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

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

[12] Jesse Doherty. McSAF: An extrensible static analysis framework for the MATLAB language. Master's thesis, August 2011.

[13] 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.

[14] John Eaton. GNU Octave. http://www.gnu.org/software/octave.

[15] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

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

[17] Clemens Grelck and Heinz Wiesinger. Towards persistent and parallel asynchronous adaptive specialisation for data-parallel array processing in SAC. In *25th symposium on Implementation and Application of Functional Languages*, August 2013.

[18] The Khronos Group. The OpenCL Specification.

[19] Pramod G. Joisha and Prithviraj Banerjee. An algebraic array shape inference system for MATLAB. *ACM Trans. Program. Lang. Syst.*, 28(5):848–907, September 2006.

[20] Andreas Klöckner. PyCUDA. http://mathema.tician.de/software/pycuda.

[21] Andreas Klöckner. PyOpenCL web page. http://mathema.tician.de/software/pyopencl.

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

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

[24] Kazuya Matsumoto, Naohito Nakasato, and Stanislav G. Sedukin. Implementing a code generator for fast matrix multiplication in OpenCL on the GPU. Technical Report 2012-002, Graduate School of Computer Science and Enginering, The University of Aizu, July 2012.

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

[26] 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.

[27] netlib.org. BLAS (Basic Linear Algebra Subprograms).

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

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

[30] 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.

[31] OpenBLAS project. OpenBLAS homepage.

[32] 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.

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

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

[35] 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.

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

[37] T. Schneider, T. Hoefler, S. Wunderlich, T. Mehlan, and W. Rehm. An optimized ZGEMM implementation for the Cell BE. In *Proceedings of the 9th Workshop on Parallel Systems and Algorithms (PASA)*, pages 113–122, Dresden, Germany, February 2008.

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

[39] 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.

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

[41] Guangming Tan, Linchuan Li, Sean Triechle, Everett Phillips, Yungang Bao, and Ninghui Sun. Fast implementation of DGEMM on Fermi GPU. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 35:1–35:11, New York, NY, USA, 2011. ACM.

[42] Philippe Tillet, Karl Rupp, Siegfried Selberherr, and Chin-Teng Lin. Towards performance-portable, scalable, and convenient linear algebra. In *5th USENIX Workshop on Hot Topics in Parallelism*, 2013.

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

[44] 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.

[45] Wikipedia. List of CLI languages — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=List_of_CLI_languages&oldid=636919108, 2014. [Online; accessed 15-December-2014].

[46] Wikipedia. List of JVM languages — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=List_of_JVM_languages&oldid=637537166, 2014. [Online; accessed 15-December-2014].

Appendix A
VRIR textual syntax grammar

```
grammar Vrir;


module: '(' ModuleTok AttrName Name AttrIndexing Intval '(' FnsTok
    function* ')' ')' ;

function: '(' FunctionTok AttrName? Name vtype arglist '(' BodyTok
    statement* ')' symtable ')';

arglist: '(' ArglistTok fnargs* ')' ;

fnargs: '(' 'arg' AttrId Intval ')'
 | Intval
 ;

symtable: '(' SymtableTok sym* ')' ;

sym: '(' SymTok AttrId Intval AttrName Name vtype ')' ;

vtype: vbasetype
 | arraytype
 | tupletype
 | fntype
 | domaintype
```

```
 | voidtype
 ;

voidtype: VoidTok
 | '(' VoidTok ')'
 | '(' VtypeTok AttrName VoidTok ')'
 ;

vbasetype: '(' basetype (AttrCtype Intval)? ')' ;

basetype: Int32Tok
 | Int64Tok
 | Float32Tok
 | Float64Tok
 | BoolTok
 ;

numtype: vbasetype
 | arraytype
 ;

alayout: RowmajorTok
 | ColmajorTok
 | StridedTok
 ;

arraytype: '(' ArrayTypeTok AttrLayout alayout AttrNdims Intval vtype
    ')' ;

tupletype: '(' TupleTypeTok numtype+ ')' ;

domaintype: '(' DomainTypeTok (AttrNdims Intval)? vbasetype+ ')' ;

infntype: numtype
 | fntype
 ;

```

```
59 outfntype: numtype
60  ;
61
62 outfndecl: '(' OuttypesTok outfntype+ ')'
63  | '(' OuttypesTok voidtype ')'
64  ;
65
66 fntype: '(' FnTypeTok '(' IntypesTok infntype* ')' outfndecl ')' ;
67
68 statement: assignstmt
69  | stmtlist
70  | forstmt
71  | pforstmt
72  | whilestmt
73  | ifstmt
74  | breakstmt
75  | continuestmt
76  | returnstmt
77  | exprstmt
78  ;
79
80 stmtlist: '(' StmtlistTok (AttrOnGpu boolval)? '(' StmtsTok statement*
       ')' ')';
81
82 assignstmt: '(' AssignstmtTok '(' LhsTok expr+ ')' '(' RhsTok expr ')'
       ')' ;
83
84 itervars: '(' ItervarsTok itersym+ ')' ;
85 itersym: Intval
86  | '(' SymTok AttrId Intval ')'
87  ;
88
89 loopdomain: '(' LoopDomainTok expr ')'
90  | '(' LoopDomainTok range ')'
91  ;
92
93 range_stop: '(' StopTok expr ')' ;
```

```
range_start: '(' StartTok expr ')' ;
range_step: '(' StepTok expr ')' ;

range: '(' RangeTok AttrExclude boolval range_start? range_step?
    range_stop? ')' ;


forstmt : '(' ForstmtTok itervars loopdomain '(' BodyTok statement* ')'
    ')' ;

sharedvars: '(' SharedTok Intval* ')'
 | '(' SharedTok AutoTok ')'
 ;

pforstmt: '(' PforstmtTok itervars loopdomain sharedvars '(' BodyTok
    statement* ')' ')' ;
whilestmt: '(' WhilestmtTok '(' TestTok expr ')' '(' BodyTok statement*
    ')' ')';
elsestmt: '(' ElseBodyTok statement* ')' ;
ifstmt: '(' IfstmtTok '(' TestTok expr ')' '(' IfBodyTok statement* ')'
    elsestmt? ')' ;
breakstmt: '(' BreakTok ')' ;
continuestmt: '(' ContinueTok ')' ;
returnstmt: '(' ReturnstmtTok ( '(' ExprsTok expr+ ')' )? ')' ;
exprstmt: '(' ExprstmtTok expr ')' ;

expr: plus
 | minus
 | mult
 | divide
 | lt
 | leq
 | gt
 | geq
 | eq
 | neq
 | indexexpr
```

```
126  | andexpr
127  | orexpr
128  | notexpr
129  | name
130  | fncall
131  | domain
132  | realconst
133  | libcall
134  | alloc
135  | dim
136  | tuple
137  | tupleindex
138  | negate
139  | cast
140  | complexexpr
141  | realexpr
142  | imagexpr
143  | dimvec
144  | fnhandle
145  ;
146
147
148 complexexpr: '(' ComplexTok vtype expr expr ')' ;
149 realexpr: '(' RealTok AttrId Intval vtype ')' ;
150 imagexpr: '(' ImagTok AttrId Intval vtype ')' ;
151 dimvec: '(' DimvecTok AttrId Intval vtype ')' ;
152
153
154 plus: '(' PlusTok vtype '(' LhsTok expr ')' '(' RhsTok expr ')' ')' ;
155 minus: '(' MinusTok vtype '(' LhsTok expr ')' '(' RhsTok expr ')' ')' ;
156 mult: '(' MultTok vtype '(' LhsTok expr ')' '(' RhsTok expr ')' ')'
157 divide: '(' DivTok vtype '(' LhsTok expr ')' '(' RhsTok expr ')' ')' ;
158 lt: '(' LtTok vtype '(' LhsTok expr ')' '(' RhsTok expr ')' ')' ;
159 leq: '(' LeqTok vtype '(' LhsTok expr ')' '(' RhsTok expr ')' ')' ;
160 gt: '(' GtTok vtype '(' LhsTok expr ')' '(' RhsTok expr ')' ')' ;
161 geq: '(' GeqTok vtype '(' LhsTok expr ')' '(' RhsTok expr ')' ')' ;
162 neq: '(' NeqTok vtype '(' LhsTok expr ')' '(' RhsTok expr ')' ')' ;
```

```
163 eq: '(' EqTok vtype '(' LhsTok expr ')' '(' RhsTok expr ')' ')' ;
164 andexpr: '(' AndTok vtype '(' LhsTok expr ')' '(' RhsTok expr ')' ')';
165 orexpr: '(' OrTok vtype '(' LhsTok expr ')' '(' RhsTok expr ')' ')' ;
166 notexpr: '(' NotTok vtype expr ')' ;
167 name: '(' NameTok AttrId Intval vtype ')' ;
168 fncall: '(' FncallTok AttrFnname Name vtype '(' ArgsTok expr* ')' ')' ;
169 fnhandle: '(' FnhandleTok AttrFnname Name vtype ')' ;
170
171
172 baseindex: expr | range;
173 aindex: '(' IndexTok AttrBoundscheck boolval AttrNegative boolval
        (AttrGrow boolval)? baseindex ')'
174  | '(' IndexTok AttrMaxBounds boolval AttrMinBounds boolval
         AttrNegative boolval (AttrGrow boolval)? baseindex ')' ;
175
176 indexexpr: '(' IndexTok AttrArrayid Intval (AttrCopyslice boolval)?
       vtype '(' IndicesTok aindex+ ')' ')' ;
177 domain: '(' DomainTok domaintype range+ ')' ;
178
179 floatconst: Floatval | Intval;
180 realconst: '(' RealConstTok AttrIval Intval vtype ')'
181  | '(' RealConstTok AttrDval floatconst vtype ')'
182  ;
183
184
185 libcall: '(' LibcallTok AttrLibfunc Name vtype '(' ArgsTok expr+ ')'
       ')' ;
186 alloc: '(' AllocTok AttrFunc AllocFunc vtype '(' ArgsTok expr+ ')' ')' ;
187 dim: '(' DimTok AttrArrayid Intval AttrDimid Intval vtype')' ;
188 tuple: '(' TupleTok vtype '(' ElemsTok expr+ ')' ')' ;
189 tupleindex: '(' TupleIndexTok AttrIndex Intval AttrTupleid Intval vtype
       ')' ;
190 negate: '(' NegateTok vtype expr ')' ;
191 cast: '(' CastTok vtype expr ')' ;
192 boolval: TrueTok | FalseTok;
193
194
```

```
195 ModuleTok: 'module';
196 FunctionTok: 'function';
197 AssignstmtTok: 'assignstmt';
198 ForstmtTok: 'forstmt';
199 PforstmtTok: 'pforstmt';
200 StmtlistTok: 'stmtlist';
201 IfstmtTok: 'ifstmt';
202 WhilestmtTok: 'whilestmt';
203 ReturnstmtTok: 'returnstmt';
204 ContinueTok: 'continue';
205 BreakTok: 'break';
206 ExprstmtTok: 'exprstmt';
207
208 Lp: '(';
209 Rp: ')';
210 VtypeTok: 'vtype';
211 Int32Tok: 'int32';
212 Int64Tok: 'int64';
213 Float32Tok: 'float32';
214 Float64Tok: 'float64';
215 VoidTok: 'void';
216 BoolTok: 'bool';
217
218
219 FnsTok: 'fns';
220 BodyTok: 'body';
221 StmtsTok: 'stmts';
222 ArrayTypeTok: 'arraytype';
223 TupleTypeTok: 'tupletype';
224 FnTypeTok: 'functype';
225 IntypesTok: 'intypes';
226 OuttypesTok: 'outtypes';
227
228 AttrOnGpu: ':onGpu';
229 AttrNdims: ':ndims';
230 AttrId: ':id';
231 AttrCtype: ':ctype';
```

```
232 AttrFnname: ':fnname';
233 AttrLayout: ':layout';
234 AttrIval: ':ival';
235 AttrDval: ':dval';
236 AttrLibfunc: ':libfunc';
237 AttrDimid: ':dimid';
238 AttrTupleid: ':tupleid';
239 AttrFunc: ':func';
240 AttrArrayid: ':arrayid';
241 AttrIndex: ':index';
242 AttrName: ':name';
243 AttrBoundscheck: ':boundscheck';
244 AttrNegative: ':negative';
245 AttrGrow: ':grow';
246 AttrCopyslice: ':copyslice';
247 AttrOp: ':op';
248
249 ExprTok: 'expr';
250 RhsTok: 'rhs';
251 LhsTok: 'lhs';
252 PlusTok: 'plus';
253 MinusTok: 'minus';
254 MultTok: 'mult';
255 DivTok: 'div';
256 LtTok: 'lt';
257 LeqTok: 'leq';
258 GtTok: 'gt';
259 GeqTok: 'geq';
260 EqTok: 'eq';
261 NeqTok: 'neq';
262 AndTok: 'and';
263 OrTok: 'or';
264 IndexTok: 'index';
265 IndicesTok: 'indices';
266 FncallTok: 'fncall';
267 DomainTok: 'domain';
268 RealConstTok: 'realconst';
```

```
TupleTok: 'tuple';
TupleIndexTok: 'tupleindex';
AllocTok: 'alloc';
DimTok: 'dim';
NegateTok: 'negate';
NameTok: 'name';
LibcallTok: 'libcall';
ArgsTok: 'args';
DomainTypeTok: 'domaintype';
SymTok: 'sym';
ItervarsTok: 'itervars';
LoopDomainTok: 'loopdomain';
RangeTok: 'range';
StartTok: 'start';
StepTok: 'step';
StopTok: 'stop';
SharedTok: 'shared';
AutoTok: 'auto';
ExprsTok: 'exprs';
AllocFunc: 'zeros' | 'ones' | 'empty' ;
IfBodyTok: 'if';
ElseBodyTok: 'else';
ElemsTok: 'elems';
CastTok: 'cast';
SymtableTok: 'symtable';
ArglistTok: 'arglist';
TestTok: 'test';
RowmajorTok: 'rowmajor';
ColmajorTok: 'colmajor';
StridedTok: 'strided';
XchangeTok: 'xchange';
ImagTok: 'imag';
ComplexTok: 'complex';
RealTok: 'real';
DimvecTok: 'dimvec';
NotTok: 'not';
FnhandleTok: 'fnhandle';
```

```
306 AttrMinBounds: ':minbounds';
307 AttrMaxBounds: ':maxbounds';
308 AttrIndexing: ':indexing';
309 AttrExclude: ':exclude';
310
311
312 Name: ('a'..'z' | 'A'..'Z') ('a'..'z' | 'A'..'Z' | '0'..'9' | '_')*;
313 FalseTok: '%0';
314 TrueTok: '%1';
315 Intval: ('+'|'-')? ('0' .. '9')+;
316 FloatvalDot: '.' ('0' .. '9')+;
317 Floatval: Intval FloatvalDot
318  | Intval FloatvalDot? ('e' | 'E') Intval;
319 WS: (' ' | '\t' | '\n')+ {$channel = HIDDEN;};
```

Appendix B
Rules for region detection analysis

We summarize the rules for region detection analysis in a tabular format for easy reference. These rules were described in more detail in *Chapter 6*.

Statement properties	Rule
RHS is array library operation. Example $A = B + C$. Here $name(LHS) = A$ and $operands(RHS) = B, C$	$$cvalueVars[i,L] = cvalueVars[i+1,L]$$ $$cshapeVars[i,L] = cshapeVars[i+1,L] - name(LHS) + operands(RHS)$$ $$idx[i,L] = \begin{cases} i & if\ idx[i+1,L] == -1 \\ idx[i,L & if\ idx[i+1,L] \neq -1 \end{cases}$$
RHS is array allocation Example: $A = zeros(m,n)$. Here $name(RHS) = A$, $operands(RHS) = m,n$	$$cvalueVars[i,L] = cvalueVars[i+1,L] + operands(RHS)$$ $$cshapeVars[i,L] = cshapeVars[i+1,L] - name(LHS)$$ $$idx[i,L] = \begin{cases} i & if\ idx[i+1,L] == -1 \\ idx[i,L] & if\ idx[i+1,L \neq -1 \end{cases}$$

RHS is array element read from array *arr* inside affine loop, LHS is not critical variable. Let the set of variables used in the subscript expression on the RHS, excluding the loop-indices, be *subscriptUses*. Example: $t = A[i+m]$ where i is a loop-index. Here $subscriptUses = m$ and $arr = A$.	$cvalueVars[i,L] = cvalueVars[i+1,L] + subscriptUses$ $cshapeVars[i,L] = cshapeVars[i+1,L] + arr$ $idx[i,L] = \begin{cases} i & if\ idx[i+1,L] == -1 \\ idx[i,L] & if\ idx[i+1,L] \neq -1 \end{cases}$
RHS is array element read in non-affine loop, LHS is not critical variable	$cshapeVars[i,L] = cshapeVars[i+1,L]$ $cvalueVars[i,L] = cvalueVars[i+1,L]$ $idx[i,L] = \begin{cases} i & if\ idx[i+1,L] == -1 \\ idx[i,L] & if\ idx[i+1,L] \neq -1 \end{cases}$

RHS is an element read from a loop-invariant array *arr* inside an affine loop, and LHS is a critical value variable. Let the set of variables used in the subscript expression on the RHS, excluding the loop-indices, be $subscriptUses$. Example: $t0 = A[i+t1]$. Here $arr = A$ and $subscriptUses = t1$.	$$cshapeVars[i,L] = cshapeVars[i+1,L] + arr + subscriptUses$$ $$cvalueVars[i,L] = cvalueVars[i+1,L] + arr$$ $$idx[i,L] = \begin{cases} i & if\ idx[i+1,L] == -1 \\ idx[i,L] & if\ idx[i+1,L] \neq -1 \end{cases}$$
RHS is an array slice of array *arr*. Let the set of variables used in the slice range expression be $subscriptUses$. Example: $B = A(m:n)$. Here $subscriptUses = m,n$, $arr = A$ and $name(LHS) = B$.	$$cvalueVars[i,L] = cvalueVars[i+1,L] + \{m,n\}$$ $$cshapeVars[i,L] = cshapeVars[i+1,L] - name(LHS) + arr$$ $$idx[i,L] = \begin{cases} i & if\ idx[i+1,L] == -1 \\ idx[i,L] & if\ idx[i+1,L] \neq -1 \end{cases}$$

RHS is an array *arr* subscripted with an array. Let the array used as subscript be *subArr*. Example $C = A(B)$. Here $arr = A$, $subArr = B$ and $name(LHS) = C$	$cshapeVars[i,L] = cshapeVars[i+1,L] + subArr + arr - name(LHS)$ $cvalueVars[i,L] = cvalueVars[i+1,L] + subArr$ $idx[i,L] = \begin{cases} i & if\ idx[i+1,L] == -1 \\ idx[i,L] & if\ idx[i+1,L] \neq -1 \end{cases}$
LHS is an array element write inside an affine loop. Let the array being written to be *arr* and the set of variables used in the subscript expression, excluding the loop index, be $subscriptUses$. Example: $A[i+t0] = t1$. Here $subscriptUses = t0$ and $arr = A$.	$cvalueVars[i,L] = cvalueVars[i+1,L] + subscriptUses$ $cshapeVars[i,L] = cshapeVars[i+1,L] + arr$ $idx[i,L] = \begin{cases} i & if\ idx[i+1,L] == -1 \\ idx[i,L] & if\ idx[i+1,L] \neq -1 \end{cases}$

LHS is an array *arr* subscripted with an array *subArr* and there is no array growth. Example $A(B) = C$. Here $arr = A$, $subArr = B$ and $name(RHS) = C$.	$$cvalueVars[i,L] = cvalueVars[i+1,L] + subArr$$ $$cshapeVars[i,L] = cshapeVars[i+1,L] + subArr$$ $$+arr + name(RHS)$$ $$idx[i,L] = \begin{cases} i & if\ idx[i+1,l] == -1 \\ idx[i,l] & if\ idx[i+1,l] \neq -1 \end{cases}$$
LHS is a scalar variable, RHS is a scalar arithmetic expression and LHS is not a critical variable. Example $t0 = t1 + t2$.	$$cvalueVars[i,l] = cvalueVars[i+1,L]$$ $$cshapeVars[i,l] = cshapeVars[i+1,L]$$ $$idx[i,L] = \begin{cases} i & if\ idx[i+1,L] == -1 \\ idx[i,L] & if\ idx[i+1,L] \neq -1 \end{cases}$$
LHS is a scalar variable, and RHS is a scalar expression involving addition or subtraction of scalar variables. Let the operands on RHS be $operands(RHS)$. Example: $t0 = t1 + t2$. Here $name(LHS) = t0$ and $operands(RHS) = t1, t2$.	$$cvalueVars[i,L] = cvalueVars[i+1,L] - name(LHS)$$ $$+operands(RHS)$$ $$cshapeVars[i,L] = cshapeVars[i+1,L]$$ $$idx[i,L] = \begin{cases} i & if\ idx[i+1,L] == -1 \\ idx[i,L] & if\ idx[i+1,L] \neq -1 \end{cases}$$

Table B.1 Data-flow rules for region detection analysis operating on assignment statements