Abstracties voor het programmeren van grafische processoren
in hoogniveau-programmeertalen

Abstractions for Programming Graphics Processors
in High-Level Programming Languages

Tim Besard

Promotor: prof. dr. ir. B. De Sutter
Proefschrift ingediend tot het behalen van de graad van
Doctor in de ingenieurswetenschappen: computerwetenschappen

UNIVERSITEIT
GENT

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. K. De Bosschere
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2018 - 2019

ISBN 978-94-6355-244-8
NUR 980
Wettelijk depot: D/2019/10.500/52

Examination Committee

Prof. Filip De Turck, *chair*
Department of Information Technology
Faculty of Engineering and Architecture
Ghent University

Prof. Koen De Bosschere, *secretary*
Department of Electronics and Information Systems
Faculty of Engineering and Architecture
Ghent University

Prof. Bjorn De Sutter, *supervisor*
Department of Electronics and Information Systems
Faculty of Engineering and Architecture
Ghent University

Prof. Jutho Haegeman
Department of Physics and Astronomy
Faculty of Sciences
Ghent University

Prof. Jan Lemeire
Department of Electronics and Informatics
Faculty of Engineering
Vrije Universiteit Brussel

Prof. Christophe Dubach
School of Informatics
College of Science & Engineering
The University of Edinburgh

Prof. Alan Edelman
Computer Science & Artificial Intelligence Laboratory
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology

Dankwoord

Ik wist eigenlijk niet waar ik aan begon, toen ik in 2012 in de catacomben van het Technicum op gesprek ging over een doctoraat. Of ik al eens met LLVM gewerkt had. Ondertussen zijn we vele jaren verder, werk ik op een bureau waar er wel daglicht is, en is het eindpunt van deze studie zowaar in zicht. Dat mag natuurlijk wel, zo vertelt men mij, na 7 jaar. Die periode was doorspekt met de nodige doodlopende onderzoeksrichtingen en dubieuze projecten, maar werd gelukkig afgerond met het successvolle onderzoek dat ik nu in dit boek kan presenteren.

Vooreerst wil ik mijn promotor, Bjorn De Sutter, bedanken om dit onderzoek mogelijk te maken. Zijn inzichten bleken essentieel bij het verwerven van de nodige IWT en FWO beurzen. Ook bij het schrijven van wetenschappelijke artikels was hij steeds van de partij om te waken over de kwaliteit, en densiteit, van het afgeleverde werk.

Ook de andere leden van de examencommissie wil ik van harte bedanken, om de tijd te nemen om dit proefschrift te lezen en er kritische vragen over te formuleren. *I would like to thank prof. Christophe Dubach for his critical but constructive view on my work, and for taking the time to travel to Ghent and be part of my examination committee.*

A very special word of thanks goes out to prof. Alan Edelman and Viral Shah, for inviting me to MIT and hosting me for 5 months. The time I spent there reinvigorated my research, and has made it possible for me to graduate today. Valentin, Jarrett, Andreas, Jiahao: you guys are great, and I will fondly remember my time at the Julia Lab.

Ik wil natuurlijk ook alle collega's in Gent bedanken, en in het bijzonder mijn bureaugenoten van de afgelopen jaren: Bart, Bert, Christophe, Jens, Jonas, Panagiotis, Pieter, Ronald, Sander, en Stijn; jullie waren er altijd om de onderzoekseilanden te overbruggen en een aangename werksfeer te creëeren. Bart, als schaduwpromotor was jij er voor alle praktische zaken, of gewoon voor wat weetjes over de universiteit.

Ten slotte wil ik mijn familie en vrienden bedanken voor het leven buiten dit onderzoek. Jullie waren er altijd voor de nodige *reality check*, in een wereld waar IT niet altijd zo belangrijk is. Liesa, zonder jou zou dit alles veel minder zin hebben. Bedankt om mee te gaan in dit verhaal en het avontuur dat ons nog te wachten staat.

Tim Besard
Gent, 1 juni 2019

Samenvatting

Softwareontwikkeling is lange tijd gestoeld geweest op hardware die exponentieel krachtiger werd, waardoor ook de complexiteit van applicaties enorm is kunnen toenemen. Deze periode is echter voorbij: traditionele CPUs (Central Processing Units) verdubbelen niet langer elke paar jaar in snelheid, waardoor rekenintensieve applicaties steeds frequenter gebruik moeten maken van hardwareversnellers zoals GPUs (Graphics Processing Units).

Tezelfdertijd blijft ook de vraag naar steeds krachtiger en complexere applicaties bestaan. Aan deze behoefte is moeilijk te voldoen met traditionele programmeertalen, zoals C en C++, die veel expertise eisen van de programmeur. Bijgevolg winnen zogenaamde hoogniveau-talen, zoals Python en MATLAB, snel aan populariteit. Deze programmeertalen werken op een hoog abstractieniveau waar de programmeur kan focussen op applicatielogica zonder aandacht te spenderen aan hoe die applicatie uiteindelijk zal uitgevoerd worden.

Het is echter moeilijk om hoogniveau-talen te gebruiken om hardwareversnellers zoals GPUs te programmeren. Vooreerst berust het ontwerp van deze talen vaak op elementen die slecht combineren met externe processoren, zoals interpretatie of dynamische compilatie. Tegelijk worden hardwareversnellers hoofdzakelijk gebruikt om de prestatie te verhogen, terwijl hoogniveau-programmeertalen focussen op productiviteit zelfs als dat ten koste gaat van prestatie. Daarom worden hardwareversnellers voornamelijk geprogrammeerd met laagniveau-talen die een goede prestatie garanderen, in combinatie met een hoogniveau-taal om de overige applicatielogica te implementeren.

Deze splitsing van een applicatie in twee of meer talen, waarbij de ene dient om prestatiegevoelige code te implementeren en de andere om op productieve wijze de applicatielogica te definiëren, levert veel problemen op. Het gebruik van meerdere talen werpt in essentie een barrière op die zowel hergebruik, abstractie, en optimalisatie van code verhindert, maar ook samenwerking tussen programmeurs bemoeilijkt.

In dit proefschrift reiken we een alternatief aan, waarbij we **één enkele hoogniveau-programmeertaal gebruiken voor zowel applicatielogica als GPU-code**. Hierbij behouden we de productiviteit van de hoogniveau-taal zonder daarbij in te boeten op de prestatie van code wanneer die uitgevoerd wordt op de hardwareversneller. We bouwen hiervoor verder op de Julia programmeertaal, een hoogniveau-taal die specifiek ontworpen geweest is voor efficiënte uitvoering op CPUs.

Om code geschreven in Julia uit te voeren op een processor zoals een GPU is een nieuwe back end nodig voor de compiler, die broncode omzet naar machinecode voor uitvoering. Traditioneel worden deze alternatieve back ends verweven met de bestaande compiler, of wordt er zelfs een geheel aparte implementatie van de programmeertaal voorzien die specifiek is voor een bepaalde processor.

Als onderdeel van dit onderzoek definiëren we **interfaces tot de hoogniveau-compiler voor het implementeren van dynamische back ends**. Hiermee wordt het mogelijk een nieuwe back end te ontwikkelen zonder die in de bestaande compiler te moeten integreren en daarvoor te moeten voldoen aan eisen op vlak van kwaliteit of licentie. Toch wordt de bestaande compiler hergebruikt waardoor nieuwe back ends snel en efficiënt geïmplementeerd kunnen worden. Tegelijk wordt een parallelle implementatie vermeden, wat de compatibiliteit van broncode met verschillende back ends ten goede komt.
We hebben deze interfaces toegevoegd aan de Julia programmeertaal, en de implementatie ervan bijgedragen aan het bijhorende open-source project. De interfaces bieden toegang tot de verschillende IRs (Intermediate Representations) zoals ze bestaan in de Julia compiler, waaronder IR code van de LLVM (Low Level Virtual Machine) bibliotheek die de compiler gebruikt. Om efficiënt met deze IR te werken hebben we het pakket *LLVM.jl* ontwikkeld om toegang te krijgen tot de LLVM API (Application Programming Interface) vanuit Julia.

We demonstreren vervolgens het potentieel van deze interfaces aan de hand van **een GPU back end voor Julia**. Deze back end, beschikbaar als open-source software onder de naam *CUDAnative.jl*, maakt het mogelijk om Julia code uit te voeren op CUDA (Compute Unified Device Architecture) GPUs. De prestatie van deze hoogniveau-code is vergelijkbaar met equivalente laagniveau-code geschreven in CUDA C, wat we aantonen aan de hand van de Rodinia benchmark suite voor heterogene applicaties. Het programmeren van hoogniveau-code in Julia is echter veel productiever, minder foutgevoelig, en vereist geen bijkomende expertise op vlak van andere, laagniveau-programmeertalen.

Julia biedt als hoogniveau-programmeertaal tevens taalconstructies die het mogelijk maken om krachtige abstracties te definiëren. Automatische specialisatie van generieke code kan bijvoorbeeld gebruikt worden voor hogere-orde reeksabstracties. Hiermee kunnen abstracte operaties op reeksen gescheiden worden van hun concrete implementatie die bepaalt in welk geheugen de elementen zich bevinden, hoe de operatie uitgevoerd wordt, etc. Dit resulteert in compacte en leesbare code, waarbij de hoogniveau-programmeur geen kennis moet hebben van hoe de onderliggende datastructuur geïmplementeerd is.

Aan de hand van de CUDAnative.jl GPU back end hebben we onderzoek gevoerd naar dergelijke **reeksabstracties voor GPUs** zonder de typische barrière tussen hoogniveau-applicatiecode en laagniveau-infrastructuur. We hebben dit geïmplementeerd in het open-source *CuArrays.jl* pakket, en hebben ermee aangetoond dat deze abstracties krachtig genoeg zijn om realistische applicaties te implementeren. De applicaties zijn tevens platform-agnostisch aangezien ze gebruik maken van hogere-orde reeksabstracties. We hebben aangetoond dat hiermee applicaties eenvoudig uitgevoerd kunnen worden op CPUs, GPUs met CuArrays.jl, en andere platformen zoals gedistribueerde clusters van CPUs en GPUs met DistributedArrays.jl.

Tenslotte tonen we aan dat reeksabstracties ook nuttig kunnen zijn voor algoritmische optimalisaties, zoals het **automatisch afleiden van reeksoperaties op de GPU**. Dergelijke afgeleiden zijn noodzakelijk voor het berekenen van gradiënten zoals ze voorkomen bij neurale netwerken en andere ML (Machine Learning) applicaties. Vaak wordt de programmeur echter beperkt tot welbepaalde operaties waarvoor de afgeleide reeds geïmplementeerd is. Met behulp van automatische differentiatie kan arbitraire code afgeleid worden, maar vaak ten koste van prestatie. Door gebruik te maken van de structuur van de `broadcast` reeksabstractie kunnen we operaties toch efficiënt afleiden, zelfs wanneer de operatie gebruik maakt van dynamisch controleverloop. Onze implementatie van deze techniek in Julia bouwt verder op CuArrays.jl, waardoor ook uitvoering op de GPU mogelijk is.

De compilerinterfaces en bijhorende GPU back end uit dit proefschrift vormen een belangrijke basis voor onderzoek naar programmeren op hoog niveau en abstracties voor hardwareversnellers met een algemene, niet-domeinspecifieke programmeertaal. We hebben aangetoond hoe ons werk het programmeren van GPUs in Julia vereenvoudigt en verbetert, en dit op verschillende abstractieniveaus. Hoewel belangrijk, zijn deze verbeteringen slechts incrementeel en verwachten we dat toekomstig onderzoek de hoogniveau-mogelijkheden van de Julia programmeertaal verder zullen benutten voor krachtiger abstracties en nieuwe GPU programmeermodellen.

Summary

Software development has long been based on hardware that grows exponentially faster, which has allowed application complexity to increase accordingly. This free lunch is over, however, and traditional CPUs (Central Processing Units) don't double their performance every couple of years anymore. As a result, compute-intensive applications have increasingly been relying on hardware accelerators like GPUs (Graphics Processing Units) to satisfy their computational demands.

At the same time, the demand for powerful and complex applications remains. Traditional programming languages, like C and C++, are ill-suited to meet this demand since they require significant programmer expertise. Instead, high-level languages like Python and MATLAB have been gaining popularity as they allow the programmer to focus on application logic and not care about how that application will be executed.

However, it is difficult to use high-level languages to program hardware accelerators like GPUs. First and foremost, the design of these languages often relies on techniques that are not compatible with external accelerators, such as interpretation and dynamic compilation. At the same time, hardware accelerators are mostly used to improve performance, while high-level languages focus on productivity even at the expense of performance. As a result, accelerators are typically programmed with low-level languages that guarantee good performance, in combination with a high-level language to implement the remaining application logic.

Partitioning an application into two or more languages, where one language is used to implement performance-sensitive code and another for the application logic, causes many problems. The use of multiple languages essentially introduces a barrier that prevents reuse, abstraction and optimization of code, but also complicates communication between programmers working on different parts of the codebase.

In this dissertation, we present an alternative approach where **a single high-level language is used to implement both application logic and GPU code**. We do so while maintaining the productivity of the high-level language, without sacrificing the performance of code when it is executed on the hardware accelerator. We start from the existing Julia programming language, a high-level, general-purpose language that was specifically designed for efficient execution on CPUs.

To execute Julia code on an accelerator like a GPU, we need to add a back end to its compiler. This part of the language implementation is responsible for lowering source code to executable machine code. Traditionally, alternative back ends are either added to and integrated with the existing compiler, or implemented as a wholly separate compiler specific to one particular accelerator.

As part of this research, we define **interfaces to the high-level compiler for implementing external back ends**. With these interfaces, it is possible to develop a new back end without having to integrate with the existing compiler and, e.g., comply with its requirements in terms of code quality or licensing. At the same time, existing compiler functionality is reused, which greatly lowers the required effort to implement a new back end. By avoiding a separate compiler, code compatibility between individual back ends is also improved and inevitable differences between implementations are avoided.
We have added these interfaces to the Julia programming languages, and contributed their implementation to the corresponding open-source project. The interfaces grant access to the different IRs (Intermediate Representations) as they exist throughout the compilation process, which includes IR code of the LLVM (Low Level Virtual Machine) library that the Julia compiler uses. To interact efficiently with this library, we have created the *LLVM.jl* package to interface with the LLVM API (Application Programming Interface) from Julia.

We then demonstrate the potential of these interfaces by implementing a **GPU back end for the Julia language**. This back end, available as an open-source package under the name *CUDAnative.jl*, makes it possible to execute Julia code on CUDA (Compute Unified Device Architecture) GPUs. The performance of this high-level GPU code is comparable to equivalent low-level code written in CUDA C, which we demonstrate using the Rodinia benchmark suite for heterogeneous computing. However, high-level GPU programming in Julia is much more productive, less error prone, and requires no additional expertise in terms of other, low-level programming languages.

As a high-level language, Julia has several language features that enable powerful abstractions. For example, automatic specialization of generic code can be used to build higher-order array abstractions. These abstractions can be used to separate abstract operations on arrays from their concrete implementation responsible for allocating memory, executing the operation, etc. This results in concise and readable code, and does not require the high-level programmer to know how the underlying data structure is implemented.

Using the CUDAnative.jl GPU back end, we have conducted research into these **array abstractions for GPUs** without the typical barrier between high-level application code and low-level infrastructure. We have implemented this research in the *CuArrays.jl* package, and used it to demonstrate how array abstractions are powerful enough to implement realistic applications. Courtesy of the Julia's higher-order array abstractions, these implementations are platform-agnostic. We illustrate this by executing the applications on a variety of platforms, including CPUs, GPUs with CuArrays.jl, and distributed clusters of CPUs and GPUs using DistributedArrays.jl.

Finally, we show that array abstractions are also useful for algorithmic optimizations, such as **automatic differentiation of GPU array abstractions**. Differentiation is necessary to compute gradients as they occur in neural network and other ML (Machine Learning) applications. Typically, the programmer is restricted to specific operations for which the derivative has already been implemented. Using automatic differentiation, arbitrary code can be derived, but often at the expense of performance. By exploiting the structure of the `broadcast` array abstraction, we can efficiently differentiate operations even when they use dynamic control flow. Our implementation in Julia builds on CuArrays.jl, which makes it possible to use our technique on the GPU.

The compiler interfaces and accompanying GPU back end as presented in this dissertation provide an important basis for research into high-level programming and abstractions for hardware accelerators using a general-purpose programming language. We have demonstrated this by improving the GPU programming experience in Julia at different levels of abstraction. These improvements are significant but incremental, and we expect future research to fully exploit the high-level features of the Julia language for the purpose of novel abstractions and new GPU programming models.

Contents

List of Tables

List of Figures

List of Acronyms

AD . . . . . Automatic Differentiation

API . . . . . Application Programming Interface

AST . . . . . Abstract Syntax Tree

BLAS . . . . . Basic Linear Algebra Subroutines

CFG . . . . . Control Flow Graph

CI . . . . . Continuous Integration

CPU . . . . . Central Processing Unit

CUB . . . . . CUDA Unbound

CUDA . . . . . Compute Unified Device Architecture

DSL . . . . . Domain Specific Language

FFI . . . . . Foreign Function Interface

GC . . . . . Garbage Collector

GPU . . . . . Graphics Processing Unit

HLO . . . . . High Level Optimizer

HM-LSTM . . . . . Hierarchical Multiscale LSTM

HPC . . . . . High-Performance Computing

I/O . . . . . Input/Output

IACA . . . . . Intel Architecture Code Analyzer

IPC . . . . . Inter-Process Communication

IR . . . . . Intermediate Representation

ISA . . . . . Instruction Set Architecture

JIT . . . . . Just-In-Time

LAPACK . . . . . Linear Algebra Package

LLVM . . . . . Low Level Virtual Machine

LMS . . . . . Lightweight Modular Staging

LOC . . . . . Lines Of Code

MCA . . . . . Machine Code Analyzer

ML . . . . . Machine Learning

MLIR . . . . . Multi-Level IR

MPI . . . . . Message Passing Interface

NUMA . . . . . Non-Unified Memory Architecture

NVTX . . . . . NVIDIA Tools Extensions

NVVM . . . . . NVIDIA Virtual Machine

OJA . . . . . Optimal Jacobian Accumulation

OpenCL . . . . . Open Compute Language

PTX . . . . . Parallel Thread Execution

RDMA . . . . . Remote Direct Memory Access

RPC . . . . . Remote Procedure Call

SDK . . . . . Software Development Kit

SM . . . . . Shared Multiprocessor

SPMD . . . . . Single Program, Multiple Data

SSH . . . . . Secure Shell

TPU . . . . . Tensor Processing Unit

WMMA . . . . . Warp Matrix Multiply and Accumulate

XLA . . . . . Accelerated Linear Algebra

Chapter 1
Introduction

1.1 Context

In recent years, the performance of traditional microprocessors has not increased as it used to in the decades before. Commonly attributed to the end of Moore's law, this evolution drives hardware vendors and software developers alike to look at accelerators, specialized processors that are optimized for specific, typically parallel workloads, and perform much better at them than general-purpose processors [82, 151, 135, 119, 3]. Multiple hardware vendors are working on such accelerators and release many new devices every year. One of the most popular and ubiquitous hardware accelerators are GPUs (Graphics Processing Units): Originally designed for massively-parallel graphics workloads, they are well-suited to support the ever higher computational demands of, among others, the machine-learning community [147].

At the same time, contemporary software is large and complex which necessitates the use of development tools that focus on programmer productivity [146, 64, 54, 85, 75, 143]. High-level programming languages are one such tool, and have gained significant traction over the past decade [145]. One area where high-level programming languages have not broken through, however, is that of accelerator programming: Hardware vendors typically only provide low-level toolchains that focus on reaching peak performance at the cost of developer productivity, while the design of many high-level languages makes them unsuitable for efficient execution on these platforms [95].

1.2 Key Challenges

To use hardware accelerators efficiently with high-level languages, several key challenges need to be addressed. First and foremost, code written in the language needs to be compiled to a representation that is suitable for execution on the accelerator. This conversion should not introduce any run-time overhead, i.e., it should be possible to match the performance of low-level languages despite using a high-level language.

Adding support for a new hardware platform to a given programming language typically involves modifying or rewriting part of its compiler. This is a laborious task, exacerbated by the fact that many new hardware accelerator platforms are released every year. Instead, the design of the programming language and its compiler should be adapted such that new platforms can be targeted more efficiently.

Where low-level programmers are used to writing code at a low abstraction level, high-level languages encourage to think and work with much more abstract programming constructs. In many high-level languages, however, these constructs are constrained by their so-called "two-language" design: Performance-critical codes are implemented in a low-level language, impeding optimizations and restricting usability. We propose to use abstractions that are defined in the language itself, and make it possible to compose the abstraction with arbitrary user code for the purpose of performance, productivity and portability.

1.3 The Julia Programming Language

The contributions of this dissertation are built in and make use of the Julia programming language. This high-level language is built on the LLVM (Low Level Virtual Machine) toolkit, a comprehensive compilation framework used by many industrial-strength compilers. Both Julia and LLVM are open-source projects and are developed by a global community consisting of both industry and academic contributors.

We chose to work with the Julia programming language because of the design of its compiler: Use of the LLVM toolkit and its many hardware back ends facilitates research into hardware accelerators. Crucially, the language was co-designed with an LLVM-based compiler in mind, driving language design decisions that improve the ability to generate high-quality machine code from high-level source code. The compiler and its intermediate representations are also accessible through first-class objects and interfaces, facilitating research into programming languages and extensions thereof. These elements will be discussed in Chapter 2, and are essential to our work on a GPU compiler.

1.4 Structure and Contributions

This dissertation presents abstractions and techniques that enable efficient use of high-level languages on hardware accelerators. Instead of designing a new domain-specific language, or working with libraries written in a low-level language, we adapt and repurpose the existing

general-purpose Julia language and its compiler to provide a productive GPU programming environment with several novel features that improve programmer productivity without sacrificing performance.

Each chapter in this dissertation starts with a short introduction to summarize my personal contributions as well as the scientific novelty of the research presented in that chapter. This is followed by a brief overview of related work as well as necessary background information.

In Chapter 2 I describe the Julia programming language, and my work on compiler interfaces for dynamic compiler back ends. I show how these interfaces can be used to extend the language and its compiler beyond the primary target it was developed for. The interfaces have been contributed to the upstream Julia compiler, and empower multiple back ends and compiler extensions.

In Chapter 3, I describe the GPU back end I have researched and developed on top of these compiler capabilities. This back end is available as the open-source CUDAnative.jl package, and is compatible with the latest stable version of Julia. In this chapter, I also demonstrate the use and describe the implementation of high-level Julia language features for GPUs, and argue that they significantly improve programmer productivity without sacrificing performance.

These chapters are based on a journal publication and a talk:

- Tim Besard, Christophe Foket, and Bjorn De Sutter. "Effective Extensible Programming: Unleashing Julia on GPUs". In: *Transactions on Parallel and Distributed Systems (TPDS)* (2018). ISSN: 1045-9219. DOI: 10.1109/TPDS.2018.2872064. arXiv: 1712.03112 [cs.PL]

- Tim Besard. "Just Compile It: High-level Programming on the GPU with Julia". Presented at the European LLVM Developers Meeting (EuroLLVM). 2019

In Chapter 4 I explain Julia's array interfaces and show how the compiler from Chapter 3 makes it possible to implement these abstractions for GPU arrays, and how they can be used to implement realistic GPU applications. I argue that this approach offers unprecedented flexibility, appealing to both GPU experts and novice programmers. I worked on this approach as part of the CuArrays.jl package.

In Chapter 5 I argue how the design of Julia's array abstractions also facilitates portability of code. I demonstrate how this applies to the infrastructure from Chapter 4, with a variety of use cases.

These chapters are based on a journal publication:

- Tim Besard, Valentin Churavy, Alan Edelman, and Bjorn De Sutter. "Rapid Software Prototyping for Heterogeneous and Distributed Platforms". In: *Advances in Engineering Software (AES)* (2019). DOI: 10.1016/j.advengsoft.2019.02.002

Finally, in Chapter 6 I argue how the structure of array abstractions can be used for high-level algorithmic optimizations. Specifically, I describe my work on automatic differentiation of fused broadcast expressions, executed on the GPU by means of the CUDAnative.jl compiler. The performance of these kernels is competitive to state-of-the-art tensor compilers that are part of current machine-learning frameworks.

This chapter is based on a workshop presentation:

- Jarrett Revels, Tim Besard, Valentin Churavy, Bjorn De Sutter, and Juan Pablo Vielma. "Dynamic Automatic Differentiation of GPU Broadcast Kernels". Presented at the Workshop on Systems for ML at the Conference on Neural Information Processing Systems (NeurIPS). 2018. arXiv: 1810.08297 [cs.MS]

Each of these contributions are available as free and open source software on the GitHub.com public code hosting platform, along with documentation, examples, and unit tests. I describe in Chapter 7 how this open-source nature has facilitated collaboration and enabled new applications based on this research.

1.4.1 Other Contributions

In addition to the contributions I present in this dissertation, I have also contributed to the following academic works during my PhD research:

- Tim Besard, Bjorn De Sutter, Andrés Frías-Velázquez, and Wilfried Philips. "Case Study of Multiple Trace Transform Implementations". In: *International Journal of High Performance Computing Applications (IJHPCA)* 29.4 (2015), pp. 489–505. DOI: 10.1177/1094342015584091

- Mike Innes, Stefan Karpinski, Viral Shah, David Barber, Pontus Stenetorp, Tim Besard, James Bradbury, Valentin Churavy, Simon Danisch, Alan Edelman, Jon Malmaud, Jarrett Revels, and Deniz Yuret. "On Machine Learning and Programming Languages". Presented at the Conference on Systems and Machine Learning (SysML). 2018

I have also organized the third Belgian Julia user meetup in Ghent, and presented the following Julia-related talks throughout my PhD:

- Tim Besard. "High-Level GPU Programming with CUDA.jl". Presented at the first Belgian Julia user meetup (Ghent, Belgium, Sept. 29, 2015)
- Tim Besard. "Julia on the GPU". Presented at the second Belgian Julia user meetup (Brussels, Belgium, Apr. 13, 2016)
- Tim Besard, Valentin Churavy, and Simon Danisch. "GPU Programming with Julia". Presented at JuliaCon 2017 (Berkeley, CA, United States, June 20, 2017)
- Tim Besard. "Programming NVIDIA GPUs in Julia with CUDAnative.jl". Presented at JuliaCon 2017 (Berkeley, CA, United States, June 21, 2017)
- Tim Besard. "Interfacing with LLVM using LLVM.jl". Presented at JuliaCon 2017 (Berkeley, CA, United States, June 22, 2017)
- Tim Besard and Mike Innes. "Julia: A Fresh Approach to GPU Computing". Presented at the GPU Technology Conference (GTC) (San Jose, CA, United States, Mar. 28, 2018)
- Tim Besard. "Effectively using GPUs with Julia". Presented at the third Belgian Julia user meetup (Ghent, Belgium, Dec. 21, 2018)
- Tim Besard. "High-Level Language Design for Extensible Accelerator Programming". Presented at the Workshop on High-Level Programming for Heterogeneous and Hierarchical Parallel Systems (HLPGPU) at the Conference on High Performance and Embedded Architecture and Compilation (HiPEAC) (Valencia, Spain, Jan. 23, 2019)
- Tim Besard. "Introduction to Julia". Presented at the Belgian TensorFlow user meetup (Ghent, Belgium, Feb. 6, 2019)
- Tim Besard. "High-Level Language Design for Extensible Accelerator Programming". Presented at the Workshop on Embedded Multicore Programming at the HiPEAC Computing Systems Week (CSW) (Edinburgh, United Kingdom, Apr. 17, 2019)

Chapter 2
Dynamic Compiler Back Ends

In this chapter, we propose a set of interfaces to the high-level language's compiler for accessing its different IRs (Intermediate Representations) and the processes that generate and optimize that IR. With these interfaces, developers can influence the existing language implementation and, e.g., improve compatibility with new hardware or run-time environments without having to modify the existing implementation or create a new compiler altogether.

We will describe the design of these interfaces, as implemented in the Julia programming language. This high-level language already features powerful multi-stage metaprogramming capabilities as well as other design elements that make it ideally suited for research into language extensions. In Chapter 3 we will demonstrate the use of our added language interfaces to create a GPU back end for Julia.

The scientific contributions of this chapter consist of the design of programming interfaces to the high-level compiler, and their realization as part of the Julia programming language, for the purpose of reducing the effort to create new compiler back ends and extend programming languages to a new platform or environment. These contributions have been published in a peer-reviewed journal.[1]

[1] Tim Besard, Christophe Foket, and Bjorn De Sutter. "Effective Extensible Programming: Unleashing Julia on GPUs". In: *Transactions on Parallel and Distributed Systems (TPDS)* (2018). ISSN: 1045-9219. DOI: 10.1109/TPDS.2018.2872064. arXiv: 1712.03112 [cs.PL].

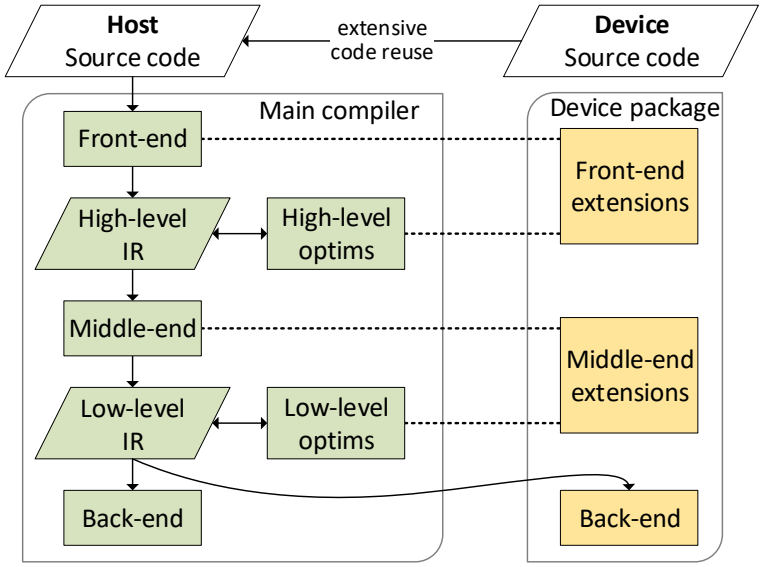

Figure 2.1: Abstract overview of the proposed toolchain that improves reuse of existing compiler functionality. Dashed arrows indicate generic interactions; solid arrows represent the flow of code.

2.1 Proposed Toolchain

Figure 2.1 shows an overview of the proposed toolchain. It features several interfaces to work with the compiler and its intermediate representations. An external device package uses the introduced interfaces to add support for new hardware, without modifying the existing language implementation. For example, it could refuse to generate code for certain language features, such as exceptions or dynamic memory allocations, or replace them with compatible or optimized alternatives.

Such a setup has multiple advantages. For one, it keeps the existing language implementation stable, while new implementations can be developed independently as external packages. This makes it easier to experiment, as these packages do not need to meet the support, quality, or licensing requirements of the existing implementation. It also makes it easier to cope with the rapid development pace of accelerator hardware, providing the means for vendors to contribute more effectively to the language ecosystem.

Another important advantage is the ability to reuse the existing language implementation. Our proposed compiler interfaces make it possible to share functionality between an existing language implementation and external derivatives, avoiding needless reimplementation of compiler

functionality by reconfiguring the existing compiler to generate code that is compatible with the platform at hand. Furthermore, the proposed interfaces not only facilitate external language implementations, but also improve compatibility with existing code as it avoids the inevitable differences between individual compiler implementations.

In some cases, even more reuse of existing infrastructure than suggested in Figure 2.1 is possible. When the back-end compiler used in the general-purpose tool flow can also target accelerators, there is no need to reimplement a device back end in the device package. Instead, that existing back-end compiler can then be used for host and device code, as will be the case for the GPU back end in Chapter 3. But even in other situations it might not be necessary to reimplement a full device back end in the device package: If third-party device code generators can be reused, the device back end only has to translate the low-level IR code to an IR accepted by that third-party code generator.

Conceptually, the compiler interfaces shown in Figure 2.1 are generally applicable. Their actual instantiation, however, will be specific to the host language and accelerator at hand. For this dissertation, we will focus on the design the interfaces around a single language and accelerator platform, respectively the Julia programming language and CUDA (Compute Unified Device Architecture) GPUs. We expect further research into such interfaces to generalize the design and improve reusability across languages and accelerators [58]. Preliminary results include work on targeting Google TPUs (Tensor Processing Units) with XLA.jl [59] and WebAssembly with ExportWebAssembly.jl [138].

2.2 Related Work

There exist several high-level languages with support for programming hardware accelerators through an external back end. However, these implementations often reimplement large parts of the compiler, leading to inevitable differences between both language implementations. For example, Numba is a JIT (Just-In-Time) compiler for Python, building on the CPython reference language implementation. As Figure 2.2 shows, the Numba compiler takes Python bytecode and compiles it to optimized machine code. Due to the high-level nature of Python bytecode, the Numba interpreter and subsequent compilation stages duplicate much functionality from CPython: CFG (Control Flow Graph) construction, type inference, liveness analysis, and implementations of built-in functions. As a result, each release of Numba is tailored to the specifics of certain CPython versions, and needs to be updated when changes

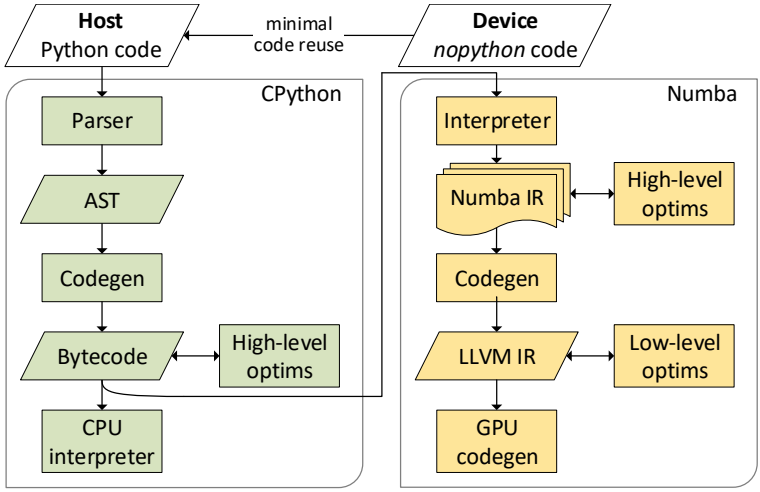

Figure 2.2: Overview of the CPython and Numba compilation processes for host and device code.

are made to the language implementation [87]. The semantics of code also differ slightly depending on whether it is interpreted by CPython or compiled with Numba [87], further impeding compatibility with existing Python code.

The idea of extending an existing language implementation with external functionality has been extensively researched in the past, and has seen a recent revival, but to our knowledge has not focused on extensibility of compiler processes for the purpose of targeting new hardware and environments with minimal code duplication [139, 159]. Furthermore, the proposed interfaces not only facilitate external language implementations, but also improve compatibility with existing code as it avoids the inevitable differences between individual compiler implementations.

2.3 Background: The Julia Programming Language

We chose to implement the proposed toolchain from Section 2.1 in the Julia programming language, a high-level, high-performance dynamic programming language for technical computing [30]. It features a type system with parametric polymorphism, multiple dispatch, powerful metaprogramming capabilities, and other high-level features [26]. The most remarkable aspect of the language and its main implementation is speed: carefully written Julia code performs exceptionally well on traditional

microprocessors, approaching the speed of code written in statically-compiled languages like C or FORTRAN [31, 23, 123]. This is accomplished through compilation, lowering high-level source code to high-quality, stand-alone machine code. Built on top of the LLVM toolkit, which supports a great number of hardware platforms, the language and its open-source implementation form a great basis for our research into high-level language support for hardware accelerators.

2.3.1 Design

Julia's competitive performance originates from clever language design that avoids the typical compilation and execution uncertainties associated with dynamic languages [27, 29]. For example, Julia features a systemic vocabulary of types, with primitive types (integers, floats) mapping onto machine-native representations. The compiler uses type inference to propagate type information throughout the program, tagging locations (variables, temporaries) with the type known at compile time [107, 108]. If a location is fully typed and the layout of that type is known, the compiler can often use stack memory to store its value. In contrast, uncertainty with respect to the type of a location obligates variably-sized heap allocations, with type tags next to values and dynamic checks on those tags as is common in many high-level languages.

Similarly, types are used to express program behavior and eliminate execution uncertainty by means of multiple dispatch. This type of function dispatch, also called multimethods, selects an appropriate method based on the run-time type of all of its arguments. It is a generalization of single-dispatch polymorphism of, e.g., C++, in which only the type of the `this` or `self` object is used to disambiguate a method call. For example, Listing 1 does not use multiple dispatch and defines `intersect` methods that only dispatch on the first argument, returning differently-typed objects by branching on the type of values. Conversely, Listing 2 defines multiple methods that dispatch on all arguments, and consequently are more narrowly-typed in terms of arguments and returned values. In the case of a sufficiently typed call, this enables the compiler to dispatch statically to the correct method and avoid run-time branches, possibly even stack allocating the returned value if its layout is known.

Incorporating all arguments in dispatch also makes it possible to overload methods that would be out-of-reach with single dispatch. For example, Listing 3 defines a dual number type, an extension of real numbers with an epsilon component for the purpose of, e.g., automatic differentiation. Using multiple dispatch, we implement methods for algebraic addition and multiplication that propagate epsilon components

```
function intersect(a::Rect, b)
  if isa(b, Rect)
    c = Rect(...)
  else if isa(b, Line)
    c = Line(...)
  end

  return c::Union{Rect,Line}
end


function intersect(a::Line, b)
  if isa(b, Rect)
    c = Line(...)
  else if isa(b, Line)
    c = Point(...)
  end

  return c::Union{Line,Point}
end
```

Listing 1: Single-dispatch polymorphism and branches that lead to type-unstable functions, returning differently-typed objects based on run-time values.

```
function intersect(a::Rect, b::Rect)
  c = Rect(...)
  return c::Rect
end

function intersect(a::Rect, b::Line)
  c = Line(...)
  return c::Line
end

# similar definitions for intersect(::Line, ...)
```

Listing 2: Functionality of Listing 1 expressed with multiple dispatch.

```
struct Dual{N<:Number} <: Number
  re::N
  ep::N

  # constructor with default value for epsilon component
  Dual{N}(re::N, ep::N=zero(N)) where {N} = new{N}(re, ep)
end


using Base: *, +

*(x::Dual, y::Dual) = Dual(x.re * y.re, x.ep*y.re + x.re*y.ep)
*(x::Dual, y::Number) = Dual(x.re * y, x.ep*y)
*(x::Number, y::Dual) = Dual(x * y.re, x*y.ep)

+(x::Dual, y::Dual) = Dual(x.re + y.re, x.ep + y.ep)
...


A = Dual.(rand(Float64, 2,2))
B = rand(Complex{Int}, 2)
A * B
```

Listing 3: Illustration of multiple dispatch facilitating reuse by allowing fine-grained method overloads.

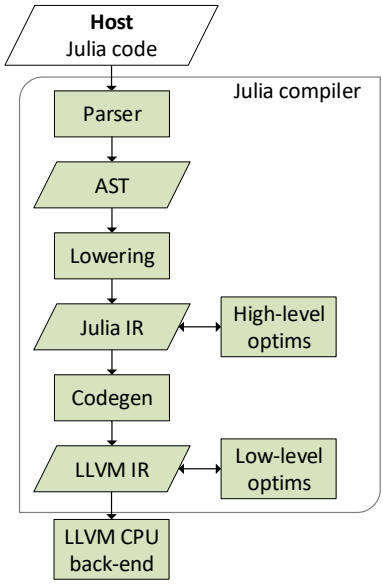

Figure 2.3: Overview of the compilation process for Julia code.

by extending respectively the `+` and `*` functions from the standard library on lines 12 to 16. The definition on line 14 is not possible in a single-dispatch language such as Python, where special methods `__mul__` and `__rmul__` exist specifically for the purpose of defining commutative multiplication as a workaround to overcome the limitations of single-dispatch. Such a workaround does not generalize, however, and fails to compose with optimized functionality such as matrix-matrix multiplication as implemented in NumPy [148]. As a result, users would be forced to reimplement larger pieces of functionality, while complicating reuse of existing functionality. This pattern is especially common for operators, and the Julia standard library uses multiple dispatch extensively to implement these methods [30].

Finally, the Julia language avoids features that would result in guaranteed run-time dynamism, or that would require extensive compiler analyses to avoid so. The Python language, for example, defers certain semantic evaluations such as function binding to the run-time phase. This makes it hard or impossible to generate static code without complicated compilation techniques like partial evaluation or deoptimization. Instead, Julia enforces early function binding.

2.3.2 Implementation

The combination of this design and aggressive specialization on run-time types enables the Julia compiler to generate mostly statically-typed intermediate code, without the need for JIT compilation techniques traditionally used by high-level language implementations (tracing, speculative execution, deoptimization, etc.). This allows the Julia developers to outsource the back-end part of the compilation flow to existing compiler frameworks for static languages. In particular, the Julia IR is a good fit for the LLVM compiler framework, which is commonly used as a basis for industrial-strength compilers for static languages [90]. The Julia compiler targets this framework by emitting LLVM IR as the low-level IR from Figure 2.1, and uses the vast array of LLVM optimization passes (often tailored for or assuming statically-typed straight-line IR) to optimize code and ultimately compile it to high-performance CPU (Central Processing Unit) machine code. Figure 2.3 shows this existing Julia compilation tool flow. In the remainder of this chapter, we refer to it as the main compiler because it is the part of the flow that will generate machine code for the main, general-purpose CPU(s) that serve as a host to accelerators. The last main processing step, CPU code generation, is implemented entirely by means of LLVM. To facilitate interactions with this C++ library, those parts of the Julia compiler that interface with LLVM are also written in C++, making it possible to directly invoke its APIs (Application Programming Interfaces).

As a testament to the performance this design can achieve, most of the Julia standard library is written in Julia itself (with some obvious exceptions for the purpose of reusing existing libraries), while offering good performance [31, 23]. The managed runtime library written in C is very limited and only required for dynamic code that might trigger compilation, and certain language features such as garbage collection and stack unwinding. As a solution to the so-called two-language problem, where multiple programming languages are required to implement the performance and productivity parts of a single application, this greatly lowers to barrier to contributing to the Julia project. Indeed, the number of contributors to the main Julia language repository is greater than that of the Python reference implementation, despite the latter being a significantly older and well-known project.[2]

[2]According to GitHub, the `python/cpython` repository dates back to Aug 9 1990 having 748 contributors as of 2019, while `JuliaLang/julia` dates back to Aug 23 2009 with 797 unique contributors.

	Access	Modify
① AST	✓	✓
② Julia IR	✓	✓
③ LLVM IR	✓	✓
Machine code	✓	indirectly

Table 2.1: Existing metaprogramming interfaces in the Julia programming language to access compiler IRs.

Coincidentally, this design also makes the language well-suited for accelerator programming. Such hardware often features a different architecture and ISA (Instruction Set Architecture), operating independently from the main processor, with control and data transfers happening over a shared bus. In many cases, this makes it hard or impossible to share code, such as runtime libraries, between host and device. With Julia, however, it is entirely possible to write high-level code that compiles to self-contained IR that does not rely on code or data on the host.

2.3.3 Metaprogramming

In addition, Julia features powerful multi-stage metaprogramming and reflection capabilities, as shown in Table 2.1. Source code can be introspected and modified using *macros*, or using the `parse` and `eval` functions. The high-level Julia IR is accessible with the `code_lowered` and `code_typed` reflection functions, and can be modified with *generated functions*. These mechanisms are powerful, flexible, and user-friendly, because they have been co-designed together with the source language and the tool flow in support of metaprogramming and reflection, and because Julia is a homoiconic programming language, i.e., code can be accessed and transformed as structured data from within the language. As such, these interfaces already offer some of the flexibility required to target new hardware, e.g., to define constructs with non-standard semantics or special code generation without the need for new language features. They do not, however, make it possible to influence or replace the processes that generate IR. The resulting IR is often tailored to the host microprocessor, and cannot be used for targeting other hardware.

Low-level LLVM IR can be inspected by invoking `code_llvm` and injected via the `llvmcall` metaprogramming interface. Machine code is accessible through `code_native` and can be inserted indirectly as inline assembly in LLVM IR. These interfaces are much less powerful and flexible, however. Most importantly, the interfaces to LLVM IR only pass

string representations of the IR code. This generic and neutral form of interface fits the separation of concerns between Julia and LLVM, but restricts its usefulness. It suffices for the main compiler because (i) typical metaprogramming and reflection codes do not require access to LLVM IR or machine code, (ii) `llvmcall` is currently only needed by the standard library to inject small, literal snippets of LLVM IR, e.g., to add support for atomics, and (iii) the main compiler is implemented mostly in C++, and thus has direct access to the LLVM APIs.

2.4 Language Interfaces

As a starting point to generate accelerator code, the language should offer access to the code at different stages of compilation, such as ASTs (Abstract Syntax Trees), IRs, and machine code. This makes it possible for programmers to implement functionality that cannot be expressed in source code by generating one of these intermediate forms of code, without the need to alter the language or compiler. It can also be used to transform IR, or use it as a starting point for further compilation.

This kind of multi-stage metaprogramming is a common approach for retargeting a language and its implementation. The technique is typically used to build high-performance abstractions through custom code generation. Examples include MetaML [142], Scala's LMS (Lightweight Modular Staging) and its derivatives [128, 115, 141], etc. However, reuse of compiler functionality is often limited, and the multi-stage programming interfaces typically require invasive syntax or use of special types that may not compose well with existing code. Other approaches, such as Google's MLIR (Multi-Level IR), introduce stages in the compiler IR [91]. Mainly designed to facilitate reuse of compiler functionality across front and back ends, it does however not offer the same flexibility as staging at the language level.

In the case of the Julia programming language, too, there exist several metaprogramming interfaces that provide access to code at different stages of compilation, as shown in Table 2.1. Together with the performance-oriented design described before, they have enabled use of Julia beyond the primary target it was developed for, such as multithreaded systems with ParallelAccelerator.jl [5], and OpenCL (Open Compute Language) GPUs with CLArrays.jl [77]. However, due to the limited scope of the metaprogramming interfaces, these packages have had to reimplement significant parts of the compiler while needing to keep up with upstream development. This is a laborious task. Case in point, both packages have not yet been updated to work with Julia 1.0.

	Reconfigure	Replace
AST	-	-
④ Julia IR	`InferenceParams`	`InferenceHooks`
⑤ LLVM IR	`CodegenParams`	`CodegenHooks`
Machine code	-	-

Table 2.2: Additional interfaces to the Julia compiler for controlling code generation processes.

For efficient and maintainable external language implementations, access to the code generated at each stage of compilation does not suffice. In addition, the metaprogramming interfaces should be augmented with access to the processes that generate that code. For example, when compiling code for an environment that does not support the Julia runtime library, the compiler should not generate code that calls the runtime. A typical case is that of exceptions, which rely on the runtime for stack unwinding and error reporting. Calls to the runtime are generated as part of the code generation process that lowers Julia IR to LLVM IR. To generate code that does not require the runtime library without access to the code generation process, one needs to rid the Julia IR from exceptions, or remove calls to the runtime from the generated LLVM IR. Both approaches are fragile, because they involve modeling behavior and duplicating parts of the main compiler.

2.4.1 Parameters and Hooks

To overcome this problem and improve the reusability of the compiler, we added the four interfaces from Table 2.2 that offer additional control over code generation processes. More specifically, both the lowering of ASTs to Julia IR, and Julia IR to LLVM IR can now be altered through parameters and hooks to reconfigure or replace individual components of these code generation processes. Applied to the above example of code generation without a runtime library, a so-called `CodegenParam` could be used to disallow exceptions altogether, or alternatively a `CodegenHook` could change the generated code not to rely on the runtime library. The GPU back end from Chapter 3 uses these interfaces to replace or customize code generation for exceptions, dynamic memory allocation such as garbage collection, and other functionality that typically requires runtime support libraries, Of course, the nature of these parameters and hooks are specific to the language and its compiler, but the approach is generic and enables extensive reuse of existing functionality.

For now, we have only introduced such interfaces to the processes that generate Julia and LLVM IR; The parsing phase that converts source-code to an AST is superficial and generic enough not to need adjustment for GPU execution, while machine code generation is extremely target-specific and does not offer many opportunities for reuse.

2.4.2 Future Extensions

The GPU back end from Chapter 3 demonstrates how the combination of Julia's multi-stage metaprogramming interfaces with our additions from Table 2.2 is sufficiently powerful to add support for a nontrivial hardware target with enough idiosyncrasies to require various changes to the generated code and how it is processed. At the same time, we are exploring additional interfaces and approaches that might offer better usability in certain scenarios.

For example, `CodegenHook`s are executed during compilation, and their implementation should act accordingly: They do not have direct access to run-time values, but instead have to generate code that works with run-time values. An alternative mechanism is provided by Cassette.jl, not only making it possible to hook (or *overdub*) arbitrary code patterns without the need for dedicated `CodegenHook`s, but also to implement the replacement function with ordinary code that is executed at run-time and as a result has direct access to run-time values [124]. This greatly improves usability, but the approach is not yet mature enough to replace the use of `CodegenHook`s in the back end from Chapter 3. Specifically, Julia's type inference cannot always generate efficient code for overdubbed functions, resulting in GPU-incompatible calls to the Julia runtime library.

Another challenge is that there might not be a singular point to hook the compiler. For example, we explained above how exceptions in Julia are implemented with calls to the Julia runtime library, and how we can hook the compiler to lower exceptions differently. However, there are multiple points in the compiler where these calls are emitted, and there is no guarantee that the `CodegenHook` covers all of them. Furthermore, it is also possible for Julia source code to call the runtime directly. To improve this, we are investigating how to represent these and other operations in the relevant compiler IRs. This would make it possible for a compiler post-processing pass to systematically lower these operations differently.

2.5 Code Generation

The codegen step in the main compiler translates (i.e., lowers) Julia IR into LLVM IR. The C++ part of that code generator directly invokes LLVM IR builder interfaces to do so; the pre-existing part implemented in Julia itself uses the aforementioned string-based interfaces.

For the device package in support of an accelerator, we want to avoid both mechanisms as much as possible. The string-based approach is too fragile; using C++ is not productive for the package developer (likely an expert in Julia and the targeted accelerators, but not necessarily in C++). We hence strive to provide the necessary interfaces and functionality to let developers create new language implementations for accelerators in the Julia language itself, and shielding them from as many LLVM details as possible. This greatly lowers the required effort to support new hardware, as much less code is required when the necessary accelerator-oriented compiler functionality can be written in a productive programming language. As a testament thereto, the GPU support presented in Chapter 3 only requires about 2500 LOC (Lines Of Code).

Furthermore, no changes to the language's compiler are then required, which enables the distribution of the new language implementations (i.e., the device packages) independently of the existing implementation, e.g., with a built-in package manager. The new implementation can be quickly iterated and improved upon, while keeping the core language and its compiler stable. Such a development model is especially interesting for accelerator vendors, where the rapid pace of hardware developments necessitates frequent changes to the toolchain. This contrasts with the relatively slow developments in host compilers and with conservative upgrade policies by system administrators.

To accomplish this, we (1) extend the LLVM IR metaprogramming interface to accept arbitrary IR, and (2) provide a high-level wrapper to the LLVM libraries to efficiently generate that IR from within Julia.

2.5.1 Extended LLVM IR Metaprogramming

The existing `llvmcall` interface uses stringly-typed arguments: Code is passed using literal strings of LLVM IR, a restrictive but often convenient and concise way to insert simple operations such as calls to compiler intrinsics, or to use specific hardware instructions. To make it possible to call arbitrary functions, we extended the interface to accept both an IR snippet and the global declarations that LLVM sometimes requires. Listing 4 shows how this improves the capabilities of the interface, without sacrificing the ease-of-use or conciseness.

```
function floor(x::Float64)
  llvmcall(
    ("""declare double @llvm.floor.f64(double)""",
     """%2 = call double @llvm.floor.f64(double %0)
        ret double %2"""),
    Float64, Tuple{Float64}, x)
end
```

Listing 4: Calling a compiler intrinsic with LLVM IR metaprogramming and stringly-typed IR.

When dealing with more complicated functionality, the IR cannot be easily encoded as a string. One alternative, as developed for the Julia C++ FFI (Foreign Function Interface) [57], is to provide a pointer to the underlying LLVM function object. The Julia compiler will then link that function into the current LLVM module and insert a regular function call, effectively adding the function's IR to the current program.

In the case of the Julia C++ FFI, the LLVM function object is constructed by the Clang library. This library is written in C++, and has full access to the LLVM IRBuilder APIs. The following section will describe how we accomplish this from within Julia, without access to the LLVM C++ APIs.

2.5.2 LLVM Wrapper

To facilitate interactions with LLVM, we have created the *LLVM.jl* package.[3] It provides a high-level Julia wrapper to the LLVM C API, using Julia's built-in C FFI to interact efficiently with the underlying libraries. Although the C API is limited in comparison with the C++ API, it has turned out to be extensive enough to implement the GPU back end from Chapter 3 which includes several analyses and transformations at the LLVM IR level. Where the C API falls short, we add additional API entry points to the Julia runtime library.

The LLVM.jl package can be used to inspect, modify or emit IR code. It greatly improves the usability of the extension interfaces that operate at the IR level. In addition, the package enables reuse of back-end compiler functionality that is part of LLVM, including the vast array of optimization passes that are part of LLVM, or the many back ends to generate machine code from LLVM IR.

[3] Available at https://github.com/maleadt/LLVM.jl

```
@generated function floor(x::Float64)
  eltyp = convert(LLVMType, Float64)

  # create an LLVM module and function
  mod = LLVM.Module("llvmcall")
  ft = LLVM.FunctionType(eltyp, [eltyp,])
  f = LLVM.Function(mod, "floor", ft)

  # get the intrinsic (here, with the same type as the calling function)
  intr = LLVM.Function(mod, "llvm.floor.f64", ft)

  # generate IR
  LLVM.Builder() do builder
    bb = LLVM.BasicBlock(f, "entry")
    LLVM.position!(builder, bb)

    input = LLVM.parameters(f)[1]
    output = call!(builder, intr, [input])

    ret!(builder, output)
  end

  # inject the IR and call it
  fp = convert(Ptr{Cvoid}, LLVM.ref(f))
  return :( llvmcall($fp, Float64, Tuple{Float64}, x) )
end
```

Listing 5: Calling a compiler intrinsic with LLVM IR metaprogramming and LLVM.jl.

Simple Operations

Listing 5 reimplements Listing 4 with functionality from LLVM.jl. It demonstrates of use the LLVM APIs as wrapped by LLVM.jl. Higher-level interfaces that facilitate interoperability with the Julia compiler are provided by LLVM.jl as well. The example is implemented using a generator function, declared with `@generated`, which builds the expressions that should be executed at run time. The generator is expanded at compile time, during type inference: It generates LLVM IR and returns an `llvmcall` expression with an LLVM function object argument, instead of encoding the IR as a literal string.

The example demonstrates how a relatively simple operation, calling the `llvm.floor.f64` compiler intrinsic, is now a convoluted process that involves a lot of boilerplate: converting Julia types to their LLVM counterparts, creating an LLVM module, resolving the intrinsic function, etc. For these kinds of tasks, the string-based interface is still useful.

```
const llvmtypes = IdDict{Any,String}(
    Int64 => "i64",
    # ...
)

@generated function unsafe_load(p::Ptr{T}) where {T}
  eltyp = llvmtypes[T]

  if v"..." <= Base.libllvm_version <= v"..."
    return quote
      llvmcall(
        $"""%2 = inttoptr i64 %0 to $eltyp*
            %3 = load $eltyp, $eltyp* %2, align 8
            ret $eltyp %3""",
        T, Tuple{Ptr{T}}, p)
    end
  else
    error("Unsupported LLVM version")
  end
end
```

Listing 6: Loading values from a pointer with LLVM IR metaprogramming and stringly-typed IR. The semantics of `unsafe_load` have been simplified for brevity.

Complex Operations

More complicated operations, e.g., where the IR depends on the type of the arguments, quickly run into the restrictions of stringly-typed IR. As an example, we consider the implementation of a custom function to load values from a pointer. In Julia, accessing, e.g., an element in an array is implemented by a call to the `unsafe_load` function from the standard Julia library. Its body contains a call to an intrinsic function that is recognized by the code generator in the main Julia compiler, which then lowers it to appropriate LLVM IR code. Implementing an optimized version of `unsafe_load` for loading values on accelerators using the same intrinsics mechanism would similarly require the introduction of one or more intrinsics in the main compiler, and writing the necessary lowering support in C++ using LLVM APIs. This is cumbersome, inflexible, and unproductive.

Again, with LLVM IR metaprogramming we can provide an optimized implementation of `unsafe_load` within Julia and without having to modify the compiler. Listing 6 shows a simplified implementation using the `llvmcall` interface and IR encoded as a string. Here, the use of a generator function illustrates an additional option: Since the gen-

erator is expanded during type inference, the generated expressions can be specialized on the types of the arguments. This is strictly more powerful than using symbolic macros: the pointer argument `p` is not only known by name, but its type `Ptr{T}` as determined by type inference in the Julia compiler is also known to the generator function, with `T` being a type variable referring to the actual run-time element type of the pointer. The generated code hence depends on the inferred types, and can be customized and optimized for them at each invocation of `unsafe_load`.

However, the example runs into the limitations of a string-based metaprogramming interface: We have to hard-code the supported types, the body of the generated function is full of string manipulations, the literal IR snippet is dependent on the LLVM version, etc. This makes the implementation hard to understand and maintain. Moreover, it only supports certain pointer types, and more complicated functionality where the actual IR instructions depend on the argument types instead of only the types of the instructions operands, would be hard or impossible.

The code in Listing 7 implements the same `unsafe_load` function but uses LLVM.jl to build the IR. This does not suffer from the issues mentioned above. The code is generic for all pointer types, is portable across LLVM versions, and does not rely on string manipulations to generate IR.

Furthermore, it is much simpler to generate specialized IR that depends on the types of the arguments. This is useful beyond specialization on plain argument types. We can use the mechanism to encode arbitrary information in the type system for use during code generation. For example, the GPU back end from Chapter 3 encodes compile-time address space information in the type of its device pointers and uses that to generate memory accesses optimized for the different types of memories in a GPU memory hierarchy, as will be explained in Section 3.3.2.

High-Level Compiler Development

Developing a compiler in Julia benefits from the productivity improvements that come with a high-level language. In addition, the LLVM.jl package provides high-level wrappers that facilitate use of the LLVM C APIs. Common operations are mapped onto idiomatic constructs, such as iterators for looping over properties of objects, closures for constructing IR passes, etc.

```
@generated function unsafe_load(p::Ptr{T}) where {T}
  eltyp = convert(LLVMType, T)

  # create a LLVM module and function
  mod = LLVM.Module("llvmcall")
  param_typs = [LLVM.PointerType(eltyp)]
  ft = LLVM.FunctionType(eltyp, param_typs)
  f = LLVM.Function(mod, "unsafe_load", ft)

  # generate IR
  LLVM.Builder() do builder
    bb = LLVM.BasicBlock(f, "entry")
    LLVM.position!(builder, bb)

    ptr = LLVM.parameters(f)[1]
    val = LLVM.load!(builder, ptr) # the actual load

    LLVM.ret!(builder, val)
  end

  # inject the IR and call it
  fp = convert(Ptr{Cvoid}, LLVM.ref(f))
  return :( llvmcall($fp, $T, Tuple{Ptr{$T}}, p) )
end
```

Listing 7: Loading values from a pointer with LLVM IR metaprogramming and LLVM.jl. The semantics of `unsafe_load` have been simplified for brevity.

Listings 8 and 9 compare the code to generate and execute IR that computes the sum of two 32-bit integers, respectively in C++ using the official LLVM C++ API, and in Julia with LLVM.jl. For brevity, the C++ version leaves out the various headers that need to be included. Even for such a short example, the Julia version shows a 33% reduction in LOC, while using much more readable programming patterns. At the same time, the high-level operations map onto identical API calls, i.e., there is no cost to the use of these abstractions.

The package also reconstructs the type hierarchy that exists within LLVM but is not exposed by the C API due to limitations of the C programming language. For example, creating a constant integer value in LLVM using the C++ API yields a value of type `llvm::ConstantInt`. This type is a subclass of `llvm::Value`, but the C API returns an opaque `llvm::Value` pointer without that concrete type information. As a result, the C API needs to provide specialized API functions that differentiate between the different `llvm::Value` subclasses. For example, `LLVMConstAdd` and `LLVMBuildAdd` serve to add respectively two constants or two run-time values. Both functions take arguments that are generic pointers to `llvm::Value`, but are not actually compatible with all such values: Invoking a function with an unsupported value, e.g., calling `LLVMConstAdd` with a pointer to `llvm::Instruction`, which is a subtype of `llvm::Value`, results in a run-time assertion failure. LLVM.jl reconstructs the original C++ type hierarchy and provides a single `add!` function that correctly dispatches based on the type of the arguments. Using wrongly-typed arguments will then result in a graceful method error.

```
1  llvm::LLVMContext context;
2
3  std::unique_ptr<llvm::Module> mod =
4      llvm::make_unique<llvm::Module>("module", context);
5
6  auto T_int32 = llvm::IntegerType::get(context, 32);
7  auto fun_type = llvm::FunctionType::get(T_int32, {T_int32,T_int32},
8                                          false);
9  auto sum = llvm::Function::Create(fun_type,
10                                   llvm::Function::ExternalLinkage,
11                                   "sum", mod.get());
12
13 // generate IR
14 llvm::IRBuilder<> builder(context);
15 {
16   auto *entry = llvm::BasicBlock::Create(context, "entry", sum);
17   builder.SetInsertPoint(entry);
18
19   auto args = sum->arg_begin();
20   llvm::Value *arg1 = &(*args);
21   args = std::next(args);
22   llvm::Value *arg2 = &(*args);
23   auto *tmp = builder.CreateAdd(arg1, arg2, "tmp");
24   builder.CreateRet(tmp);
25
26   mod->dump();
27   verifyFunction(*sum);
28 }
29
30 // execute
31 auto engine = llvm::EngineBuilder(std::move(mod))
32                   .setEngineKind(llvm::EngineKind::Interpreter)
33                   .create();
34 {
35   std::vector<llvm::GenericValue> args(2);
36   args[0].IntVal = llvm::APInt(32, 1, /*isSigned=*/true);
37   args[1].IntVal = llvm::APInt(32, 2, /*isSigned=*/true);
38
39   auto res = engine->runFunction(sum, args);
40   res.IntVal.dump();
41 }
```

Listing 8: Generate and execute code to compute the sum of two 32-bit integers using the LLVM C++ API.

```
mod = LLVM.Module("module")

T_int32 = LLVM.Int32Type()
fun_type = LLVM.FunctionType(T_int32, [T_int32, T_int32])
sum = LLVM.Function(mod, "sum", fun_type)

# generate IR
Builder() do builder
  entry = BasicBlock(sum, "entry")
  position!(builder, entry)

  tmp = add!(builder, parameters(sum)..., "tmp")
  ret!(builder, tmp)

  println(mod)
  verify(mod)
end

# execute
Interpreter(mod) do engine
  args = [GenericValue(T_int32, 1),
          GenericValue(T_int32, 2)]

  res = LLVM.run(engine, sum, args)
  println(convert(Int, res))
end
```

Listing 9: Generate and execute code to compute the sum of two 32-bit integers using LLVM.jl in Julia.

Chapter 3
CUDA Language Implementation

In this chapter, we describe how we used the compiler interfaces from Chapter 2 to extend the existing Julia implementation to generate code for accelerator hardware. We choose to target GPUs, massively parallel accelerators with a distinctive enough architecture to make optimization worthwhile, yet broadly usable for many kinds of applications. Specifically, we focus on NVIDIA GPUs with CUDA, because of the mature toolchain and hardware availability.

We implemented these programming capabilities as part of a regular Julia package that can be installed on any system. The package makes it possible to program GPUs at the same low abstraction level of CUDA C. This is a deliberate choice, as high-level abstractions typically come at a cost, either in terms of performance or by complicating the development of low-level programs. In Chapter 4, we will describe a higher-level programming model that builds on this infrastructure.

The main scientific contribution of this chapter is an implementation of the Julia programming language for GPUs, making it possible to program GPUs in a high-level language with the same performance as low-level C code. By integrating with the existing Julia implementation, GPU code can use many of the language's features and more easily reuse existing code. In this chapter, I also describe the compilation techniques that I developed in order to support high-level language semantics on the GPU. These contributions have been published in a peer-reviewed journal.[1]

[1] Tim Besard, Christophe Foket, and Bjorn De Sutter. "Effective Extensible Programming: Unleashing Julia on GPUs". In: *Transactions on Parallel and Distributed Systems (TPDS)* (2018). ISSN: 1045-9219. DOI: 10.1109/TPDS.2018.2872064. arXiv: 1712.03112 [cs.PL].

3.1 Background and Related Work: Graphics Processing Units

GPUs are massively parallel accelerators that can speed up compute-intensive general-purpose applications. However, that generality is constrained: Most GPUs need to be treated like a coprocessor (with separate memory spaces, controlled by a host processor, mostly unable to perform I/O (Input/Output) operations, etc.), and can only efficiently execute codes that exhibit specific kinds of parallelism. As a result, GPUs are relatively hard to program: Programmers have to deal with the intricacies of coprocessor programming, and need experience with parallel programming to assess if and how specific problems can be solved effectively on a GPU.

At the same time, vendor-supported development environments for programming GPU accelerators typically work with low-level programming languages. NVIDIA's CUDA, for instance, uses CUDA C, while AMD and Intel GPUs are programmed using OpenCL C. The constructs in these low-level languages map closely to available hardware features, making it possible to reach peak performance, as potentially costly abstractions are avoided. However, the lack of such abstractions also complicates GPU programming, not only requiring parallel programming skills and domain knowledge to map the problems, but also low-level programming competence and GPU hardware know-how for the actual implementations [95]. Furthermore, due to a lack of abstractions, these implementations are often hardware-specific, or perform significantly worse on different hardware [53]. Libraries like CUB (CUDA Unbound) [104] or Thrust [70] aim to raise the abstraction level and portability using C++ templates, but fall short due to the low-level nature of C++ and limited applicability across vendor toolkits.

Rather than programming accelerators directly, developers can also use optimized host libraries that are called from the host processor and not directly from the device. Hardware vendors provide such libraries, implementing popular interfaces like BLAS (Basic Linear Algebra Subroutines) [52] and LAPACK (Linear Algebra Package) [4]. There also exist third-party libraries like ArrayFire [101] and ViennaCL [132] that abstract over devices and platforms. These libraries typically export a C API, which eases their use outside of the vendor-supplied development environment. For example, the CUDA BLAS library cuBLAS [113] can be used from Python [47], Julia [79], Octave [102], etc. However, compilers for these languages cannot reason about code in the libraries, and they cannot optimize code across calls to it. Moreover, library-driven

development requires programming in terms of abstractions, which are typically coarse-grained to amortize the cost of configuring the accelerator, initiating execution, etc. Most libraries are also unable to compose their abstractions with custom device code. As a result, library-based programming can be unfit for implementing certain types of applications.

Using high-level languages to program accelerators directly provides a middle ground between high-level host libraries and direct programming with vendor toolkits: Direct programming can offer fine-grained control over compilation and execution, while the use of a high-level language and its abstraction capabilities can improve programmer productivity. However, existing implementations of high-level languages for accelerators do not integrate well with the rest of the ecosystem. Some come in the form of an embedded DSL (Domain Specific Language), such as PyGPU [93] or Copperhead [37], which programmers have to learn and to which they have to adapt their code. Others work with a dedicated data parallel language, like Lift [140], Futhark [68] and Quasar [63], that are well-suited for programming accelerators but cannot be used to implement the rest of the application. Some general-purpose high-level languages can be compiled for accelerators by relying on partial evaluation, such as R with FastR [61] or by using the AnyDSL programming system [92], but that evaluation is not always certain and often forces the user to program at a higher abstraction level. Continuum Analytics' Numba [87] does directly compile the general-purpose Python language, but only supports a subset that is appropriately called `nopython` because it does not support many of Python's high-level features that do not map well onto GPUs, while duplicating compiler functionality from the CPython reference implementation as shown in Figure 2.2. The interfaces from Chapter 2 aim to avoid this duplication, and integrate with the existing language implementation for the purpose of improved code compatibility and an effective compiler implementation.

3.2 Structure

The implementation of Julia for CUDA GPUs is an instantiation of the abstract device package in Figure 2.1. Shown in Figure 3.1, it is distributed as a regular Julia package named *CUDAnative.jl*[2] and does not require any modifications to the underlying Julia compiler. Instead, it reconfigures the existing compiler to generate GPU compatible code.

[2]Available at https://github.com/JuliaGPU/CUDAnative.jl

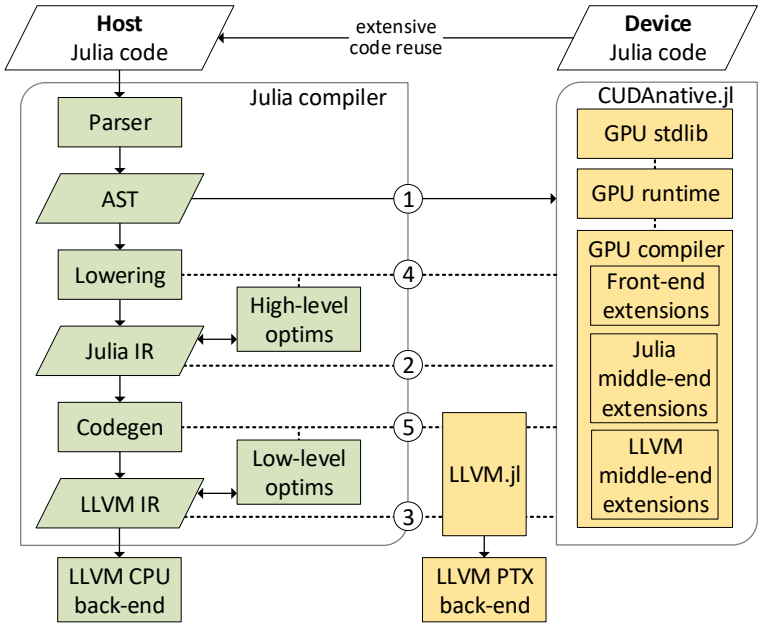

Figure 3.1: Overview of the compilation process for Julia code with CUDAnative.jl by means of the compiler extension interfaces from Chapter 2. Dashed arrows indicate generic interactions; solid arrows represent the flow of code.

The package consists of three major components: a standard library with GPU-specific functionality, a compiler to generate GPU machine code from Julia sources, and a run-time system to invoke the compiler and manage it together with the underlying GPU hardware. Together with the main compiler, which serves as a JIT compiler for host CPUs, this package serves as a JIT compiler for CUDA GPUs.

CUDAnative.jl is designed to enable GPU kernel programming in high-level Julia at the low-level CUDA abstraction level. This is a deliberate design decision, as eagerly raising the abstraction level typically comes at a cost, either in terms of performance or by complicating the development of low-level GPU programs. Nonetheless, we do benefit from the use of a high-level language, and support many of its features in the context of GPU kernel programming in order to improve programmer productivity. Examples include the use of rich array objects with multidimensional indexing instead of raw pointers and fragile pointer arithmetic, support for exceptions that point to the exact source code location in kernel code, the use of those exceptions to trap common bugs like out-of-bounds array accesses or invalid conversions between numbers, and so on. We also rely on Julia's powerful metaprogramming capabilities to design high-level library interfaces, such as an `@atomic` macro to prefix expressions with and make them behave atomically.

The following sections will detail the individual components of CUDAnative.jl, and how they enable or support the compilation of high-level source code to high-performance GPU machine code.

3.3 Standard Library

The CUDAnative.jl standard library focuses on providing definitions for low-level GPU operations that are required for writing effective GPU applications. For example, to access registers containing current thread and block indexes, define synchronization barriers, or allocate shared memory. It also includes certain functions that are specific to the Julia language, and are expected by the Julia compiler to be available. Examples include functions to allocate and free memory, "box" a value (allocate it on the heap and attach a type tag), etc.

As CUDA provides a large amount of these low-level GPU operations, some of which only rarely used by GPU programmers, only a subset is implemented by the CUDAnative.jl standard library. Table 3.1 shows the available functionality as of May 2019, which includes many of the commonly-used CUDA extensions to the C language, certain functionality from the host API that is also available on the device (provided

Category	Available functionality
Indexing	`threadIdx`, `blockDim`, `blockIdx`, `gridDim`, `warpSize`
Memory	static and dynamic shared memory, dynamic allocations, fences, `ldg`
Synchronization	thread synchronization (plain, count, and, or), warp synchronization
Atomics	all atomic operations (`xch`, `add`, `sub`, `inc`, `dec`, `and`, `or`, `xor`, `min`, `max`)
Cooperative groups	cooperative kernels, group synchronization
Dynamic parallelism	dynamic kernels, device synchronization
Warp intrinsics	voting (any, all, ballot), shuffle
Time	`clock`, `nanosleep`
I/O	`printf`, `assert`

Table 3.1: Overview of CUDA functionality provided by the CUDAnative.jl standard library for kernel programming.

by `libcudadevrt`), and common math primitives and bit-manipulation functions as found in the CUDA device library, `libdevice`. Additional functionality is developed when the need arises, and can be done so quickly as will be detailed in the next section.

3.3.1 Implementation

Whereas many languages would implement low-level definitions using compiler intrinsics – built-in functions handled specially by the compiler – the Julia language is expressive enough to implement much of this functionality using Julia code itself. Built-in functions might still be necessary to implement very low-level interactions, but the amount of these functions and their responsibilities are greatly reduced. For example, where CPython implements the `print` function entirely in C as part of the compiler, Julia only relies on a `write` function to write bytes to standard output.

Even when the language is not expressive enough, intrinsics can be avoided by generating lower-level code directly using the metaprogramming interfaces from Table 2.1. For example, atomics are implemented with literal snippets of LLVM IR and wrapped in user-friendly language constructs by means of macros. The GPU standard library in CUDA-

native.jl relies heavily on this type of programming, with help from the LLVM API wrapper from Section 2.5 to facilitate interactions with the LLVM IR.

Julia's expressiveness and metaprogramming capabilities allow for most of its standard library to be written in Julia itself. This makes the standard library much easier to extend or override, e.g., using type-based multiple dispatch as demonstrated in Listing 2. CUDAnative.jl relies on this extensibility to improve GPU compatibility or performance of existing language features, as the next section illustrates.

3.3.2 Pointers with Address Spaces

Pointer address spaces identify, in an abstract way, where pointed-to objects reside. They serve optimization purposes such as identifying pointers to garbage-collected memory, or have a physical meaning depending on the hardware being targeted. In the case of PTX (Parallel Thread Execution) code emitted for NVIDIA GPUs, address spaces differentiate between state spaces: storage areas with particular characteristics in terms of size, access speed, sharing between threads, etc. The PTX compiler uses this information to emit specialized memory operations, such as `ld.global` or `st.shared`. If no address space is specified, untagged operations are emitted (`ld` or `st`) which make the GPU determine the state space at run time by checking against a memory window. While implementing initial CUDA support for Julia, we observed that these untagged operations significantly lower the performance of memory-intensive benchmarks.

LLVM's existing optimizations to infer address spaces across memory operations fall short when memory allocation sites are invisible [155]. Pointers passed to a kernel as arguments, which often happens when entry-point kernels take (pointers to) arrays as arguments, have their allocation in host code, which is invisible to the GPU compiler.

In Julia, pointers are represented by instances of the `Ptr` type. They are regular objects with no special meaning, and operations on these pointers are implemented using normal methods. As such, we can easily define our own pointer type. Listing 10 shows how CUDAnative.jl provides a custom `DevPtr` type representing a pointer with address-space information. By implementing the excepted method interface, which includes the `unsafe_load` method defined on line 8, `DevPtr` objects can be used in place of `Ptr` objects. This then yields specialized memory operations that perform better.

```
# custom pointer with address-space information
struct DevPtr{T,AS}
  ptr::Ptr{T}
end

# loading an indexed value from a pointer
@generated function unsafe_load(p::DevPtr{T,AS},
                                i::Int=1) where {T,AS}
  eltyp = convert(LLVMType, T)

  # create a LLVM module and function
  ...

  # generate IR
  LLVM.Builder() do builder
    ...

    # load from ptr with AS
    ptr = LLVM.gep!(builder, LLVM.parameters(f)[1],
                    [parameters(f)[2]])
    devptr_typ = LLVM.PointerType(eltyp, AS)
    devptr = LLVM.addrspacecast!(builder,ptr,devptr_typ)
    val = LLVM.load!(builder, devptr)

    ...
  end

  # inject the IR and call it
  ...
end
```

Listing 10: Optimized GPU pointers in CUDAnative.jl, building on Listing 7.

The implementation of `unsafe_load` in Listing 10 uses the metaprogramming techniques explained in Section 2.5. A generator function builds specialized LLVM IR and injects it back in the compiler, with the relevant address-space-specific load on line 22. This allows to implement low-level functionality that cannot be expressed using pure Julia code, without the need for additional compiler intrinsics.

Note how the `DevPtr` type from line 2 only contains a single `ptr` field and as such has the exact same memory layout as the existing `Ptr` type. The address space information is only known by the type system, and does not affect the memory representation of run-time pointers.

3.3.3 NVIDIA Device Library

Another important source of low-level GPU operations is `libdevice`, a bitcode library shipped as part of the CUDA toolkit. This library contains common functionality implemented for NVIDIA GPUs, including math primitives, certain special functions, bit manipulation operations, etc. The CUDAnative.jl package provides wrappers for these operations, compatible with counterpart functionality in the Julia standard library. This often raises the abstraction level, and improves usability. For example, `libdevice` provides 4 different functions to compute the absolute value: `__nv_abs` and `__nv_llabs` for respectively 32-bit and 64-bit integers, and similarly `__nv_fabs` and `__nv_fabsf` for 32-bit and 64-bit floating-point values. The Julia wrapper provides the same functionality, but as different methods of a single generic function `abs`.

3.4 Compiler Back End

Together with the main Julia compiler, the CUDAnative.jl infrastructure of Figure 3.1 instantiates the design from Figure 2.1, with the unaltered Julia and LLVM IRs as the high and low-level IRs. Together with host Julia code, device code is processed by the main compiler's parser, which lowers syntactical constructs and expands macros. Both host and device code can include application code as well as library code, and there is no inherent difference between either type of code. There is no need for an explicit annotation or encapsulation of device code, greatly improving opportunities for code reuse. For example, barring use of incompatible language features, much of the Julia standard library can be used to implement device code.

3.4.1 Compilation Process

The main interface for calling functions on a GPU resembles a call to an ordinary Julia function: `@cuda kwargs... function(args...)`, where `kwargs` represents an optional set of keyword arguments that influence the execution: `threads=...` and `blocks=...` to configure the grid, `shmem` and `stream` to respectively set the amount of dynamic shared memory and the stream on which the kernel will be executed, and keywords like `minthreads`, `maxthreads` and `maxregs` to influence code generation by LLVM and the underlying PTX JIT.

Because of the way `@cuda` is implemented in the GPU standard library using metaprogramming, the Julia compiler invokes the GPU compiler in CUDAnative.jl whenever such a call occurs in the code. That GPU compiler then takes over the compilation of the called code. Using the existing interfaces from Table 2.1, the new interfaces from Table 2.2, and the LLVM.jl wrapper, the GPU compiler configures and invokes the existing main compiler components for lowering the (expanded) AST into GPU-oriented Julia IR, for performing high-level optimizations on it, for generating GPU-oriented LLVM IR, and for performing low-level optimizations on that IR. Through the new interfaces, the execution of these compilation steps is repurposed with new GPU-specific functionality that is implemented in CUDAnative.jl. For the front end, most of the GPU-specific functionality actually resides in the GPU standard library as discussed in the previous section; the front-end extensions in the GPU compiler are therefore minimal.

The resulting low-level, GPU-optimized LLVM IR is then compiled to PTX by means of the LLVM NVPTX back end, which is again accessed with the LLVM.jl wrapper package from Section 2.5. This use of an external GPU back end compiler rather than one embedded in the device package diverges slightly from the design in Figure 2.1. For its CPU back end, the Julia compiler already relies on LLVM CPU back ends. So any Julia distribution already includes LLVM. The fact that LLVM can also generate excellent PTX code for CUDA devices when it is fed well-formed and optimized LLVM IR code, voids the need for including a third-party GPU compiler or a reimplementation thereof in the device package [155]. Without putting any burden on system administrators or users to install additional packages or tools, we can simply reuse the LLVM PTX back end.

Alternatively, we could have used the NVIDIA Compiler SDK (Software Development Kit). This SDK is shipped as part of the CUDA toolkit, and provides the NVVM (NVIDIA Virtual Machine) library to compile LLVM IR to PTX assembly. For a variety of reasons however,

we cannot practically use this library. First and foremost, the latest iteration of NVVM only supports IR from LLVM 5.0. Julia uses LLVM 6.0, and there is no backwards compatibility. Worse, the LLVM IR version and subset supported by NVVM changes frequently, e.g., NVVM 1.5 as shipped with CUDA 9.2+ supports IR from LLVM 5.0, NVVM 1.4 from CUDA 9.0 and 9.1 supports LLVM 3.8, etc. This makes it hard or impossible to support multiple versions of the CUDA toolkit. Finally, the closed-source nature of the library is a significant hurdle for compiler research. Access to the source code that implements NVPTX has been important during the development of CUDAnative.jl, both educationally and to add support for IR patterns as generated by the Julia compiler.

3.4.2 Optimization Passes

Before generating machine code, LLVM optimization passes extensively optimize the LLVM IR. These passes are both used to lower Julia specific high-level constructs, such as exception handlers and memory allocations, as well as to optimize the IR for performance. Most of these passes are implemented in C++, either as part of the LLVM framework or within the Julia compiler. However, with LLVM.jl it is also possible to write IR passes in Julia, and CUDAnative.jl provides several such passes. We will now describe the most interesting of these optimizations.

Entry-Point Calling Conventions

One optimization that drastically improves performance, is rewriting the calling convention of entry-point functions. Semantically, Julia passes objects of an immutable type by copying, while mutable types are passed by reference. The actual calling convention as generated by the Julia compiler also passes aggregate immutable types by reference. This does not change semantics, but reduces stack memory usage. In the case of GPU kernel invocations, this means that not the aggregate argument itself, but only a pointer to the argument will be stored in the designated parameter state space (see Section 3.3.2). This space has special semantics that map well onto typical function argument behavior — read-only access instead of read-write, per-kernel sharing instead of per-thread — and typically offers better performance than loading arguments from other memories. However, by passing arguments by reference only the pointer will be loaded from parameter space, and not the underlying objects. In other words, the Julia array objects that themselves contain pointers to the actual buffers to be manipulated by the GPU, are not moved into designated GPU memories to optimize performance.

```
; original device function, with a pointer-to-array as argument
define void @device_function({ [1 x i64], { i64 } }* %array_ptr) {
top:
  %1 = getelementptr inbounds { [1 x i64], { i64 } },
                              { [1 x i64], { i64 } }* %array_ptr,
                              i64 0, i32 1, i32 0
  %2 = bitcast i64* %1 to i64**
  %data_ptr = load i64*, i64** %2
  store i64 1, i64* %data_ptr
  ret void
}

; optimized kernel function, with the actual array object as argument
define void @kernel_function({ [1 x i64], { i64 } } %array) {
top:
  %1 = extractvalue { [1 x i64], { i64 } } %array, 1, 0
  %data_ptr = inttoptr i64 %1 to i64*
  store i64 1, i64* %data_ptr
  ret void
}
```

Listing 11: Simplified LLVM IR for a GPU function that takes a single 1-dimensional array argument and sets the first item to 1. In the case of the kernel function, the calling convention is rewritten to make better use of the parameter state space and avoid a global memory load.

To solve this problem, we let the GPU compiler enforce an adapted calling convention for entry-point kernel functions: Immutable aggregates are also passed by value, instead of by reference. This does not change semantics, as objects of mutable types are still passed by reference. Regular, non-entrypoint functions are not changed, as their arguments are not stored in the parameter state space. Finally, the CUDA-native.jl run-time system passes all immutable arguments by value instead of by reference. This yields a speedup of up to 20% on memory-intensive Rodinia benchmarks.

We implement this change at the LLVM IR level by generating a wrapper function that takes values as arguments instead of references, stores said values in a stack slot, and passes references to those slots to the original entry-point function. After forced inlining and optimization, all redundant operations disappear. Listing 11 shows the effect of this transformation in the case of passing a device array, represented as an aggregate structure containing the array size and a data pointer, to a function with a single parameter. Note that this IR is simplified and

does not show the result of other optimizations such as the explicit use of pointers with address spaces as discussed in Section 3.3.2.

Alternatively, this change could have been implemented by altering code generation for function calls, e.g., using a `CodegenHook`. However, function calling conventions are not well abstracted in LLVM, and as a result are entrenched in many of the Julia code generating routines. It is not worthwhile to try to abstract this functionality only for the purpose of rewriting entry-point functions.

Exception Control Flow

Another set of passes aims to improve compatibility of nontrivial control flow caused by exceptions. Typical kernels, as written in CUDA C, have simple control flow with few function exits. These exits tend to be *warp uniform*, i.e., they are executed by all threads simultaneously.

Code as generated by Julia does not follow this pattern: exceptions can be thrown by individual threads, resulting in *warp divergent* kernel exits. As it turns out, the PTX assembler does not handle this type of control flow very well, and exceptions have been a recurring source of subtle bugs.[3] The phenomenon is related to how GPUs implement control flow by execution masking, which requires the assembler to maintain a branch synchronization stack and use it to re-converge threads after a divergent branch [32]. LLVM provides a so-called CFG structurizer pass, which transforms control flow into a form that is suitable for execution on hardware that uses execution masking, but it fails to deal with divergent control flow that arises from `trap` instructions.

To prevent this issue, CUDAnative.jl provides a number of LLVM passes that ensure Julia exceptions (and other sources of warp divergent control flow) can be executed safely on the GPU:

1. lower exceptions to GPU-compatible IR
2. hide unreachable control flow
3. hide `trap`

The first pass lowers Julia exceptions that call into the CPU runtime library with a GPU-compatible alternative that generates certain output and halts execution. Listing 12 shows how the verbosity of the output changes with the Julia debug level: at the lowest level, intended for performance-critical or bug-free code, execution is halted without any

[3] https://github.com/JuliaGPU/CUDAnative.jl/issues/4 details such an issue, where the interplay between warp divergent control flow and shared memory results in silent corruption of stack memory.

```
$ julia -g0 oob.jl
  ERROR: CUDA error: an illegal instruction was encountered

$ julia -g1 oob.jl
  ERROR: an exception occurred during kernel execution.
         Run Julia on debug level 2 for device stack traces.
  ERROR: CUDA error: an illegal instruction was encountered

$ julia -g2 oob.jl
  ERROR: an exception occurred during kernel execution.
  Stacktrace:
   [1] checkbounds at abstractarray.jl:449
   [2] setindex! at CUDAnative/HLG9m/src/device/array.jl:79
   [3] kernel at oob.jl:6
  ERROR: CUDA error: an illegal instruction was encountered
```

Listing 12: GPU exception behavior at different debug levels.

output except for an `ILLEGAL_INSTRUCTION` error message. At the default optimization level, a call to the GPU print function is inserted, rendering a default error message to standard output. This message is identical for all exceptions, resulting in minimal overhead of a single module-scope constant string and a single function call in the uncommon case where an exception occurs. At higher debug levels we also encode the source-code location, and use that to print a rich exception trace. This bloats the module and code size, but is useful for debugging.

The generated IR for GPU exceptions, as well as other IR produced by the Julia compiler, ends in a call to `trap`. This compiler intrinsic serves to interrupt execution, and is lowered to target-specific code that generates a fault. At the same time, it informs the LLVM optimizer that certain regions of the function, i.e., code that follows the `trap` instruction, are unreachable. It is modeled by means of the `unreachable` compiler intrinsic, and is inferred from the presence of a call to `trap`.

During optimization, LLVM rewrites this so-called unreachable control flow as it expects execution of these unreachable regions to be highly unlikely. Basic blocks that contain the `unreachable` intrinsic are merged and moved to the end of the function. If some of the unreachable blocks were warp divergent, this optimization introduces warp divergent control flow that the PTX assembler does not handle well. This is a common scenario, e.g., with exceptions that stem from array accesses within divergent regions of the function.

To make sure the generated code is compatible with the PTX assembler, the second pass rewrites the function to avoid unreachable code.

```
define void @original(i1) {
entry:
  br i1 %0, label %exit, label %conditional

conditional:
  call void @llvm.trap()
  unreachable

exit:
  ret void
}

define void @rewritten(i1) {
entry:
  br i1 %0, label %exit, label %conditional

conditional:
  call void @llvm.trap()
  br label %exit

exit:
  ret void
}
```

Listing 13: LLVM IR with unreachable control flow, and the rewritten alternative that branches to a closely related basic block.

```
define void @rewritten(i1) {
entry:
  br i1 %0, label %exit, label %conditional

conditional:
  call void asm sideeffect "trap;", ""()
  br label %exit

exit:
  ret void
}
```

Listing 14: LLVM IR with a call to trap using inline assembly.

This prevents LLVM from introducing warp divergent control flow. Listing 13 shows how we insert a branch to a closely-related basic block in the function, typically the reachable successor of any predecessor. This transformation is of course semantically invalid, but that does not matter: By still preceding the branch with a call to the `trap` intrinsic, execution will be halted and the invalid branch will never be executed. The transformation only serves to restructure control flow and avoid introducing warp divergent branches.

Finally, the third pass serves to hide calls to `trap` since LLVM infers that any code following it is unreachable. The solution taken by CUDAnative.jl and shown in Listing 14 is to replace the intrinsic by opaque inline assembly that performs the same operation, but cannot be analyzed by the LLVM optimizer.

3.5 CUDA API Wrapper

The CUDAnative.jl package provides functionality related to compiling code for CUDA GPUs, but another important aspect of GPU applications is to interface directly with the device, e.g., to allocate memory, upload compiled code, and manage execution of kernels. CUDA provides two mostly interchangeable interfaces for this: the low-level driver API, and the runtime API with higher-level semantics and automatic management of certain resources and processes.

The example CUDA C vector addition in Listing 15 uses the runtime API to initialize and upload memory, launch the kernel, and fetch results. The syntax for calling kernels (line 29) hides much of the underlying complexity: setting-up a parameter buffer, initializing the execution configuration, acquiring a reference to the compiled kernel code, etc.

```
1  // auxiliary macros
2  #define cudaCall(err)
3  #define frand() (float)rand() / (float)(RAND_MAX)
4
5  __global__ void vadd(const float *a, const float *b, float *c) {
6    int i = blockIdx.x * blockDim.x + threadIdx.x;
7    c[i] = a[i] + b[i];
8  }
9
10 const int len = 100;
11
12 int main() {
13   // initialize data
14   float *a = new float[len], *b = new float[len];
15   for (int i = 0; i < len; i++) {
16     a[i] = frand();
17     b[i] = frand();
18   }
19   size_t bytesize = len * sizeof(float);
20
21   // allocate and upload
22   float *d_a, *d_b, *d_c;
23   cudaCall(cudaMalloc(&d_a, bytesize));
24   cudaCall(cudaMemcpy(d_a, a, bytesize, cudaMemcpyHostToDevice));
25   cudaCall(cudaMalloc(&d_b, bytesize);
26   cudaCall(cudaMemcpy(d_b, b, bytesize, cudaMemcpyHostToDevice));
27   cudaCall(cudaMalloc(&d_c, bytesize);
28
29   vadd<<<1, len>>>(d_a, d_b, d_c);
30
31   // fetch back
32   float *c = new float[len];
33   cudaCall(cudaMemcpy(c, d_c, bytesize, cudaMemcpyDeviceToHost));
34
35   // clean-up
36   cudaCall(cudaFree(d_c));
37   cudaCall(cudaFree(d_b));
38   cudaCall(cudaFree(d_a));
39   return 0;
40 }
```

Listing 15: Vector addition in CUDA C, using the CUDA run-time API.

```
using CUDAdrv, CuArrays

function vadd(a, b, c)
  i = (blockIdx().x-1) * blockDim().x + threadIdx().x
  c[i] = a[i] + b[i]
  return
end

# initialize data
len = 100
a = rand(Float32, len)
b = rand(Float32, len)

# allocate and upload
d_a = CuArray(a)
d_b = CuArray(b)
d_c = similar(d_a)

@cuda threads=len vadd(d_a, d_b, d_c)

# fetch back
c = Base.Array(d_c)
```

Listing 16: Vector addition in Julia using CUDAdrv.jl and CUDAnative.jl.

To improve the usability of the CUDA API from Julia, we have created the *CUDAdrv.jl* package[4] wrapping the CUDA driver API. It offers the same level of granularity as the driver API, but wrapped in high-level Julia constructs for improved productivity. Similar to the CUDA runtime API, it automates management of resources and processes, but always allows manual control for low-level programming tasks. This makes the wrapper suitable for both application developers and library programmers.

Listing 16 shows an implementation of the vector addition from Listing 15 in Julia using CUDAdrv.jl for all device interactions. It shows how the API wrapper vastly simplifies common operations: Memory allocation and initialization is encoded through different constructors of an array type (as provided by the CuArrays.jl package, but building on CUDAdrv.jl), API error codes are automatically checked and converted to descriptive exceptions, GPU memory is automatically freed by the Julia garbage collector, etc.

[4] Available at https://github.com/JuliaGPU/CUDAdrv.jl

3.6 Run-Time System

While no particular attention was paid so far to the fact that the Julia compiler is a JIT compiler, the CUDAnative.jl run-time system makes it possible to program GPUs using dynamic programming principles, and to invoke those programs almost at the speed of statically-compiled kernels.

3.6.1 Kernel Launching

Whereas launching a kernel from CUDA C is a fully static phenomenon, our `@cuda` Julia macro enables a much more dynamic approach. The GPU compiler is invoked, and hence kernels are compiled, upon their first *use*, i.e., right before an `@cuda` call is first evaluated. At that point, the invoked kernel is specialized and optimized for both the active device and the run-time types of any arguments. For additional, later occurrences of kernel invocations on arguments with different run-time types, newly specialized and optimized code is generated.

The specialized host code that is generated from the `@cuda` invocation in Listing 16 is shown in Listing 17. Lines 3 to 13 contain the result of compile-time computations: Arguments to the `@cuda` macro are decoded during macro expansion, and a generator function (not shown) precomputes values and determines the kernel function signature. This signature can differ from the types of the objects passed to `@cuda`, e.g., the invocation on line 19 in Listing 16 passes `CUDAdrv.Array`s, but the kernel is compiled for GPU-compatible `CuDeviceArray` objects. The run-time conversion of `CUDAdrv.Array` objects to their `CuDeviceArray` counterpart happens as part `cudacall` on line 26.

In addition to recompiling specialized and optimized kernels for changing run-time types, the CUDAnative.jl run-time system keeps track of the so-called *method age*, which indicates the time of definition of the function or any of its dependents. The concept of method age was added to the main Julia compiler in support of dynamic method redefinitions: Whenever a source code fragment is edited, the containing method's age changes, and the new version will be used for future method calls.

CUDAnative.jl also supports this concept of age. At run time, the method age and the active CUDA context are queried. These determine whether a kernel needs to be recompiled: A newer age indicates a redefinition of the method or any callee, while the context determines the active device and owns the resulting kernel object. These properties are hashed with the type signature, and used to query the compilation cache on line 21 of Listing 17. Upon a cache miss, the kernel is compiled

```
# results of compile-time computations
## at parse time
grid = (1,1,1)
block = (len,1,1)
shmem = 0
stream = CuDefaultStream()
kernel = vadd
args = (d_a, d_b, d_c)
## during type inference
types = (CuDeviceArray{Float32,2,AS.Global},
         CuDeviceArray{Float32,2,AS.Global}
         CuDeviceArray{Float32,2,AS.Global})
partial_key = hash(kernel, types)

# determine the run-time environment
age = method_age(kernel, $types)
ctx = CuCurrentContext()
key = hash(partial_key, age, ctx)

# cached compilation
kernel = get!(kernel_cache, key) do
    dev = device(ctx)
    compile(dev, kernel, types)
end

cudacall(kernel, types, args, grid, block, shmem, stream)
```

Listing 17: Lowered code generated from the `@cuda` invocation in Listing 16.

and added to the cache. Finally, control is handed over to CUDAdrv.jl on line 26 where arguments are converted and the kernel is launched.

The above calling sequence has been carefully optimized: Run-time operations are avoided as much as possible, caches are used to prevent redundant computations, code is specialized and aggressively inlined to avoid unnecessary dynamic behavior (e.g., iterating or introspecting arguments or their types), etc. The fast path, i.e., when no device code needs to be compiled, contains only the bare minimum interactions with the Julia compiler and CUDA API. As a result, the time it takes to launch a kernel is almost equivalent to a fully static kernel call in CUDA C (see Section 3.7.6), despite all dynamic programming capabilities. When code does need to be compiled, the time it takes to do so is acceptably low for interactive programming purposes, as will be evaluated in Section 3.7.4.

3.6.2 Interactive Programming

The support for method redefinitions with CUDAnative.jl makes it possible to program a GPU interactively, e.g., using Project Jupyter, a popular programming environment among scientists and teachers for programming interactively in Julia, Python or R [118, 122]. The environment revolves around so-called notebooks, documents that can contain both computer code, the results from evaluating that code, and other rich-text elements. The contents of these notebooks can be changed or evaluated in any order and at any time, requiring a great deal of flexibility from the underlying execution environment, e.g., to recompile code whenever it has been edited. CUDAnative.jl makes it possible to use this highly dynamic style of programming in combination with GPUs, for example to develop GPU kernels by iteratively redefining device methods and evaluating the output or performance.

This capability provides an excellent demonstration of the advantages of (i) our vision of adding interfaces for main compiler repurposing, and (ii) our implementation of CUDA with a pure Julia device package. This enables tight integration of GPU support into the existing compiler, which in turn makes the integration of GPU support in a project like Jupyter seamless, both for the developers of the GPU support, and from the perspective of Jupyter users, who get the same interactivity for host and GPU programming. All we needed was a careful design of the compilation cache, which was needed anyway, and 5 lines of code to include the method age in the hashes used to access the cache.

3.6.3 Reflection and Introspection

The preexisting implementation of Julia for CPUs offers a variety of high-level tools that help the programmer work with the compiler and the code that it generates. For example, functions are represented by first-class objects that can be assigned and passed around, the compiler can be queried to retrieve source code for these functions or inspect the generated code, several logging features can be activated to have the compiler report or time each action, etc.

CUDAnative.jl offers similar functionality that offers an unprecedented window into the GPU compiler. Listing 18 demonstrates the different `@device_code` macros: Each of them configure the CUDAnative.jl compiler to save and return one specific form of intermediate code as it exist during compilation, and can be used to debug compilation failures, optimize performance, or ensure GPU execution. The macros mimic the well-known `@code` macros as they exist for the Julia CPU compiler, with the difference that they report on all executed GPU kernels that occur during evaluation of the right-hand side expression. This makes it possible to reflect on complex applications without direct access to the expressions that launch kernels: Instead of wrapping the innermost GPU kernel invocations within a codebase, the user only need to surround any outer expression, e.g., the call to `main`, with a `@device_code` macro to catch all consequent GPU executions.

Where the `@device_code` macros are primarily intended as a debugging tool, CUDAnative.jl also offers first-class introspection functionality that can be used as part of an application. Listing 19 demonstrates the use of kernel objects: Compiled kernels are represented by first-class objects for which properties like the memory usage, register usage, or maximal thread usage can be queried. These properties can be used to determine the optimal launch configuration, compile the kernel differently (e.g., coaxing the PTX JIT into using fewer registers by specifying the `maxregs` option to `cufunction`), or switch to a different implementation of the algorithm. Such low-level control can be useful to ensure compatibility with a low-end GPU, to maximize occupancy, etc.

3.7 Evaluation

To asses the performance of low-level GPU applications written in Julia, we have ported CUDA C benchmarks from the Rodinia benchmark suite for heterogeneous computing [41] to Julia with CUDAnative.jl.[5]

[5] Available at https://github.com/JuliaParallel/rodinia/

```
julia> @device_code_lowered @cuda threads=len vadd(d_a, d_b, d_c)
1-element Array{Any,1}:
 CodeInfo(
1   ...
|   return
)

julia> @device_code_typed @cuda threads=len vadd(d_a, d_b, d_c)
1-element Array{Any,1}:
 CodeInfo(
1   ...
|   return
) => Nothing

julia> @device_code_llvm @cuda threads=len vadd(d_a, d_b, d_c)
define void @ptxcall_vadd({ [1 x i64], { i64 } } %a,
                          { [1 x i64], { i64 } } %b,
                          { [1 x i64], { i64 } } %c) {
entry:
    ...
    ret void
}

julia> @device_code_ptx @cuda threads=len vadd(d_a, d_b, d_c)
.visible .entry ptxcall_vadd(.param .b8 a[16],
                             .param .b8 b[16],
                             .param .b8 c[16]) {
    ...
    ret;
}

julia> @device_code_sass @cuda threads=len vadd(d_a, d_b, d_c)
ptxcall_vadd:
    ...
    RET;
```

Listing 18: Introspecting code generated by the CUDAnative.jl compiler for the kernel invocation in Listing 16.

```
julia> args = (d_a, d_b, d_c)
(object of type CuArray{2,Float32},
 object of type CuArray{2,Float32},
 object of type CuArray{2,Float32})

julia> compatible_args = cudaconvert.(args)
(object of type CuDeviceArray{2,Float32,AS.Global},
 object of type CuDeviceArray{2,Float32,AS.Global},
 object of type CuDeviceArray{2,Float32,AS.Global})

julia> types = typeof.(compatible_args)
(CuDeviceArray{Float32,1,CUDAnative.AS.Global},
 CuDeviceArray{Float32,1,CUDAnative.AS.Global},
 CuDeviceArray{Float32,1,CUDAnative.AS.Global})

julia> kernel_object = cufunction(vadd, types)
object of type CUDAnative.Kernel


julia> CUDAnative.registers(kernel_object)
24

julia> CUDAnative.memory(kernel_object)
(local = 8, shared = 0, constant = 0)

julia> CUDAnative.maxthreads(kernel_object)
1024
```

Listing 19: Construction of a kernel object and introspection of its properties, based on the kernel from Listing 16.

To enable an accurate performance comparison, we have translated the kernel code as literally as possible, without performing optimizations or changes to make them more Julia idiomatic. Still, there are plenty of semantic differences between the C and Julia language that required a significant effort: Arrays are represented by objects instead of pointers (ruling out pointer arithmetic), indexing is column major and uses 1-based indices, types of literals as well as their promotion behavior differs, etc. Having limited resources, we selected the smallest benchmarks of the suite (in terms of line count), also taking into account the use of GPU features that are not yet supported by CUDAnative.jl, such as constant memory. Apart from the latter, our selection of benchmark is in no way biased by the features of their kernels.

The non-kernel code was mostly translated literally from C to Julia, sometimes at the expense of performance. For example, many benchmarks initialize matrices with double for loops, processing elements in row-major order. As Julia uses column-major storage, we changed the iteration order of these loops, unless that would result in a major redesign of the benchmark.

3.7.1 Experimental Set-Up

CUDA C code is compiled with the NVIDIA CUDA compiler version 9.1.85, NVIDIA driver 390.59, and Linux 4.9.0 from Debian Stretch (64-bit). Julia measurements are done with the first release candidate of Julia 0.7 and publicly available Julia packages CUDAnative.jl 0.8.4, CUDAdrv.jl 0.8.4 and LLVM.jl 0.9.12 using LLVM 6.0. Our test system contains an NVIDIA GeForce GTX 1080 GPU, two quad-core Intel Xeon E5-2637 v2 CPUs, and 64 GiB of DDR3 ECC memory.

3.7.2 Methodology

All execution times reported in Tables 3.2 and 3.3 are in milliseconds, and show the mean value with error margins determined by propagating the standard deviation across operations [62]. The results are obtained by launching each benchmark multiple times on a fully idle machine, where each process first runs the application code 5 times to warm up the system. We measure execution times with the `nvprof` tool from the CUDA toolkit, and use the NVTX (NVIDIA Tools Extensions) library to extend the profile with CPU timings. Benchmark inputs are the defaults parameters from Rodinia 3.1, as shown in Table 3.4.

Some interesting observations can be made up-front. First, the often near-zero fractions in columns (b/a) and (d/c) in Table 3.2 and Ta-

benchmark	kernels	CUDA C					
		total		GPU kernel			
		LOC	(a) run time	LOC	PTX	(b) run time	(b/a)
lud	3	263	1.1 ±6.2%	126	1588	0.42 ±0.7%	39%
particlefilter	4	611	60.1 ±0.5%	152	2063	38.60 ±0.2%	64%
backprop	2	631	40.6 ±4.7%	56	230	0.19 ±0.3%	0%
nw	2	340	50.9 ±15.2%	118	1341	1.89 ±0.2%	4%
leukocyte	3	1665	181.8 ±1.1%	384	1474	86.09 ±0.1%	47%
pathfinder	1	166	196.9 ±2.3%	49	163	0.25 ±0.5%	0%
hotspot	1	265	137.4 ±1.3%	91	237	0.11 ±0.5%	0%
nn	1	270	253.6 ±1.6%	11	53	0.03 ±0.7%	0%
bfs	2	184	1278.1 ±0.9%	33	143	6.49 ±0.2%	1%
streamcluster	1	952	6457.2 ±5.1%	28	202	510.61 ±0.5%	8%
average			±3.9%			±0.4%	16%

Table 3.2: Features and performance of selected Rodinia benchmarks implemented in CUDA C.

benchmark	kernels	JULIA									
		total		GPU kernels				JIT compilation		total - JIT	
		LOC	(c) run time	LOC	PTX	(d) run time	(d/c)	(e) run time	(e/c)	(f = c-e) run time	(f/c)
lud	3	202	478.9 ±1.0%	110	2586	0.48 ±0.4%	0.10%	477.5 ±1.0%	100%	1.4 ±5.4%	0.29%
particlefilter	4	409	522.5 ±1.8%	123	1304	34.30 ±1.3%	6.56%	480.5 ±1.8%	92%	42.0 ±2.3%	8.04%
backprop	2	317	105.8 ±3.7%	54	257	0.17 ±0.8%	0.16%	62.3 ±1.8%	59%	43.5 ±6.4%	41.14%
nw	2	255	214.7 ±3.0%	110	633	1.94 ±0.2%	0.90%	169.1 ±1.2%	79%	45.6 ±9.8%	21.26%
leukocyte	3	856	725.2 ±2.9%	275	1344	67.92 ±0.1%	9.37%	434.0 ±1.7%	60%	291.2 ±4.6%	40.16%
pathfinder	1	140	237.2 ±2.8%	52	152	0.25 ±0.8%	0.10%	38.7 ±2.1%	16%	198.6 ±3.0%	83.70%
hotspot	1	228	184.7 ±1.7%	87	247	0.11 ±0.3%	0.06%	65.4 ±1.7%	35%	119.3 ±1.7%	64.61%
nn	1	148	505.7 ±3.7%	11	61	0.03 ±1.6%	0.01%	100.1 ±1.8%	20%	405.6 ±4.2%	80.21%
bfs	2	135	1688.4 ±2.2%	28	161	6.03 ±0.1%	0.36%	40.4 ±1.5%	2%	1648.0 ±2.3%	97.61%
streamcluster	1	647	6827.6 ±3.4%	30	162	501.95 ±0.0%	7.35%	31.2 ±1.8%	0%	6796.4 ±3.4%	99.54%
average			±2.6%			±0.6%		±1.6%		±4.3%	

Table 3.3: Features and performance of selected Rodinia benchmarks implemented in Julia using CUDAnative.jl

Benchmark	Input parameters
backprop	`input_points`=65536
bfs	`file`=`graph1M.txt` (graph with 1048576 nodes)
hotspot	`grid_rows_cols`=512 `pyramid_height`=2 `sim_time`=2
	`temperature_file`=`temp_512` (matrix of 512x512 values)
	`power_data`=`power_512` (matrix of 512x512 values)
leukocyte	`frames`=10 `input`=`testfile.avi` (176MB video)
lud	`matrix_dim`=256
nn	`records`=5 `latitude`=30 `longitude`=90
	`input`=`list640k_64.txt` (64 × 100k hurricanes)
nw	`max_rows`=2048 `max_rows`=2048 `penalty`=10
particlefilter	`dim_x`=128 `dim_y`=128 `frames`=10 `particles`=10000
	(executed in double-precision mode)
pathfinder	`cols`=100000 `rows`=100 `pyramid_height`=20
streamcluster	`min_centers`=10 `max_centers`=20 `dimensions`=256
	`points`=65536 `chunk_size`=65536 `cluster_size`=1000

Table 3.4: Default input parameters from Rodinia 3.1

ble 3.3 indicate that the Rodinia benchmarks spend only a fraction of their time in GPU kernels. As our contributions are almost exclusively in the generation of those kernels, it follows that total execution time of Rodinia benchmarks is not a good metric to evaluate our contributions. Secondly, the small error rates on kernel execution times reveal that, despite their short run times, they are very well suited for a reliable comparison of kernel performance, i.e., for assessing our contributions. Finally, those short run times result from running kernels on unrealistically small data sets, and are not representative of kernel run times in real-world GPU deployments. The ratio between the kernel run times and other contributions to the total execution times is therefore mostly meaningless.

3.7.3 Kernel Performance

Table 3.5, Figure 3.2 show how, on average, we measure a speedup of 4% compared to CUDA C kernels compiled with `nvcc`, the official compiler by NVIDIA for CUDA C code. This shows how CUDAnative.jl can be realistically used for GPU kernel programming. Furthermore, the result is close to the relative speedup of 1% as achieved by `gpucc` on a wider

benchmark	kernels	JULIA/CUDA C		
		GPU kernels	total	total - JIT
		(d/b)	(c/a)	(f/a)
lud	3	1,14	439,49	1,26
particlefilter	4	0,89	8,69	0,70
backprop	2	0,88	2,60	1,07
nw	2	1,03	4,22	0,90
leukocyte	3	0,79	3,99	1,60
pathfinder	1	1,01	1,21	1,01
hotspot	1	1,03	1,34	0,87
nn	1	1,00	1,99	1,60
bfs	2	0,93	1,32	1,29
streamcluster	1	0,98	1,06	1,05
average		**0,96**	3,87	1,10

Table 3.5: Performance comparison of selected Rodinia benchmarks implemented in Julia using CUDAnative.jl vs CUDA C.

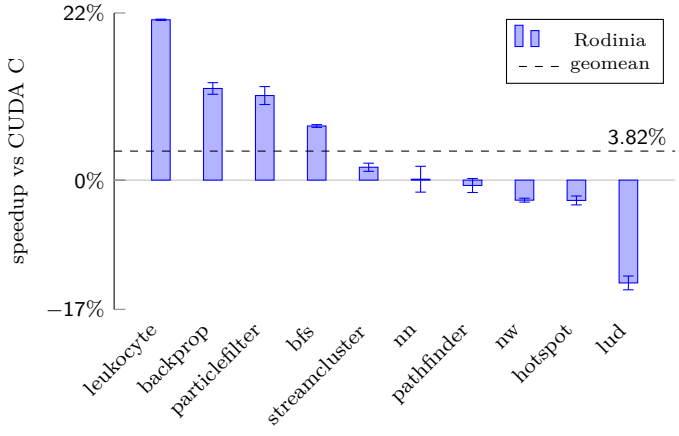

Figure 3.2: Visualization of the GPU kernel performance ratio from Table 3.5 of selected Rodinia benchmarks implemented in Julia using CUDAnative.jl vs CUDA C.

range of Rodinia benchmarks [155]. `gpucc` is an open-source compiler for CUDA C code, built on the same LLVM back end as CUDAnative.jl. We can conclude from this result that using Julia for GPU kernel programming does not incur a substantial slowdown. The difference in performance compared to `gpucc` can be attributed to only testing a subset of Rodinia, but also to improved vectorization due to optimized variable alignment characteristics that differ from C.

The slowdown as observed with the `lud` benchmark can be attributed to Julia defaulting to 64-bit integers on 64-bit hardware. In contrast, C language compilers only use 32 bits to represent `int` values. The large speedups as seen with `particlefiler`, `backprop` and `leukocyte` are a result of load-store vectorization at the LLVM level whereas `nvcc` relies on `ptxas` to perform this optimization on PTX code. These optimizations are peephole transformations that work on low-level machine code, and as such are less powerful than optimizations by LLVM on its higher-level compiler IR.

3.7.4 Compilation Overhead

Looking at total application performance in Table 3.5, we measure an overwhelming 287% overhead. Much of that overhead can be attributed to the time it takes to JIT compile code. However, consistently taking less than a second to compile the kernels (column e in Table 3.3), these large fractions of the total run time spent on JIT compilation are more a side effect of the benchmarks' short-running kernels than of the excessive compilation times. Real-world GPU applications typically execute the same kernels on the same types of data over and over again, for which kernels only need to be compiled once. Such sub-second JIT compilation times will be amortized (almost) completely in real-world applications. Even within our range of benchmarks running anywhere in between 1 ms and 6827 ms, this amortization can already be observed in the decreasing numbers in column (e/c) of Table 3.3. We thus conclude that the Julia JIT compilation is fast enough not to impose a performance burden for real-world GPU applications.

It is noteworthy that almost no JIT compilation time is spent outside the kernels: almost all non-kernel code, i.e, almost all host code, is either compiled ahead of time or interpreted according to heuristics in the Julia compiler [109, 81].

In interactive programming settings such as Jupyter, kernels are recompiled when their code has been edited, or when other code edits result in kernels being invoked on new data types. Moreover, while code is still being developed, it will often be invoked on smaller data sets for

testing. In such a setting, individual kernel JIT compilation times do matter, as they are not amortized. For the selected Rodinia kernels, Table 3.3 shows how individual kernel compilation times range from 12 to 283 ms, with a median compilation time of 70 ms; this is sufficiently low for realistic interactive development.

The JIT compilation times are strongly correlated with the sizes of the kernels in number of PTX instructions, more so than with the number of Julia LOC, with correlation coefficients of respectively 0.91 and 0.69. One reason is loop unrolling: when the compiler unrolls loops, even small ones in LOC can become big in terms of IR that the compiler needs to handle. The PTX sizes for Julia kernels are typically somewhat larger than, and in some cases notably smaller than, the CUDA sizes, without this resulting in comparably large performance differences. The reason is that PTX code, be it statically-compiled CUDA C or JIT-compiled Julia, is further optimized by the PTX assembler as part of the CUDA driver before execution. That final optimization step performs several peephole optimizations, removing most remaining differences between PTX versions of the kernels.

3.7.5 Application Performance

If we disregard the compilation overhead, Table 3.5 shows how total application performance of the selected Rodinia benchmarks performs 10% slower in Julia with CUDAnative.jl than it does in C with CUDA C. On the one hand, these times are not very relevant as our contributions are not related directly to host code. On the other hand, the numbers show that the host computation times of Julia code compare pretty well to those of C code, despite the fact that we invested a limited effort to optimize the literally translated Julia code.

Where benchmarks perform considerably worse, this is due to the ports being literal translations of C code. For example, `leukocyte` depends on global mutable data that necessitates heap allocations given Julia semantics, and processes data in row-major order. The `nn` benchmark generates large amounts of heap-allocated strings during parsing. High-performance text parsing functionality from, e.g., the TextParse.jl package could be used instead, but such an implementation would differ significantly from the C version of the benchmark. On average, our measurements show that for the host part of Julia GPU applications, performance comparable to that of programs written in C can be expected. This result is consistent with existing literature [31, 23, 123].

	GPU time	CPU time
CUDA C	(5.88 ± 0.23) µs	(12.77 ± 0.23) µs
CUDAdrv.jl	(7.28 ± 0.76) µs	(13.85 ± 0.78) µs
CUDAnative.jl	(7.19 ± 0.46) µs	(13.38 ± 0.52) µs

Table 3.6: GPU and CPU execution times for launching an empty kernel from CUDA C and Julia.

For GPU applications, which also performs GPU-related tasks in the form of API interactions such as memory copies, device configuration, etc., this result is novel. It stems from the design of CUDAdrv.jl: Although the Julia wrappers to the CUDA APIs are high-level, they work on the same abstraction level as CUDA. This improves usability, but maintains flexibility while avoiding performance traps. The actual library calls use Julia's high-performance C FFI, which generates code to call C functions without run time overhead.

3.7.6 Run-Time System Performance

The comparable host-GPU interaction performance between CUDA C and CUDAnative.jl applications also results from their comparable kernel launching times. With statically compiled C, the run-time cost of launching a kernel is dominated by the CUDA libraries. In the case of CUDAnative.jl, Section 3.6 described how launching a kernel entails many more tasks with the goal of a highly dynamic programming environment: converting arguments, (re)compiling code, instantiating a CUDA module and function object, etc. These code paths have been carefully optimized to avoid run-time overhead as much as possible.

To determine this overhead, we launch an empty kernel and measure the elapsed execution time, both on the GPU using CUDA events and on the CPU using regular wall-clock timers. Table 3.6 shows these measurements for statically-compiled C code using the CUDA driver API, Julia code performing the exact same static operations with CUDAdrv.jl, and dynamic Julia code using CUDAnative.jl to compile and execute the empty kernel. Neither of the GPU and CPU time measurements show significant overhead when only using CUDAdrv.jl. When using CUDAnative.jl, which internally uses CUDAdrv.jl, minimal overhead is introduced by the check for the current method age. We consider this negligible: In the case of realistic kernels it is dwarfed by the time to copy the kernel parameter buffer to the device.

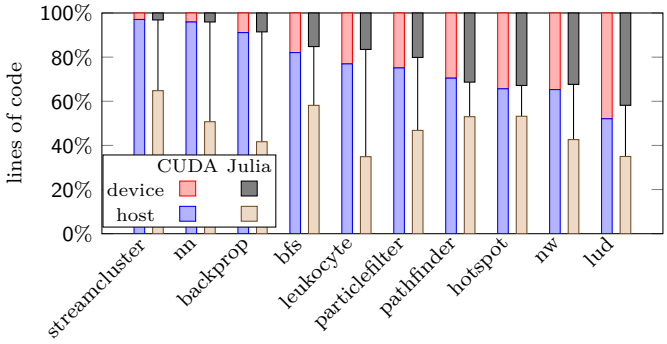

Figure 3.3: Lines of host and device code of selected Rodinia benchmarks, normalized against the total LOC of their CUDA C implementations. On average, the Julia versions are 32% shorter: device code is reduced by 8%, while the amount of host code is reduced by 38%.

3.7.7 Lines of Code

As discussed above, we maintained the semantics of the original CUDA C benchmarks in their Julia translation. Even then, we were able to express many operations much more succinctly. For example, interactions with the file system (reading paths and processing their contents), memory management, generating output, etc. can be written in fewer lines of code, relying on higher-level language features like string interpolation or scoped resource cleanup. As a technical computing language, Julia also provides high-level tools to process data, such as an interface to BLAS, and syntax to express operations on multidimensional data compactly [76]. Finally, the CUDA APIs are similarly accessible through high-level wrappers, which are semantically equivalent to the C APIs but allow for much more succinct invocations.

As a result, the LOC counts visualized in Figure 3.3 show an average reduction of 37% in LOC for the host part of the benchmarks. The device LOC only decreases 8%, as GPU code does not interface with complicated APIs, and as it does not use multidimensional expressions but typically processes scalar items as per the GPU's execution model. Even so, this style of Julia GPU programming significantly improves the programming experience, with, e.g., dynamic types, checked arithmetic, an improved programming environment, etc. On average, the total application LOC is reduced by 31%. Although this is indicative of a reduction in developer effort, familiary with GPUs and their programming model is still a requirement. The next chapter will discuss a higher-level approach, with abstractions that obviate GPU experience.

Chapter 4
High-Level Array Programming with GPUs

This chapter will discuss high-level array programming, where computations are expressed through operations on arrays. This is a programming style that is commonly used during application prototyping. The Julia programming language greatly improves the usability and capability of these array operations by relying on a compiler to provide higher-order abstractions. We will demonstrate the use of these array abstractions using several engineering applications.

Programming with arrays trivially exposes plenty of parallelism, and is a good fit for programming GPUs without GPU programming experience. As such, there exist many array programming libraries that target these parallel hardware accelerators. We have developed a similar solution for Julia, building on the GPU compiler from Chapter 3. Our implementation also covers Julia's higher-order array abstractions in order to improve the GPU programming experience. The performance of these abstractions will be evaluated in Chapter 5.

Initial research into a library with GPU array abstractions building on CUDAnative.jl was done by Mike Innes of Julia Computing, providing proof-of-concept implementations for the purpose of accelerating the Flux.jl machine-learning package. The author of this dissertation has continued that research, improving performance and adding abstractions that are compatible with a wider range of array applications, and is now the principal author of the CuArrays.jl package.

The main scientific contribution of this chapter is an implementation of Julia's array abstractions for GPUs using the compiler from Chapter 3. The design of these abstractions offers run-time flexibility that is novel in the context of GPU programming. We demonstrate this flexibility with realistic applications from the domain of computer engineering. These contributions have been published in a peer-reviewed journal.[1]

[1] Tim Besard, Valentin Churavy, Alan Edelman, and Bjorn De Sutter. "Rapid Software Prototyping for Heterogeneous and Distributed Platforms". In: *Advances in Engineering Software (AES)* (2019). DOI: 10.1016/j.advengsoft.2019.02.002.

4.1 Example Applications

Data science and engineering problems are commonly expressed in terms of vectorized operations, especially during initial prototyping. This natural, concise representation makes it easier to iterate over different implementations. They avoid the typical non-functional boilerplate of scalar processing of data items such as specifying loop bounds, indexing calculations, etc.

To illustrate this, we introduce three examples that are relevant to computer-based engineering techniques: the power method to calculate eigenvalues, gradient descent to minimize a loss function, and the Kronecker product of two matrices. These examples represent different levels of application complexity, and demonstrate different aspects of array programming. We have implemented the examples in Julia, using high-level, idiomatic code that stays as close as possible to the original mathematical descriptions.

4.1.1 Power Iteration

The power method serves as the first, simplest example. This is an eigenvalue algorithm, approximating the dominant eigenvalue of a diagonalizable matrix by means of an iterative algorithm [105]. The associated eigenvalue is then computed using the Rayleigh quotient. The Julia implementation in Listing 20 mirrors the high-level descriptions of these algorithms from the corresponding Wikipedia pages,[2,3] and uses simple operations on arrays, such as the dot product, matrix-vector multiplication, the Euclidean norm of a vector, and element-wise division. The parameter `p` of the `domeigen` function defines the number of iterations the method should perform.

Note that like all other listings in this chapter, Listing 20 is not pseudo code. It is pretty printed Julia source code. The ability to write such code, using a Unicode character set, allows engineers to produce very readable source code, at the mathematical level of abstraction at which they prefer to reason and to express their ideas.

The raison d'être of this example is to demonstrate an imperative application that only uses simple, standard array operations, i.e., limited to those defined in the base language libraries, and that does not require additional external functionality.

[2] https://en.wikipedia.org/wiki/Power_iteration
[3] https://en.wikipedia.org/wiki/Rayleigh_quotient

```
using LinearAlgebra
using Random

function domeigen(A, p)
  b₀ = similar(A, size(A, 1))
  rand!(b₀)

  # power iteration
  bₖ = b₀
  for _ in 1:p
    bₖ₊₁ = A * bₖ

    # normalize
    bₖ = bₖ₊₁ / norm(bₖ₊₁)
  end

  # Rayleigh quotient
  λ = (A*bₖ ⋅ bₖ) / (bₖ ⋅ bₖ)

  return bₖ, λ
end
```

Listing 20: Power method implementation approximating the dominant eigenvector and eigenvalue of a matrix.

4.1.2 Proximal Gradient Descent

Listing 21 implements a more complex example that combines array operations with a generically typed external library that extends the base language. The array operations now also include higher-order abstractions that compose with arbitrary user code.

Specifically, the example implements proximal gradient descent to minimize the squared error loss of a linear regression model. The example uses the ForwardDiff.jl package to determine the gradient and derivative of the loss function as defined by the user [127]. This package implements forward-mode automatic differentiation in Julia. Under the hood, it specializes user code to generate efficient machine code for computing derivatives. The ability to differentiate arbitrary user code distinguishes this Julia package from other automatic differentiation libraries. Many existing ML (Machine Learning) frameworks either require engineers to pick functions from a fixed library of functions for which gradients have been defined, while others can compute custom derivatives but only if the original function had been specified as a computational graph. By enabling us to differentiate arbitrary imperative

```
using ForwardDiff: gradient, derivative
using LinearAlgebra

# model
linear_regression(w, b, x) = w*x .+ b

# loss function
abs2(x) = abs(x^2)
mean_squared_error(ŷ, y) = sum(abs2, ŷ .- y) / size(y,2)

# get gradient w.r.t. to 'w'
loss∇w(model, loss, w, b, x, y) =
  gradient(w -> loss(model(w, b, x), y), w)

# get derivative w.r.t. to 'b'
loss∂b(model, loss, w, b, x, y) =
  derivative(b -> loss(model(w, b, x), y), b)

# optimization algorithm
function proximal_gradient_descent(model, loss, w, b, x, y; lr=.1)
  w -= lmul!(lr, loss∇w(model, loss, w, b, x, y))
  b -= lr * loss∂b(model, loss, w, b, x, y)
  return w, b
end

function main()
  # inputs and outputs
  x = ...
  y = ...

  # initial weights and bias
  w = ...
  b = ...

  model = linear_regression
  loss = mean_squared_error
  optimize = proximal_gradient_descent

  while current_loss > ...
    w, b = optimize(model, loss, w, b, x, y)
    current_loss = loss(model(w, b, x), y)
  end
end
```

Listing 21: Implementation of an ML model using proximal gradient descent method to minimize a squared error loss function.

code, ForwardDiff.jl improves productivity as well as flexibility of ML frameworks built on top of this package.

The `proximal_gradient_descent` function takes parameters that are common to many ML algorithms: `w` and `b` for respectively the vector of weights and the bias, while `x` and `y` represent the inputs and outputs that should be learned. The learning rate parameter `lr` is optional and defaults to 0.1. The function is to be called iteratively, and the weights and bias are updated in every iteration until the loss falls below an acceptable threshold.

Note that both the model and loss functions, of which lines 5 and 9 show examples, are defined independently from the optimization algorithm in `proximal_gradient_descent`. The model and loss functions are passed to the optimization algorithm as arguments, and they are simply passed on to anonymous functions that are themselves fed to the `gradient` and `derivative` functions from the ForwardDiff.jl library on line 12. This generalizes the implementation and makes it possible for the developer to iterate independently on each aspect of the implementation (loss, model, and optimization algorithm).

From the compiler's perspective, the `gradient` and `derivative` functions on lines 12 and 16 return dynamically-generated code. The Julia run-time compiler then generates specialized and statically optimized machine code. The design of the Julia language and its compiler, described in detail in Section 2.3, makes it possible to deliver good performance and enable code generation for accelerators, such as GPUs, that require static code.

The simple code of Listing 21 performs various operations on arrays much like those in Listing 20, but it also uses abstractions that compose with user code. For example, the loss function on line 9 calls the standard library operation `sum` with the user-defined function `abs2`, which is applied to all elements before they are summed. We will later discuss how this makes it possible to separate the concerns of application code from how the underlying abstractions are implemented.

The demonstrated composability with an external library, together with the portability to heterogeneous computing devices, greatly improves the ability to reuse code.

4.1.3 Kronecker Product

Finally, we describe a scenario where a more advanced user prototypes an algorithm by means of declarative code instead of an imperative subprogram. Specifically, Listing 22 implements the Kronecker product of

```
struct Kronecker{T,N,AT} <: AbstractArray{T,N}
  A::AT
  B::AT
  function Kronecker(A::AT, B::AT) where
                    {T, N, AT<:AbstractArray{T,N}}
    new{T,N,AT}(A, B)
  end
end

Base.size(K::Kronecker) = size(K.A) .* size(K.B)

function Base.getindex(K::Kronecker, i::Int, j::Int)
  I,Ix = divrem(i-1, size(B,1))
  J,Jx = divrem(j-1, size(B,2))
  K.A[I+1,J+1] * K.B[Ix+1,Jx+1]
end
```

Listing 22: Declarative implementation of the Kronecker product of two matrices.

two matrices, $\mathbf{A} \otimes \mathbf{B}$, where every element of the first matrix is multiplied with every element of the second matrix:

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} A_{11}\mathbf{B} & \dots & A_{1n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ A_{m1}\mathbf{B} & \dots & A_{mn}\mathbf{B} \end{bmatrix}$$

$$= \begin{bmatrix} A_{11}B_{11} & \dots & A_{11}B_{1q} & \dots & \dots & A_{1n}B_{11} & \dots & A_{1n}B_{1q} \\ \vdots & \ddots & \vdots & & & \vdots & \ddots & \vdots \\ A_{11}B_{p1} & \dots & A_{11}B_{pq} & \dots & \dots & A_{1n}B_{p1} & \dots & A_{1n}B_{pq} \\ \vdots & & \vdots & \ddots & & \vdots & & \vdots \\ \vdots & & \vdots & & \ddots & \vdots & & \vdots \\ A_{m1}B_{11} & \dots & A_{m1}B_{1q} & \dots & \dots & A_{mn}B_{11} & \dots & A_{mn}B_{1q} \\ \vdots & \ddots & \vdots & & & \vdots & \ddots & \vdots \\ A_{m1}B_{p1} & \dots & A_{m1}B_{pq} & \dots & \dots & A_{mn}B_{p1} & \dots & A_{mn}B_{pq} \end{bmatrix}$$

Instead of defining an imperative function that constructs an output matrix and eagerly computes the value for every element, we define a `Kronecker` type that lazily computes individual values when requested. This is called a structured matrix [71]. It is a common pattern in the Julia programming language, which provides many such arrays as part of the standard library. This example will demonstrate how our approach is composable with such infrastructure from the standard library.

```
function LinearAlgebra.norm(K::Kronecker, p::Real=2)
  A = norm(K.A, p)
  B = norm(K.B, p)
  return A * B
end
```

Listing 23: Optimized computation of the matrix norm for Kronecker products.

Just like any other array type, `Kronecker` is a subtype of `AbstractArray`, which mandates certain method definitions. One of those methods is the `getindex` method, which is used to get the value of an array that corresponds with a certain index. Whereas this method typically loads from memory, we implement it for the `Kronecker` type to compute a single value according to the definition of the Kronecker product.

Expressing computation declaratively using lazy arrays has several advantages: first and foremost, it saves on memory usage and avoids unnecessary computations. Furthermore, we can provide optimized implementations of certain methods by using problem-specific knowledge. For example, in the case of the Kronecker product we know from [88] that for matrices $\mathbf{A}$ and $\mathbf{B}$ the norm can be computed as:

$$||\mathbf{A} \otimes \mathbf{B}|| = ||\mathbf{A}|| \ ||\mathbf{B}||$$

We use this property of the Kronecker product to implement an optimized version of the `norm` function in Listing 23. This optimization greatly improves performance, as it prevents materialization of the Kronecker wrapper and as a result reduces the size of the matrices that need to be processed.

The approach from Listing 22 also composes with other lazy wrappers. For example, the Julia standard library avoids materializing matrix transpositions by using a `Transpose` wrapper that implements the expected indexing semantics. This wrapper type is also part of the `AbstractArray` hierarchy. It can hence be used as an input to our `Kronecker` type without materializing the wrapper. These and other opportunities for composability will be discussed in Chapter 5.

4.2 Related Work

In this chapter we focus on array abstractions and linear algebra, since that is the programming model most commonly used in the prototyping stage of engineering applications. Indeed MATLAB, NumPy [148] and a host of other languages that lend themselves more or less naturally to technical computing use the same programming model. High-level dynamic languages often use this model not only for its expressibility, but because they can implement the functionality as libraries in a low-level programming language and thereby gain performance [35, 129].

Array programming is also popular for working with heterogeneous devices. For example, in C and C++ there is CUB [104], Thrust [70], ArrayFire [101], OpenMP [49], OpenACC [154] and several others. These libraries achieve excellent performance, but require expertise in a low-level language and as such require a higher investment in time and effort to become proficient. Examples for higher-level languages that focus on programmer productivity include Anaconda Accelerate, GPU-accelerated numerical libraries for Python [47]; Parakeet, a runtime accelerator for an array-oriented subset of Python [130]; accelerate, a language for accelerated array processing in Haskell [38]; Copperhead, another functional data-parallel subset of Python [37], etc.

Frameworks for array programming in high-level languages are typically built on top of libraries that are implemented in a low-level language, either for performance, to be able to use the low-level vendor toolkits for working with hardware accelerators, or both. This split between the programming language that main application developers write in and the programming language that is used to implement the libraries, is an instance of the two-language problem and causes composability and extensibility problems [31, 100]. Once developers exhaust the functionality of the library and require custom functionality, e.g., because they want to take advantage of problem-specific knowledge as shown in the Kronecker example from Section 4.1.3, the library approach starts to break down and they have to resort to writing their code in the low-level language. Numba is a rare exception since it allows heterogenous programming in the same language, but it still struggles with composability and allowing for user-defined array abstractions that encode problem-specific knowledge [87].

The Julia programming language does not suffer from this problem, as the language has been co-designed with a JIT-compiler that generates high-quality machine code. The performance of scalar, loop-based programs is typically on par with implementations in a low-level language like C. As a result, the array operations themselves are also implemented

in Julia, and do not require a low-level language to achieve high performance [28]. This makes it easier to contribute to the implementations of array operations, be it as part of the Julia standard library or any of its packages. This feature is also found in the high-level Impala language part of the AnyDSL programming system, at the cost of requiring a DSL abstraction level to implement operations in and losing the ability to easily implement low-level operations [92].

The availability of a JIT compiler also enables powerful, higher-order abstractions that compose with arbitrary user code and separate the intent of the developer from the actual execution. The idea of separating the algorithm (what to compute) from the schedule (how and where to compute) is not new, and a prominent feature of Halide which uses a C++ DSL to allow programmers to write pipelines (image algorithms) independently of the schedule and execution target [121, 94]. Halide allows for automatic scheduling of pipelines, but most advanced users will want to specify their own, since a programmer with deep knowledge of the hardware can create an optimal schedule of the pipeline. The Halide approach is declarative and focuses on stencils, which is unfamiliar to a developer used to high-level languages and their use of array abstractions.

4.3 Background: Array Programming in Julia

Julia's array abstractions focus on ease of use, expressiveness through higher-order abstractions, and good performance by means of compiler specialization [28]. This approach facilitates prototyping, but also makes it possible to reuse application code outside of the prototyping phase. Applications work with generically-typed arrays without any performance penalty. In Chapter 5 we will illustrate how that enables portability across array types, and consequently across execution environments, offering flexibility that is similar to Halide.

Altogether, even though use of array abstractions is not a necessity in Julia, they are commonly used due to the natural, concise representation while being very versatile and capable of expressing a wide range of computations. At the same time, the abstractions expose a great deal of parallelism, and are therefore ideal candidates for parallel programming. This will be discussed in Section 4.4.

```
a = [1 2; 3 4]
b = [3 4; 5 6]
c = [5 6; 7 8]

map(x->x+1,         a)
map(+,              a, b)
map((x,y,z)->x+y+z, a, b, c)

broadcast(+,        a, 1)
broadcast(+,        a, [-1; 1])
```

Listing 24: Example use of the `map`, `broadcast` and `reduce` abstractions.

4.3.1 Higher-Order Array Abstractions

We discuss two higher-order abstractions that are commonly used with Julia arrays: `map` and `broadcast`. These are just two of many higher-order abstractions that exist in the Julia standard library, including `reduce`, `accumulate`, `foldr` and `foldl`, `mapreduce`, etc. Each of these abstractions compose with user code that determines *what* is computed, while the methods that implement these abstractions determine *how* and *where* that computation will happen. The implementations can be further specialized on the type of the arguments, selecting an implementation that maximizes performance or otherwise preserves the array type, e.g., to prevent memory transfers from or to a heterogeneous computing device.

`map` Abstraction

At its core, `map` transforms collections of identical shape and size by applying a function elementwise over the collections, as shown in Listing 24. The function should accept as many arguments as the amount of containers passed to `map`.

The abstraction is a prime example of a higher-order abstraction. The first argument to `map` is a function that defines the transformation. It can be any type of function, including including user-defined and anonymous functions. The JIT compiler specializes the implementation of `map`, which only deals with the semantics of the abstraction, with the transformation function as specified by the user.

Furthermore, the underlying storage is implemented by a separate container type. In the example from Listing 24 this is the standard `Array` type, which is itself specialized on the standard element type `Int`. However, it is as easy to use nonstandard types for containers and elements.

This is a clear separation of concerns, facilitating reuse by limiting the responsibility of each aspect of the overall computation.

`broadcast` Abstraction

The `broadcast` abstraction generalizes the behavior of `map` to containers of heterogeneous shapes by extruding dimensions accordingly. This greatly improves use of the abstraction with objects of different shapes and sizes. Broadcasted operations generally take the form $b : \mathbb{R}^N \to \mathbb{R}^M$ where N is the input arity and M is the output arity. We define the broadcast of this operation as:

$$\texttt{broadcast}(b, \mathbf{X}_1 \dots \mathbf{X}_N) = \texttt{map}(b, \boldsymbol{\mathcal{X}}_1 \dots \boldsymbol{\mathcal{X}}_N) = \mathbf{Y}_1 \dots \mathbf{Y}_M$$

Here, arguments $\mathbf{X}_j$ are multidimensional arrays of arbitrary shape,[4] subject to the constraint that each dimension of any argument must either have the same length as that dimension in other arguments, or must have length 1. Each $\boldsymbol{\mathcal{X}}_j$ is equivalent to the corresponding $\mathbf{X}_j$, but where length-1 dimensions are "copied" along that dimension to match its maximum length across all $\mathbf{X}_j$, such that all $\boldsymbol{\mathcal{X}}_j$ are of equal shape. The function b is then mapped elementwise across all $\boldsymbol{\mathcal{X}}_j$, resulting in the outputs $\mathbf{Y}_i$, each of which is the same shape as any $\boldsymbol{\mathcal{X}}_j$.[5]

For example, broadcasting $b : \mathbb{R}^3 \to \mathbb{R}^2$ over an $n \times m$ matrix $\mathbf{A}$, a scalar α, and an n-element vector $\mathbf{a}$ yields:

$$b.(\mathbf{A}, \alpha, \mathbf{a}) = \left(\begin{bmatrix} b(A_{11}, \alpha, a_1)_1 & \dots & b(A_{1m}, \alpha, a_1)_1 \\ \vdots & \ddots & \vdots \\ b(A_{n1}, \alpha, a_n)_1 & \dots & b(A_{nm}, \alpha, a_n)_1 \end{bmatrix}, \begin{bmatrix} b(A_{11}, \alpha, a_1)_2 & \dots & b(A_{1m}, \alpha, a_1)_2 \\ \vdots & \ddots & \vdots \\ b(A_{n1}, \alpha, a_n)_2 & \dots & b(A_{nm}, \alpha, a_n)_2 \end{bmatrix} \right) \tag{4.1}$$

As of Julia 1.0, broadcast is represented with a first-class data structure [9]. Broadcast expressions are lowered to instances of the `Broadcasted` type, which represents a tree of a broadcast operations. These objects

[4] Note that scalars and single-element arrays are equivalent under this definition of `broadcast`.

[5] Advanced broadcast implementations index directly into the $\mathbf{X}_j$ arguments to perform b elementwise invocations (as is done in Equation 4.1) rather than explicitly materialize the $\boldsymbol{\mathcal{X}}_j$ arguments.

Source code	Lowered to
$f.(\mathbf{a})$	`broadcast`$(f, \mathbf{a})$
$\mathbf{b} .= f.(\mathbf{a})$	`broadcast!`$(f, \mathbf{b}, \mathbf{a})$
$f.(\mathbf{a} .+ \mathbf{b}) .* \mathbf{c}$	`broadcast`$(\langle a, b, c\rangle \to f(a+b) * c, \mathbf{a}, \mathbf{b}, \mathbf{c})$

Table 4.1: Lowering of different forms of broadcast syntax. The last example illustrates fusion of elementwise operations.

are accessible to implementations of broadcast, and can be used to customize how broadcast is computed depending on the arguments and output types. For example, it allows for broadcast expressions on ranges to be calculated eagerly, for custom array types to opt-out of broadcast fusion and evaluate each operation individually, and for splitting broadcast expressions into chunks that can be computed in parallel.

4.3.2 Dot Expressions

To improve the usability of `broadcast`, so-called *dot expressions* can be used in Julia to denote elementwise transformations [76]. The Julia parser lowers this syntactic sugar to invocations of the `broadcast` function, as illustrated with some examples in Table 4.1. Elementwise assignments call the `broadcast!` function, which performs in-place assignment to avoid allocating an output container.

4.3.3 Broadcast Fusion

Assuming that a pair of broadcasted operations have compatible shapes and are side-effect free, the broadcast of their composition generally obeys the following relation:

$$g.(f.(\mathbf{X}_1 \ldots \mathbf{X}_N)) = (g \circ f).(\mathbf{X}_1 \ldots \mathbf{X}_N) \tag{4.2}$$

In programs containing broadcast operations, Equation 4.2 can be exploited to perform *broadcast fusion*, a compiler-level optimization that transforms compositions of broadcast calls into a single broadcast call. This transformation is performed at the AST level, and further improves the appeal of the broadcast abstraction [76].

Broadcast fusion imparts a couple of performance benefits. First, by obviating the need to compute and store intermediate results, broadcast fusion reduces memory usage, temporary allocations, and kernel invocations required to complete the computation. Second, broadcast fusion allows the fused broadcast operation to be parallelized without

re-synchronization between intermediary broadcast operations [50, 83]. This property is useful for the purpose of a GPU implementation, and can also be exploited to perform efficient automatic differentiation of fused kernels as will be explained in Chapter 6.

4.4 Heterogeneous Programming with Arrays

Building on the array abstractions from the previous section, we provide a framework for programming heterogeneous devices where it is possible for application code to only deal with *what* needs to be computed, while an underlying array type takes care of *where* the data is stored, and *how* the computations are performed. At the same time, use of higher-order functions make it possible to improve the abstractions' flexibility and compose with arbitrary user code. Before explaining how we implemented these abstractions for CUDA GPUs, this section will describe the necessary techniques with an illustrative `HeterogeneousArray` type. It aims to demonstrate how the array abstraction level is a good fit for targeting accelerator hardware, while keeping the library implementation generic enough to work with different kinds of applications.

4.4.1 Array Type Hierarchy

At its core, every array type starts with a parametric type definition that subtypes the `AbstractArray` type. In the case of an array type that is backed by actual device memory, as opposed to a so-called *view* that only changes the behavior from an underlying array, the type would contain a number of fields that provide handles to device memory. In Listing 25, we define such a `HeterogeneousArray` type that contains a single field, `handle`, to store a pointer to device memory. The constructor on line 9 accepts any array data as input, and uploads it to the device by using an the `to_device` function that is provided by the illustrative device back-end package DeviceBackend.jl. A counterpart function on line 13 implements conversion back to a CPU array, downloading from device memory using the `from_device` function.

The `AbstractArray` type also contains two type parameters, `T` and `N`, for respectively the type and dimensionality of the array. These type parameters need to be filled in for any concrete instantiation of an array, and can be used to dispatch to optimized method implementations that depend on the value of these type parameters. Examples are an optimized matrix-vector multiplication, or an implementation that calls a C library that only provides implementations for C data types. In the

```
1  using DeviceBacked: to_device, from_device
2
3  struct HeterogeneousArray{T,N} <: AbstractArray{T,N}
4    # member field storing handle to device memory
5    handle::Ptr{T}
6
7    # constructor
8    HeterogeneousArray(data::AbstractArray{T,N}) where {T,N} =
9      new{T,N}(to_device(data))
10 end
11
12 Base.convert(::Type{Array}, array::HeterogeneousArray) =
13   from_device(array.handle)
```

Listing 25: Storage handling for a heterogeneous array type.

```
1  using DeviceBacked: to_device, from_device
2
3  function Base.setindex!(array::HeterogeneousArray, i::Integer, value)
4    to_device(array.handle, i, value)
5    return
6  end
7
8  function Base.getindex(array::HeterogeneousArray, i::Integer)
9    return from_device(array.handle, i)
10 end
```

Listing 26: Scalar indexing for a heterogeneous array type.

case of `HeterogeneousArray`, the actual values of these type parameters are deduced by the constructor from the input data.

4.4.2 `AbstractArray` Interface

As part of the `AbstractArray` interface, custom array types should[6] implement certain functionality, such as the `getindex` and `setindex!` methods to fetch and to store scalar elements from the array. Examples of these methods are defined in lines 3 and 8 of Listing 26 where we rely on versions of the `from_device` and `to_device` functions of the device back-end package to load from and store to device memory.

[6]Interfaces are currently not enforced by the compiler. Typically, the relevant documentation provides a list of methods that must or may be implemented, and what the semantics of these methods should be.

```
using DeviceBacked: @on_device

function LinearAlgebra.mul!(Y::HeterogeneousArray{T, N},
                            A::HeterogeneousArray{T, N},
                            B::HeterogeneousArray{T, N}) where {T, N}
  @on_device begin
      x = A[...] * B[...]
      Y[...] = y
  end
end
```

Listing 27: In-place multiplication for a heterogeneous array type.

These scalar access methods are useful because they provide compatibility of the array type with existing code that explicitly iterates over the elements of arrays. For example, the "default" definition of matrix multiplication for `AbstractArray`s in the Julia standard library, which is designed for execution on a host CPU, uses the textbook algorithm with nested for loops that multiply and accumulate matrix elements. When that matrix multiplication is invoked on an array of type `HeterogeneousArray`, it still computes the correct result, albeit it very slowly: The nested loops is still executed on the host CPU, and every element accessed in the array on the device is transferred individually from the device to the host. This obviously is very slow, and defeats the entire purpose of accelerator hardware. Still, it provides compatibility with existing scalar code. Such code can then be incrementally ported to use array abstractions, and the results can be verified at every step.

For an array type to be usable for engineering purposes, it has to provide efficient versions of relevant array abstractions. As detailed in Chapter 2, the design of the Julia programming language facilitates such overloads. In Listing 27 we demonstrate how a custom array type can implement a generic matrix-matrix multiplication that replaces the aforementioned generic, scalar version of the standard library. The example uses the `@on_device` macro provided by the DeviceBackend.jl package to mark code that should be executed on the device. Note that the implementation is still fully generic. It can be used with any element type (e.g., plain scalars, or complex structures such as a `Dual` number type from the DualNumbers.jl package) as long as multiplication and addition are defined for the type. When this method is invoked, the run-time compiler specializes the code on the actual run-time arguments, i.e., concrete instances of `HeterogeneousArray` with values for the `T` and `N` type parameters, and on the execution context, i.e., `@on_device`.

```
1 using DeviceBacked: @on_device
2
3 function Base.copyto!(dest::HeterogeneousArray, op::Broadcasted)
4   @on_device begin
5     I = CartesianIndex(dest)
6     dest[I] = op[I]
7   end
8 end
```

Listing 28: Implementation of the broadcast interface for a heterogeneous array type.

The illustrative operations on lines 7 and 8 are syntactic sugar underneath of which implementations of `getindex` and `setindex!` are used. The examples from Listing 26 implemented these methods for the purpose of accessing elements of the heterogeneous array from the host. But in this context, these operations are executing on the device already. This obviates the transfers of the elements to and from the host processor. In the `@on_device` context, the `from_device` and `to_device` should be specialized to perform direct accesses of device memory, and the computations should be performed directly on the device.

There are several possibilities to specialize methods on their execution context. With Cassette.jl, contextual dispatch could be used to extend Julia's method dispatch and take the execution context into account [124]. CUDAnative.jl takes a simpler, type-based approach by converting objects such as arrays to a device equivalent when passing from the host to the device, and implementing separate `getindex` and `setindex!` methods for these device arrays. Alternatively, a type parameter that encodes the execution context could be added to the `HeterogeneousArray` type to similarly dispatch to different methods based on the value of this parameter.

4.4.3 `broadcast` Abstraction

Abstractions such as matrix multiplication from Listing 27 as implemented for a heterogeneous array type define both *what* is executed, *where*, and *how*. By contrast, higher-order abstractions like `broadcast` from Section 4.3.1 make the user responsible for specifying only *what* is computed. Such a separation of concerns will facilitate greater reuse of code, as Chapter 5 will discuss.

To support broadcast operations, Listing 28 provides an implementation of the `copyto!` function for `HeterogeneousArray` when it is passed a `Broadcasted` tree. This method is responsible for executing a flattened representation of broadcast expressions in the context of a certain array type, and is part of the interface that makes up the broadcast interface. Both the broadcast expression and destination array are indexed with a `CartesianIndex`, a tuple of integers that represents a multidimensional index into a container. In the case of our `HeterogeneousArray` type, we make sure this operation happens on the device by using the `@on_device` macro. Other required definitions concern the so-called broadcast style, for broadcasting between objects of different types and determine how to allocate memory for out-of-place broadcasting.

The full extent of the broadcast interface is shown in Table 4.2. It shows that this interface is fairly lightweight, only requiring two method definitions to make objects broadcastable. By doing so, generic broadcast functionality that relies on scalar iteration is reused. The broadcast interface exposes appropriate hooks to further customize this functionality, e.g., to avoid scalar iteration with heterogeneous devices by providing an implementation of `copyto!` as in Listing 28. These definitions do not need to be specific to the concrete object that is being broadcasted, but can be implemented for any supertype in order to share functionality and reduce the required number of definitions.

4.5 CuArrays.jl

The *CuArrays.jl* package[7] defines a `CuArray` type that provides an array programming abstraction for CUDA GPUs [78]. It implements common array operations, including Julia's higher-order abstractions from Section 4.3, a memory manager to amortize the cost of allocating memory and track their uses, and various high-level utilities that improve the end-user programming experience.

4.5.1 Array Operations

CuArrays.jl provides implementations of many common array operations for NVIDIA GPUs. Where possible, these implementations call out to existing, vendor-provided libraries such as cuBLAS or cuDNN. These libraries are mature and optimized for each hardware generation. Other operations, such as the higher-order abstractions from Section 4.3 are implemented using the CUDAnative.jl GPU compiler from Chapter 3.

[7] Available at https://github.com/JuliaGPU/CuArrays.jl

Function signature	Description
`BroadcastStyle(object::Type)`	Broadcasting behavior.
`similar(::Broadcasted, elem::Type)`	Allocation of output containers.
Optional methods	
`BroadcastStyle(::Style1, ::Style2)`	Precedence rules for styles.
`axes(x)`	Declaration of the indices of `x`.
`broadcastable(x)`	Convert to an object that has axes and supports indexing.
Bypassing default machinery	
`copy(::Broadcasted)`	Custom out-of-place `broadcast`.
`copyto!(dest, ::Broadcasted)`	Custom in-place `broadcast!`.
`broadcasted(f, args...)`	Override the default behavior within a fused expression.
`instantiate(::Broadcasted)`	Override the computation of the lazy broadcast's axes.

Table 4.2: Methods that make up the public broadcast interface and can be implemented to support or customize the behavior of broadcast operations.

Availability of a GPU compiler like CUDAnative.jl makes it possible to extend the applicability of many of these operations. For example, matrix multiplication as implemented by cuBLAS only supports certain real and complex element types, and is limited to specific dense memory layouts. CuArrays.jl provides a fallback implementation[8] of matrix multiplication, shown in Listing 29. Since this implementation is generically typed, it is applicable to all element types that define multiplication and addition, and supports every memory layout with well-defined indexing semantics. This greatly improve usability of the array type in regard to multiplying matrices, while still dispatching to the high-performance cuBLAS implementations whenever possible.

As a realistic example, consider automatic differentiaton with ForwardDiff.jl [127]. This package makes it possible to differentiate arbitrary Julia code by replacing scalar inputs with dual numbers. As illustrated in Listing 3, these numbers encapsulate both the original scalar input as well as a number of epsilon components. In the case of automatic differentiation, these epsilon components represent the partial derivatives with respect to that input, and operations on these dual numbers will apply the original operation to the scalar component as well as the derivative of the operation to each of the epsilon components.

If the code under differentiation contains a matrix multiplication, or any other operation that would have dispatched to a vendor-provided library, it is not possible to reuse that implementation because it most likely does not support the dual number arguments as represented by instances of the `Dual` type from ForwardDiff.jl. To support these and other uses cases[9] where the element type is not supported by the vendor-provided libraries, we provide generic implementations like the matrix-matrix multiplication in Listing 29 that support any input type by relying on specialization and JIT compilation.

[8]The implementation is naive, and does not come close to the performance of cuBLAS. However, it is only intended to be used with element types that are not supported by cuBLAS. These types are typically complex or large (e.g., `Dual` numbers, large or high-precision scalars, user-defined structures, …), resulting in performance that is often bound by bandwidth or by the cost of the scalar operations.

[9]Other examples include the use of high-accuracy types such as DoubleFloats.jl [136], intervals for bounded computations with IntervalArithmetic.jl [134], or simple scalar types that are not covered by the fairly limited set of real number types that are typically supported by the NVIDIA vendor libraries.

```
function generic_matmatmul!(C::AbstractVecOrMat{R},
                            A::AbstractVecOrMat{T},
                            B::AbstractVecOrMat{S}) where {T,S,R}
    @assert size(A,2) != size(B,1)
    @assert size(C,1) != size(A,1) || size(C,2) != size(B,2)

    function kernel(C, A, B)
        i = (blockIdx().x-1) * blockDim().x + threadIdx().x
        j = (blockIdx().y-1) * blockDim().y + threadIdx().y

        if i <= size(A,1) && j <= size(B,2)
            z2 = zero(A[i, 1]*B[1, j] + A[i, 1]*B[1, j])
            Ctmp = convert(promote_type(R, typeof(z2)), z2)
            for k in 1:size(A,2)
                Ctmp += A[i, k]*B[k, j]
            end
            C[i,j] = Ctmp
        end

        return
    end

    numthreads, numblocks = ...  # heuristic to maximize occupancy
    @cuda threads=numthreads blocks=numblocks kernel(C, A, B)

    return C
end
```

Listing 29: Generic implementation of matrix-matrix multiplication from CuArrays.jl.

4.5.2 Higher-Order Abstractions

The GPU compiler from Chapter 3 is essential to implement the higher-order array abstractions from Section 4.3.1. These abstractions are especially important for GPU computing, where naive or non-expert implementations often do not perform well. For example, `reduce` on the CPU can be implemented reasonably efficiently with a plain `for` loop that accumulates values. To get any performance on the GPU, we need a parallel reduction that effectively uses the memory hierarchy. We provide such an implementation in CuArrays.jl, using a tree-based reduction based on shared memory and shuffle instructions [66, 98].

We further illustrate this point using a simple vector addition. Listing 30 shows how to compute an element-wise addition of two `CuArray` GPU arrays using CUDAnative.jl. At this abstraction level, users need to provide a scalar kernel function that is executed according to the SPMD (Single Program, Multiple Data) programming model. Indexing semantics need to match the data as well as the limitations of the hardware, while taking into account the occupancy that results from the requested launch configuration in combination with the hardware resource usage of the kernel (e.g., using the tools from Section 3.6.3). In contrast, Listing 31 performs this operation using array operations from CuArrays.jl. Specifically, we use dot syntax to broadcast the `+` function across the input arrays. This completely avoids the need to provide a SPMD kernel. The example demonstrates how users can use the `CuArray` type with powerful, higher-order abstractions that obviate manual kernel programming. However, when flexibility is required, it is still perfectly possible to go deeper and use CUDAnative.jl to create custom SPMD kernels as with Listing 30. Both approaches can perfectly coexist in a single application.

Under the hood, the implementation of `broadcast` for `CuArray` transforms the scalar transformation to a valid SPMD kernel. Listing 32 shows a part of that implementation from the CuArrays.jl package.[10] As explained in Section 4.4.3, the `copyto!` method is responsible for executing a broadcast expression in the context of a specific array type, here `CuArray`. The implementation defines an anonymous kernel on line 6, which calculates array indices using GPU intrinsics in accordance with the dimension-matching semantics of the broadcasting abstraction. The kernel is subsequently executed in parallel on line 14 using CUDAnative.jl. This is similar to the low-level use of CUDAnative.jl as shown in Listing 30.

[10] Part of this implementation is from the GPUArrays.jl package, which contains vendor-neutral GPU implementations of common abstractions.

```
using CuArrays

a = CuArray(rand(2,2))
b = CuArray(rand(2,2))
c = similar(a)


using CUDAnative

function vadd(c::CuArray, a::CuArray, b::CuArray)
    i = (blockIdx().x-1) * blockDim().x + threadIdx().x
    c[i] = a[i] + b[i]
    return
end

numthreads, numblocks = ...  # heuristic to maximize occupancy
@cuda threads=numthreads blocks=numblocks vadd(c, a, b)
```

Listing 30: Low-level addition of GPU arrays using kernel programming interfaces from CUDAnative.jl.

```
using CuArrays

a = CuArray(rand(2,2))
b = CuArray(rand(2,2))
c = similar(a)

c .= a .+ b
```

Listing 31: High-level alternative to Listing 30, adding two GPU arrays using `broadcast` from CuArrays.jl.

```
using CUDAnative

function Base.copyto!(dest::CuArray, bc::Broadcasted)
  op = Broadcast.preprocess(op)

  function kernel(dest, op::Broadcasted)
    i = (blockIdx().x-1) * blockDim().x + threadIdx().x
    I = CartesianIndex(i)
    dest[I] = op[I]
    return
  end

  numthreads, numblocks = ...  # heuristic to maximize occupancy
  @cuda threads=numthreads blocks=numblocks kernel(dest, op)

  return dest
end
```

Listing 32: Low-level implementation of one of the methods that implement the broadcast abstraction, taken from CuArrays.jl.

Finally, high-level abstractions can also improve performance. As mentioned in Section 4.3.2, the Julia parser syntactically fuses multiple broadcast expressions together, resulting in fewer calls to `broadcast`. In the context of GPU programming, the advantages of broadcast fusion are profound: fewer kernel launches are required, memory allocations for temporary outputs can be avoided, and temporaries live in registers and do not have to be loaded from global memory.

4.5.3 Memory Management

The CuArrays.jl package also features a memory pool allocator to improve performance and usability of the underlying CUDA allocator. The CUDA API wrapper from Section 3.5 already enhanced the CUDA APIs by supporting garbage-collected memory and integration with the Julia GC (Garbage Collector). However, the Julia garbage collector is currently not extensible, and its heuristics do not consider the memory pressure on external hardware. As a result, GPU memory is not collected any faster when pressure is high, with frequent out-of-memory exceptions as a result.

CuArrays.jl provides a memory allocator that layers on top of the existing allocation routines in CUDAdrv.jl. When allocating new memory, we catch out-of-memory exceptions and trigger a sweep of the Julia garbage collector. This is expected to release unused buffers. Furthermore, allocations are taken from and released back into pools of equally sized buffers. This makes it possible to quickly re-allocate without the need to call into the slow CUDA allocator. These pools can be purged at any time, e.g., when an allocation is required that cannot be fulfilled by the device and cannot be allocated from the designated pool either.

The allocator has been tuned on memory-intensive ML models from the Flux.jl library. Compared to the reference allocator where all stale objects are aggressively collected and released with every out-of-memory situation, a convolutional image classifier for the MNIST database of handwritten digits performs 60% fewer actual allocations (56% fewer allocated bytes) while spending significantly fewer cycles on GC operations (1.22% instead of 4.23% of total application run time).

The design of this pooling memory allocator is similar to, e.g., PyTorch's `THCCachingAllocator` or TensorFlow's `PoolAllocator`. One improvement for the purpose of interactive use, a common scenario with Julia, is to keep track of each pool's usage history. When pools are underused, i.e., when it contains more buffers then are concurrently in use at any point in recent time, the pool is shrunk to reduce total device memory usage. This improves collaborative use of a single GPU with multiple interactive users as it prevents individual users from monopolizing the device's memory.

4.5.4 Low-level Flexibility

Although this chapter presents an array programming interface that is designed to raise the abstraction level, it still allows for low-level flexibility as enabled by the underlying GPU JIT compiler from Chapter 3. To illustrate this point, Listing 33 demonstrates how to compute the base-2 exponential of every element in an array, using array abstractions from Section 4.3.1. The first approach uses the `broadcast` abstraction via dot syntax, resulting in highly-readable code that is portable across array implementations. The second approach uses a PTX-specific instruction for efficient GPU execution, and demonstrates how the high-level `map` abstraction can be used together with the low-level `@asmcall` macro from LLVM.jl for inline assembly in Julia.

```
# allocate data
julia> using CuArrays
julia> A = CuArray{Float32}([1, 2, 3])
3-element CuArray{Float32,1}:
1.0  2.0  3.0

# first approach: broadcast with dot syntax
julia> 2f0 .^ A
3-element CuArray{Float32,1}:
2.0  4.0  8.0

# second approach: map with inline asembly
julia> map(a -> @asmcall("ex2.approx.f32 \$0, \$1;", "=f,f",
                         Float32, Tuple{Float32}, a),
           A)
3-element CuArray{Float32,1}:
2.0  4.0  8.0
```

Listing 33: Use of high-level array abstractions to compute the base-2 exponential of every element in an array.

The ability to use high-level abstractions with low-level code is especially useful in combination with Julia's multiple dispatch. As explained in Section 2.3, this kind of dispatch provides the flexibility to selectively override methods that would be out of reach with single dispatch semantics. A common use-case is to initially develop with only very high-level array operations, then determine where performance is lacking, and subsequently provide optimized method implementations regardless of where the original definition came from (e.g., in a third party library, or as part of the Julia standard library). Crucially, these methods need not to be incompatible with the original definition and its use cases. For example, the PTX-specific invocation of `map` in Listing 33 should be confined to a method that can only be dispatched to in the context of CUDA arrays, either by specifying that the input should be of type `CuArray`, or by using contextual dispatch as explained in Section 4.4.

Chapter 5
Array Programming for Portability

Array programming as introduced in Chapter 4 makes it possible for non-expert programmers to use GPUs and accelerate their applications. The approach has other advantages too: By decoupling the application from the underlying array implementation, we can improve portability and seamlessly execute applications on different platforms and computing environments.

This chapter will demonstrate the portability of array programming in Julia with the engineering applications from Chapter 4. We will evaluate these applications across several array types, including CuArrays.jl for GPU programming and DistributedArrays.jl for distributed execution, and assess their performance.

The DistributedArrays.jl package as described in this chapter is pre-existing software that was not developed as part of the work presented in this dissertation. Improvements to this and other packages for the purpose of GPU execution, as well as the experiments presented in this chapter, were developed by myself and Valentin Churavy of the Massachusetts Institute of Technology.

In this chapter, I contribute a scientific analysis of the portability that is enabled by array programming. I also demonstrate how careful implementation of the underlying infrastructure makes it possible to compose functionality, for the purpose of performance or productivity. These contributions have been published in a peer-reviewed journal.[1]

[1] Tim Besard, Valentin Churavy, Alan Edelman, and Bjorn De Sutter. "Rapid Software Prototyping for Heterogeneous and Distributed Platforms". In: *Advances in Engineering Software (AES)* (2019). DOI: 10.1016/j.advengsoft.2019.02.002.

5.1 Background and Related Work: Portable Distributed Computing

Distributed computing is a field of computer science that deals with distributed systems, running so-called distributed programs on groups of autonomous computational entities or nodes. The main reason to use such a system is for their greater computational power compared to individual nodes. These entities are typically connected over a network, and use some form of message passing to communicate with each other.

Programming distributed systems is typically done in a low-level language like C or C++, using interfaces like as MPI (Message Passing Interface) [106], Legion [8], and UPC++ [6]. More convenient interfaces like OpenMP [49] or Cilk [33] are incompatible due to their reliance on shared memory. There also exist special-purpose HPC (High-Performance Computing) languages such as IBM's X10 [40], Chapel [39], and Fortress [2], which were created with any number of good ideas but have failed to attract a substantial user base outside of the community that originally developed it. Other approaches make use of large and complicated libraries like Trilinos [69], PetSC [7], and Kokkos [36]. These have been developed to facilitate the reuse of common numerical infrastructure and have found a fervent following in the HPC community. They are known to achieve excellent performance in cluster environments, and are well suited for performance engineers comfortable with C/C++ and distributed programming. However, this also makes them unsuited for prototyping or other exploratory development.

Furthermore, the abstractions of these frameworks tend to break down when combining the distributed nature of systems with nonstandard computational entities, such as GPUs or other hardware accelerators. Effective use of MPI requires a specific implementation suited for use with CUDA [84], X10 requires users to map data structures to GPU memory explicitly [48], etc. Where other libraries, like Legion and UPC++, do compose together with a GPU software stack like CUDA, they require the programmer be proficient in both distributed programming practices, GPU development techniques, and the cross-over of both to optimize GPU memory transfers across individual nodes.

```
function Base.map!(f, dest::DArray, data)
  @sync for p in procs(out)
    @async remotecall_wait(p, f, dest, data) do f, dest, data
      local_dest = localpart(dest)
      map!(f, local_output, makelocal(data, localindices(dest)...))
    end
  end
end
```

Listing 34: Low-level implementation of in-place `map` taken from DistributedArrays.jl.

5.2 DistributedArrays.jl

The *DistributedArrays.jl* package[2] builds on Julia's distributed computing infrastructure to provide a Global Array-like interface [112, 80]. The package focuses on programmer productivity by using a high-level language and by automatically distributing data and operations without requiring any user control with respect to partitioning, data distribution, etc. It provides a `DArray` data structure that distributes an array across a set of processes, where each process holds a chunk of the total array. The memory is globally addressable, and RPCs (Remote Procedure Calls) are issued automatically when accessing memory that is not local to the process. This makes it possible to support scalar indexing for code compatibility reasons, while optimized implementations of operations are aware of the distribution of memory and can avoid communication overhead.

The type signature of `DArray` consists of three type parameters: `T` and `N` from the `AbstractArray` interface for respectively the element type and dimensionality, and `A` for the underlying local array type. The local array type parameter enables a great amount of flexibility, as it allows `DArray` to be mostly agnostic to the underlying array type. This again allows to separate concerns, where the `DArray` type manages communication while the underlying array `A` is responsible for the storage, computation, etc. Section 5.3.3 will show how this patterns makes it possible to compose array types that, like `DArray`, wrap other arrays.

Listing 34 is an example of an implementation of a high-level abstraction for distributed arrays in DistributedArrays.jl. It follows the owner-computes rule by which each processor performs computations on the data it owns. The example implements an in-place `map` through

[2] Available at https://github.com/JuliaParallel/DistributedArrays.jl

```
# prepare a parallel computing environment
using Distributed
addprocs(2)

using DistributedArrays

a = distribute(rand(2,2))
b = similar(a)

map!(sin, b, a)
```

Listing 35: High-level use of the `map!` abstraction with distributed arrays from DistributedArrays.jl.

a series of RPCs, predominantly operating on local memory and avoiding unnecessary communication to other processes. The master process orchestrates the communication between workers and the actual work is delegated to operations on local data. The example demonstrates the aforementioned separation of concerns: The code of Listing 34 only deals with distributing the `map` operation, and defers to the underlying array type for the actual implementation of the abstraction.

The example calls `remotecall_wait` from the Julia distributed infrastructure to invoke an anonymous function on process `p` to execute the `do ... end` block that follows. The worker process then accesses the `localpart` of the target array and localizes through `makelocal` those parts of the input `data` array that are required to compute the local part of the map. If necessary `makelocal` fetches and copies data from other workers, but if data is already locally available this copy is avoided. The call to `remotecall_wait` is a blocking RPC and is wrapped in an `@async` block, which starts a lightweight task. Tasks are used to prevent the processes, especially the master, from blocking on a call since otherwise no progress could be made and no other RPCs could be issued. Finally, the `@sync` block waits for all enclosed tasks to make sure the computation is finished when returning from the `map!` function.

The distributed computing abstractions as used in Listing 34 are defined in the Julia standard library. They use the `ClusterManager` interface for launching worker processes on distributed systems. The standard library implements this interface for local processes and for networked systems that expose the SSH (Secure Shell) protocol. External packages can be used to work with managed clusters, such as ClusterManagers.jl that implements a `ClusterManager` subtype for the Slurm workload manager [156], the Portable Batch System [67], and

others. For environments that rely on the MPI [106], `MPIManager` from MPI.jl can be used to communicate with processes over an optimized communication fabric such as InfiniBand [97]. The design of this infrastructure enables distributed code that works with distributed processes, such as DistributedArrays.jl, to be agnostic of the underlying processes and how they communicate.

The implementation as shown in Listing 34 is written by specialists that know how the DistributedArrays.jl package is structured, and how to execute code efficiently in a distributed setting. This complexity is completely hidden from the end user: Listing 35 shows how to use the `map!` abstraction from Listing 34 on a newly allocated `DArray`. This does not differ from use of the abstraction with any other array type. The only code specific to distributed computing deals with launching local processes by calling `addprocs` on line 3.

5.3 Evaluation

This section discusses how the examples from Section 4.1 and other codes can be ported to other platforms and environments by using different array types. We use the `Array` type as provided by the Julia standard library, `CuArray` from Section 4.5 and the `DistributedArray` type described in Section 5.2.

Section 5.3.1 discusses the portability of standalone applications with respect to different array implementations for different heterogeneous platforms. Section 5.3.2 focuses on libraries that provide domain-specific functionality using array abstractions, for use in standalone applications and/or in compositions with other libraries. Such libraries should be generic with respect to array types not to hinder the portability of the applications or other domain libraries in which and with which they are used. Finally, Section 5.3.3 analyzes the portability and composability of libraries that define new array types and/or extend existing array abstractions.

5.3.1 Application Portability

Array-based application code that does not rely on library functionality, such as the example from Listing 20, can be ported trivially. It suffices to use an appropriate array type by changing the array allocations to use a different constructor, for example, `CuArray(...)` instead of `Array(...)`. Operations on these arrays then dispatch to respective implementations in the corresponding array package. If that package does not provide certain operations, fallback methods from the Julia

standard library are used. For example, when passing a `CuArray` to the `domeigen` function from Listing 20, the call to `rand!` dispatches to an optimized implementation in CuArrays.jl that uses the cuRAND library. Similarly, the multiplication on line 11 is lowered to a call to `mul!`. Several implementations of `mul!` are provided in CuArrays.jl, using the cuBLAS library when possible but falling-back to a generic matrix-matrix multiplication when required for, e.g., element types that are not supported by cuBLAS. This implementation is written in Julia, and uses CUDAnative.jl to compile code for the GPU and to execute it on the GPU.

In the case of array types that support computations with user code, we can also use code that is built around the higher-order array abstractions from Section 4.3.1. These abstractions compose with user code, and require the ability to generate code for the hardware that is targeted by the array type. For example, we can take the example from Listing 35 and change the call to `distribute` to create a `CuArray` instead. The CuArrays.jl package uses CUDAnative.jl to generate code for NVIDIA GPUs. Similarly, we can take the example from Listing 31 and execute it with arrays of type `DArray{Array}`, which would result in distributed execution on the CPU. `DArray` itself does not execute the user code but defers to the inner `Array`, which uses the Julia compiler to generate code for the CPU.

Application code can also perform scalar iterations over array elements, either because the application code is written that way or because (standard) library operations used in the application code are implemented as such. As explained in Section 4.4, this type of iteration defeats the purpose of heterogeneous programming as it cannot be implemented efficiently. Still, packages like CuArrays.jl and DistributedArrays.jl support this type of iteration because it greatly simplifies the effort of porting code. Initially, one can run the application on heterogeneous hardware without any change to the code, to verify the functional correctness of the implementation. Subsequently, performance can be improved by reimplementing methods that rely on scalar iteration using array abstractions that can be executed efficiently on heterogeneous hardware. Identifying the methods that need to be reimplemented is facilitated by API calls that disallows scalar iteration. For example, both CuArrays.jl and DistributedArrays.jl provide a configuration value `allowscalar` that, when set to `false`, triggers errors upon use of inefficient scalar functionality.

Typical applications also contain multiple allocation sites. For example, the `domeigen` function from Listing 20 not only takes an array as argument, but also allocates an output container for the resulting

eigenvector. To avoid hard-coding an array type, Julia provides functions such as `similar` to allocate new containers based on existing ones. These functions make it possible to write generic code that is independent from the chosen array type. The Julia standard library is built on top of these generic programming approaches, and rapid prototyping engineers can also use it, to facilitate reuse with different array types.

In summary, during prototyping, application code can be written independently from the underlying array types. Porting the code to different types optimized for different types of heterogeneous hardware during the prototyping or afterwards requires minimal code changes, and only serves to improve performance.

5.3.2 Library Portability

When applications use code from libraries, complexity is hidden behind opaque function calls whose implementations are outside immediate control of the application developer. These implementations can be complex, might themselves depend on auxiliary libraries, and should not have to be understood by the application developer in order to port application code to another platform.

Library code that works with arrays behaves similarly to application code as described in Section 5.3.1. As long as the library only uses functionality mandated by or implemented for `AbstractArray`, and allocates new containers using generic functions like `similar`, it is possible to reuse the library code with different array types.

However, where application code is often untyped, library code typically specifies types for function arguments [158]. For code to be portable, i.e., reusable with different array types, these function signatures should use abstract array types such as `AbstractArray` or `AbstractSparseVector` and not their concrete CPU instantiations such as `Array` or `SparseVector`.

This requirement poses no problem in practice, as Julia developers in general, and library developers in particular, are not unfamiliar with such patterns of using abstract types to achieve generic array program. Those patterns are in fact recurring elements in examples, documentation, and the standard library. Furthermore, many common operations on arrays return wrapper objects, for the purpose of lazy evaluation or to avoid allocations. Those objects require the code to be generic in order to benefit from said optimizations. For example, transposing a matrix results in an array of type `Transpose`, slicing produces a `SubArray`, etc. As a result, most library code is already type-generic and should be reusable in the context of heterogeneous array programming.

We conclude that the necessary technical support and developer culture are available and even convenient to achieve portability when domain-specific libraries are developed and used.

Use of ForwardDiff.jl

As a concrete library example, consider the ForwardDiff.jl package. It implements methods to compute different kinds of derivatives of arbitrary user-defined computations on arrays and their elements [127]. For example, in the ML example from Listing 21 the `gradient` and `derivative` functions are used to differentiate the loss function of a model for use by a gradient descent optimization algorithm. The ForwardDiff.jl package is an example of a high-quality, type-generic library. Simply changing the type of the arrays as passed to the derivatives makes the example from Listing 21 work on, e.g., a GPU, without requiring any other changes to either the code in Listing 21 or the underlying library.

However, the performance of the standard implementation of the ForwardDiff.jl package was not optimal when used with heterogeneous array types. To identify functionality that needs to be optimized, we disabled scalar iteration as described in Section 5.3.1. This revealed that certain methods of the `ForwardDiff.seed!` function were implemented using scalar for loops, one of which is shown in Listing 36. By reimplementing those methods using array abstractions as shown in Listing 37 they are better suited for execution on, e.g., a GPU. In this case, the replacement uses a `broadcast` expression as a substitute for the scalar for loop. The replacement code is not more complex, and performs almost identical to the original scalar implementation.[3]

When the need to redefine a library function to obtain higher performance in a specific application arises, either during or after the rapid-prototyping phase, the redefinition does not necessarily need to happen in the library itself. It can also be done in the application, by prefixing the function name with the contained module. For example, to implement the replacement of Listing 37 in an application rather than in the ForwardDiff.jl library, it suffices to write it down as `function ForwardDiff.seed! ... end`. When a replacement definition in an application has exactly the same signature as the original definition in the library, the replacement overrides the library version.

[3]The only exception is when using small arrays, where time to allocate an array view as part of the `broadcast` invocation is significant. Compiler improvements with respect to the garbage collector are expected to improve this: https://github.com/JuliaLang/julia/issues/14955

```
1 # original, scalar implementation
2 function seed!(duals::AbstractArray{Dual{T,V,N}}, x,
3                seeds::NTuple{N,Partials{N,V}}) where {T,V,N}
4     for i in 1:N
5         duals[i] = Dual{T,V,N}(x[i], seeds[i])
6     end
7     return duals
8 end
```

Listing 36: Scalar implementation of the `seed!` function in ForwardDiff.jl.

```
1 # replacement broadcasting version
2 function seed!(duals::AbstractArray{Dual{T,V,N}}, x,
3                seeds::NTuple{N,Partials{N,V}}) where {T,V,N}
4     duals[1:N] .= Dual{T,V,N}.(x[1:N], seeds[1:N])
5     return duals
6 end
```

Listing 37: Reimplementation of the `seed!` function from ForwardDiff.jl using array abstractions.

This capability can be very useful during rapid prototyping or performance optimization: it allows the engineer to overcome deficiencies in third-party libraries without requiring the immediate help of the owners of those libraries and without having to build and then later maintain custom versions of those libraries. The effects of these additional method definitions are global, and can be used to influence functionality deep down the library as opposed to only functions that are called directly.

Furthermore, the original definition in the library can easily be kept available for the purpose of verifying the replacement implementation. It suffices to use a dispatch signature that is limited to the heterogeneous array type of choice to avoid that the original definition is overridden. For example, by changing the method signature in Listing 37 to use `CuArray` instead of `AbstractArray` for the first argument, the broadcasting version would only be used for GPU arrays, and the known-good library implementation remains available for use with `Array` objects to verify semantic equivalence of the original and replacement definitions.

We conclude that even in the case when libraries are not fully portable with respect to array types and abstractions, convenient techniques are available to a user of the library to resolve the portability issues without unnecessarily delaying or complicating the prototyping.

5.3.3 Array Infrastructure Portability

The previous examples have used arrays in a fairly straightforward manner: User code instantiates a concrete subtype of the `AbstractArray` type to express *where* data is stored, array abstractions are used to describe *what* is going to be computed, and multiple dispatch is the core mechanism to influence *how* computation happens. This section demonstrates how this separation of concerns makes it possible to compose multiple array types, and enable reuse of array infrastructure.

Kronecker Products on the GPU

The example from Listing 22 uses a custom array type for efficiently computing the Kronecker product of two matrices, and provides an optimized implementation of the `norm` function computing the matrix norm using properties of the Kronecker product to improve performance. The `Kronecker` array type is generically typed, and only requires that the two input matrices should be part of the `AbstractArray` type hierarchy. No so-called *glue code* is required for the `Kronecker` type to work with concrete array types.

For example, we can create objects of type `Kronecker{CuArray}` by calling the `Kronecker` constructor with inputs of type `CuArray`. The resulting object can be used as if it were a generic array, with the `Kronecker` type influencing *what* is computed, while the `CuArray` type defines *how* and *where* the computation happens.

With only the `getindex` function for scalar indexing defined, array operations with objects of type `Kronecker{...}` dispatch to generic implementations as described in Section 4.4. However, any optimized method that calls functions on the underlying containers compiles to specialized code that uses functionality optimized for the contained array type. For example, with a Kronecker product of `CuArray`s and the optimized but still generically-typed implementation of the matrix norm from Listing 22, calls to the `norm` function result in an execution that combines the properties of the Kronecker product that allow for an efficient calculation of the norm with a well-optimized GPU implementation of the Euclidean norm that is available in the CuArrays.jl package and that in turn invokes the cuBLAS library. This powerful example illustrates how multiple array types, each dealing with separate concerns, seamlessly compose together to form a high-performance interface that can still be used generically.

Ideally, it should also be possible to use the broadcast abstraction from Section 4.3 in combination with custom array types. However,

currently that does not work out of the box. One problem is the implementation of the type hierarchy in relation to broadcasting when wrappers are combined. For example, `Kronecker{CuArray}` is a subtype of `AbstractArray`, but not of `CuArray`. In the current language implementation, the compiler's use of available methods optimized for `CuArray` to specialize code depends on the presence of certain artifacts in the `Kronecker` class method implementations, such as whether or not those (by accident) defer to the inner `CuArray`. That dependency on the occurrence of those artifacts should be avoided, as it violates the separation of concerns and limits composibility and performance portability in ways a non-expert programmer cannot easily handle.

We expect this situation to improve in the future, since heavy use of array wrapper types is relatively new, and the current broadcast infrastructure has been designed as recently as Julia 1.0. For now, array packages such as CuArrays.jl and DistributedArrays.jl provide the necessary definitions for common array wrappers, such as the ones from the Julia standard library, to work as expected. We have since generalized these definitions into a dedicated package,[4] for use by any array type that needs to customize the behavior of wrapped arrays.

Distributed GPU Arrays

Where the previous section combines array types that have separate responsibilities, we can also compose types that involve similar concerns. For example, both the CuArrays.jl and DistributedArrays.jl packages define array types that define *where* data is stored and *how* values are computed. The `DArray` type distributes data across multiple processes and prefers computations with local memory, while the `CuArray` type uses the GPU for storage and parallel execution. As explained in Section 5.2, the distributed chunks of a `DArray` are arrays, typically regular CPU-based `Array`s, but we can use `CuArray` as the underlying data array, and thereby distribute data and computations across multiple GPUs. For `DArray` to be able to wrap and manage an array, the type only needs to implement the object serialization interface.

Similar to the example in the previous section, an object of type `DArray{CuArray}` implements the `AbstractArray` interface and can therefore be used as any other array. This kind of infrastructure portability arises from a clear separation of concerns, each type implementing specific, fine-grained methods with minimal surface area. Both types are oblivious about one another and generic code can take advantage of them jointly.

[4] Available at https://github.com/JuliaGPU/Adapt.jl

Listing 34 is an example of how `DArray` separates the responsibilities of communication and computation. Computation is delegated to a different array type, may it be `Array` for CPU or `CuArray` for GPU execution. Similarly, `broadcast` of a `DArray` is implemented by delegating the computation to a different array type without having to specify which array types are supported. This allows new array types to be bootstrapped quickly and to take advantage of these rich abstractions. For example, a transposition of any array can be represented as an object of type `Transpose{...}` without that array having to solve the problem of transposing data itself. If there exists a better approach to transposing this kind of array, it can simply be implemented as an additional method of the `transpose` function, specialized for this type.

5.4 Performance

This section analyzes the performance of different array types applied to the examples from Section 4.1. We work with Julia version 1.0.1, using the official binaries from the homepage. The following auxiliary packages are used: CUDAdrv.jl 0.8.6, CUDAnative.jl 0.9.1, and ForwardDiff.jl 0.9.0. In the case of array packages, CuArrays.jl and DistributedArrays.jl, we used development branches to incorporate fixes and improvements to the array types that we developed while working on this functionality. Most of these changes have since been incorporated in the main development branches.

All measurement are done on a dual processor system, with two Intel quad-core Xeon E5-2637 v2 CPUs totaling 8 cores and with simultaneous multi-threading support for 16 threads. The system is equipped with 64 GiB of DDR3 ECC memory, while each CPU has 15 MiB of shared cache. The system also contains 2 NVIDIA GPUs: a Kepler-era GTX TITAN with 6GB memory, and a Pascal-era GeForce GTX 1080 with 8GB memory. We use a 64-bit Debian Stretch running Linux 4.9, with CUDA 9.0 on NVIDIA driver 390.87.

Code that targets the CPU by using the `Array` or `DArray{Array}` types is allowed to take advantage of the supported 16 simultaneous CPU threads. In the case of `Array`, this is done by configuring the OpenBLAS library that empowers many of the array abstractions as implemented for `Array` to use 16 threads. This implies a single-process multi-threaded parallelization. In the case of `DArray{Array}`, single-threaded multi-process parallelization is used instead. This is done by configuring the Distributed standard library that is used by DistributedArrays.jl to launch 16 worker processes, while OpenBLAS is configured to use only

thread per process to avoid oversubscription of the system. For measurements with a single GPU, we use the GeForce GTX 1080 in a single process. When targeting multiple GPUs, e.g., with the `DArray{CuArray}` type, we use one worker process per GPU.

We used the BenchmarkTools.jl package to collect accurate timings for the experiments [42]. These timings are not limited to the compute-intensive part of the experiment, including all relevant host and device interactions such as kernel launch overhead and RPC communication time, but do exclude the time to allocate memory and initialize data as realistic applications are expected to keep data on the device across individual operations. Measurements are performed on an otherwise idle system, after tuning in order to determine the required execution and sample count for each experiment to yield accurate timings. In the charts below, we report the mean execution time.

The performance evaluation below is limited in scope. We rely on existing array packages to perform well in the contexts they were designed for, i.e., CuArrays.jl for GPU execution and DistributedArrays.jl for execution on multi-core CPU computers and distributed systems. The measurements in this section serve to illustrate how the realistic problems from Section 4.1, built on top of the array abstractions from Chapter 4, can be used with various array types to effortlessly program heterogeneous systems and to benefit from the increased performance and/or enlarged scale these systems provide. This does not necessarily imply optimal or efficient use of the hardware, but we will show that our approach facilitates that goal.

5.4.1 Power Iteration

The example from Listing 20 is a simple application that uses array abstractions. It can trivially be executed with a variety of array types, for which it suffices to change the initial allocation site. Figure 5.1 shows how the execution time of the `domeigen` function evolves with the problem size. This time includes all run-time overhead such as the time to allocate output buffers, launch GPU kernels, and communicate data across compute nodes.

The results in Figure 5.1 highlight several performance characteristics. First of all, it is clear how regular multi-threaded `Array`s have very low overhead, and scale with increasing problem size as would be expected from working with $N \times N$ arrays. A distributed `DArray{Array}` works with multiple processes that require IPC (Inter-Process Communication). This comes with a significant overhead that will be discussed below, but with large problem sizes the performance shows to scale iden-

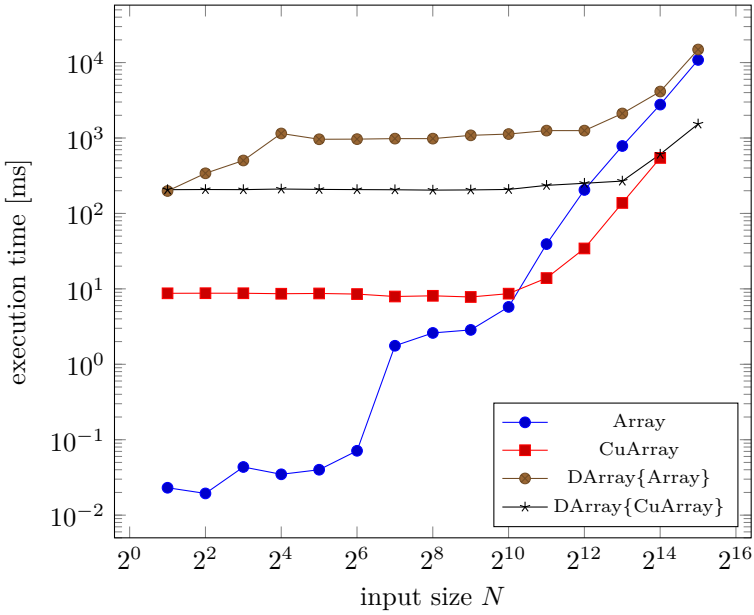

Figure 5.1: Time to execute the `domeigen` function from Listing 20 and compute the dominant eigenvector and eigenvalue of a $N \times N$ matrix. We benchmark for 1000 iterations of the power method, approximating the reference eigenvalue with sufficient accuracy. Best-case speedup over the multi-threaded CPU `Array`: 5.97× for `CuArray`, 0.73× for `DArray{Array}`, and 7.07× for `DArray{CuArray}`.

tically to multi-threaded arrays that do not require IPC. This shows how the use of `DArray{Array}` is viable for large problems, where performance of multiple processes with IPC is comparable to that of a multi-threaded application that does not require communication.

The `CuArray` measurements are for using a single GPU. Again there is a constant overhead that dominates the performance for small input sizes, albeit smaller than with `DArray{Array}`. This overhead is caused by interactions with the CUDA driver, such as allocating memory or launching GPU kernels. This overhead is quickly dwarfed for larger input sizes, however, by the performance improvements that result from using a GPU. These measurements show how performance of array applications that work with nontrivial data sets can be easily improved by using a GPU array type such as `CuArray`.

As GPUs typically have small memories, they are limited in the amount of data that can be processed. Although certain operations can be implemented with so-called out-of-core algorithms that support working set sizes larger than the available memory, and features like CUDA Unified Memory make it possible to do so without significantly changing code, these approaches come at a large performance cost [65, 89, 133]. We did not employ such techniques in the reported experiments. For that reason, the `CuArray` measurements stop at input size 2^{14}. The alternative solution of using multiple GPUs to extend the available memory requires careful management of data in order to reach good levels of performance. This data management has already been developed as part of DistributedArrays.jl, so we reuse that functionality via objects of type `DArray{CuArray}` to distribute data automatically across GPU devices. Figure 5.1 shows how this again comes with a large initial overhead for small input sizes, but ultimately the approach scales past the limits of using a single GPU and delivers performance that is better than the projected performance of using a single GPU past its maximal problem size, consistent with the increase in computing power that arises from using multiple GPUs. It shows how multiple GPUs can be easily used together to extend the supported working set size of an array application, while further improving performance despite inefficiencies in the current IPC implementation (see below).

Similarly, DistributedArrays.jl can be used to scale past single nodes without changes to the application, by using one of the cluster managers as explained in Section 5.2. This makes it possible to support working set sizes that exceed the available main memory, and to improve performance by adding more computational power than a single node has to offer. We will demonstrate this behavior using a computing system that contains multiple GPUs.

5.4.2 Performance of DistributedArrays.jl

The above results show that distributed arrays displays a constant overhead that only is amortized when the working data is sufficiently large. Some of that overhead is to be expected because IPC invariably involves communication, while types such as `Array` and `CuArray` require no such communication. That communication does not explain all the overhead, however. Some of it is actually caused by several inefficiencies in the current implementation of `DArray`.

The first major inefficiency stems from the fact that communication and computation share the same thread. Julia uses one event loop to schedule tasks and to allow forward progress to be made when a task is blocked on I/O. The event loop is current implemented using cooperative tasks, which can lead to the unfortunate situation that a worker busy with a computation and not yielding back to the event loop causes other tasks responsible for communication to stall. This in turns prevents other processes from making progress. Work is currently under way to move to a parallel thread runtime where this would not be an issue.[5]

Another slowdown is due to the many data copies occurring as part of IPC. The vector-matrix product on line 11 of Listing 20 requires sending parts of the vector to different processes. As part of that communication, extraneous copies of the data are made: The vector is first serialized on one process and copied to an IPC socket. Then it is deserialized from that socket on another process to be made available as a vector object again. There are also places within DistributedArray.jl where unnecessary additional copies are made, such as the current implementation of `copyto!(::Array, ::DArray)` where the remote data is first copied into a local buffer and then copied again into the output array. These redundant copies could be avoided by careful optimization, and communication could be improved, e.g., by using hardware capabilities such as RDMA (Remote Direct Memory Access) or NVLink for GPUs. Such optimizations are very local, and often only require certain method definitions. As an example, support for efficient communication between GPUs would require implementations of the `serialize` and `deserialize` methods for `CuArray` using the CUDA IPC programming interfaces. Since our system does not support NVLink, we did not add such definitions nor explore alternative approaches. For now, communication between GPUs happens through the CPU memory space.

Altogether, the current state of DistributedArrays.jl imposes significant communication overhead. As a result, the matrix-vector product used in Listing 20 shows little speed-up with `DArray{Array}`. It is bound

[5] https://github.com/JuliaLang/julia/pull/22631

by memory bandwidth and the cost of communication is much higher than the computational cost of the operation. When executing Listing 20 with `DArray{CuArray}`, the performance benefit of using GPUs overcomes that overhead.

Despite these limitations, distributed arrays are still useful, e.g., once the working set size is too big for one machine or one GPU, or simply when more computational power is required. Furthermore, in scenarios that require little communication, DistributedArrays.jl scales nicely as will be demonstrated below.

5.4.3 Kronecker Product

Computation of the Kronecker product from Listing 22 illustrates a scenario where much less communication is required. The Euclidean norm can easily be computed on local parts of the input arrays, after which the partial scalar results can be communicated and used to compute the total norm. Figure 5.2 shows how this does not affect measurements with multi-threaded `Array`s, which do not require IPC. The timings hence scale quadratically, as would be expected from processing the Kronecker product of $N \times N$ arrays.

For the sake of completeness, timings for a dense computation are also included, where the Kronecker product is first computed in full, yielding a $N^2 \times N^2$ matrix. Comparing measurements with these dense computation timings to the timings of the `Array` implementation that uses the Kronecker type shows the value of using a structured matrix for computing the Kronecker product and for the associated optimized implementation of, here, the matrix norm. Even for small N, computing the norm of two $N \times N$ input matrices as per the optimized implementation for Kronecker products is faster than materializing the product and computing the norm of a single $N^2 \times N^2$ matrix. The working set size is of course also significantly reduced.

With fewer demands on communication, the measurements for distributed CPU arrays using `DArray{Array}` show a much smaller overhead than were observed for the power iteration example. For significantly large problem sizes, not only is the scaling behavior identical to that of multi-threaded `Array`s that do not require IPC, the performance is in fact higher. This shows that the distributed `DArray{Array}` is not only interesting for extending the working set size using distributed systems, but that it can also improve performance on a single computer as long as the application does not require significant IPC. This performance improvement can be explained by the NUMA (Non-Unified Memory Architecture) architecture of our 2-processor system. In the

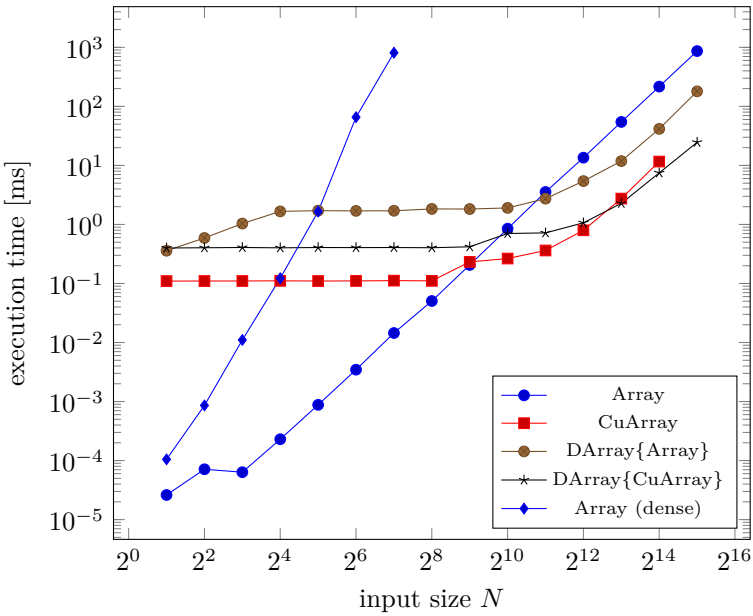

Figure 5.2: Time to compute matrix norm of the Kronecker product of two $N \times N$ matrices. Measurements marked with "dense" first compute the Kronecker product in full, while other measurements uses the structured matrix type from Listing 22 and the accompanying norm calculation from Listing 23. Best-case speedup of the sparse implementations over the multi-threaded CPU `Array`: 19.89× for `CuArray`, 5.20× for `DArray{Array}`, 35.15× for `DArray{CuArray}`.

case of `Array`, the entire array is allocated once on one of the NUMA nodes and processing from threads on a different NUMA node results in relatively slow memory accesses. With DistributedArrays.jl, data is explicitly partitioned across workers on the system. This results in data allocated in the local NUMA node, therefore minimizing memory traffic across NUMA zones.

Similar to the previous example, use of GPU hardware with `CuArray` objects significantly improves performance, but comes with a constant overhead that necessitates large input sizes. With `DArray{CuArray}`, we again manage to scale past the memory limit of a single GPU.

5.4.4 Proximal Gradient Descent

In Section 5.4.2 we mentioned a major performance penalty in the current implementation of DistributedArrays.jl due to inefficiencies with IPC. This is particularly noticeable in the machine learning example from Listing 21, where the main computational cost comes from matrix-vector multiplications as part of the `proximal_gradient_descent` method. These operations require significant communication, which is troublesome given the current implementation of IPC in DistributedArrays.jl. Indeed, Figure 5.3 shows how distributed execution with `DArray`s is dominated by the cost of communication, and even drowns out any performance benefits that come from using GPU hardware. In contrast, local execution with `CuArray` shows significant run-time improvements compared to CPU-based `Array`s, but is limited in terms of the working set size. As such, while the performance of distributed execution is far from optimal at this point, it makes it possible to scale beyond single devices and benefit from, e.g., the increase in available memory.

This example illustrates how application performance and potential improvements of using different array types are currently subject to application characteristics and how those influence the (composition of) the underlying array libraries. For example, Figure 5.3 shows how the example from Listing 21 benefits significantly from using a GPU, but currently does not improve when executed on a distributed system due to the heavy use of IPC. The example from Listing 20 does not rely as much on IPC, and Figure 5.1 shows how it benefits from using multiple GPUs in a distributed setting. At the other end of the spectrum, the example from Listing 22 does hardly use any IPC and as a result Figure 5.2 shows how use of distributed CPU and GPU resources yields significant speedups.

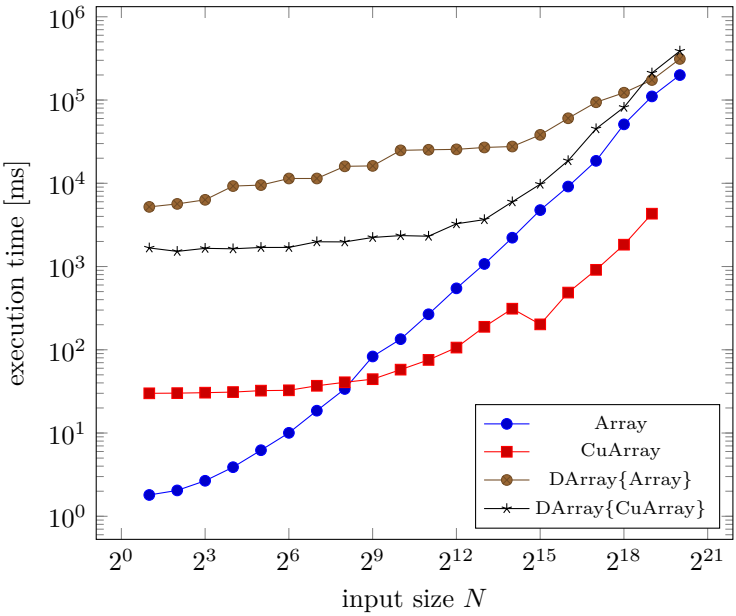

Figure 5.3: Time to perform 25 iterations of proximal gradient descent from Listing 21 to optimize a network of 100 parameters for N outputs. The implementation uses linear regression as a user defined model and performs enough iterations for the loss to reach 0.01 given random inputs from a normal distribution with $\mu = 0$ and $\sigma^2 = 1$. Best-case speedup over the multi-threaded CPU `Array`: 27.89× for `CuArray`, 0.64× for `DArray{Array}`, 0.62× for `DArray{CuArray}`.

5.5 Optimization Opportunities

The code examples analyzed so far have been written using high-level, idiomatic code that stays close to the mathematical definitions. This coding style is common with prototyping code, and as we have shown does still allow for good performance and portability towards heterogeneous computing environments.

After the initial prototyping phase in other high-level languages, developers typically rewrite (part of) their code in a high-performance language. With a high-level language that is designed for performance, as Julia is, this translation step can be avoided. Bezanson et al. [31] show how instead the Julia language features great performance from the get go, and makes it possible to optimize code within the language itself to the point where it reaches or even goes beyond the performance of statically compiled languages such as C or Fortran.

Furthermore, a one-language solution makes it easier for domain experts and code optimization experts to communicate and work together. Results can be passed between R&D and production teams, and prototyping code can be improved until fit for reuse by other projects or programmers. This avoids one-off solutions, improving the productivity and performance of future prototyping efforts.

5.5.1 Array Programming

In the case of array programming, common optimizations include using pre-allocated buffers and in-place operations for matrix operations, replacing operations on small containers with explicit loops, optimizing the iteration order, etc. By using generically typed functionality, or functionality that is expected to be implemented for all array types (such as methods from the `AbstractArray` interface, common linear algebra operations like matrix-matrix multiplication, etc), it is possible for such optimizations to be type-generic and reusable in the context of different array types.

As an example, consider how every iteration of the `for` loop in the `domeigen` function of Listing 20 allocates two temporary containers to store the outputs of the operations on line 14. Listing 38 shows an alternative version that pre-allocates two containers before the loop and uses in-place operations to prevent new allocations. This trivial optimization significantly improves performance, especially in the case of small inputs where the overhead of allocating memory is similar to the run time of the actual array operations. For example, with CPU arrays of size 64×64 or smaller, this optimization improves performance by

```
function domeigen(A, p)
  ...

  # power iteration
  b_k = b_0
  b_{k+1} = similar(b_k)
  for _ in 1:p
    mul!(b_{k+1}, A, b_k)

    # normalize
    b_k .= b_{k+1} ./ norm(b_{k+1})
  end

  ...
end
```

Listing 38: Optimization of the power iteration loop from Listing 20, using pre-allocated buffers and in-place array operations.

```
using CuArrays
using ForwardDiff: Chunk, DEFAULT_CHUNK_THRESHOLD

Chunk(x::CuArray, threshold = DEFAULT_CHUNK_THRESHOLD) = Chunk{8}()
```

Listing 39: Optimizing the use of ForwardDiff.jl from Listing 21 for GPU execution.

up to 15%. Furthermore, the change is fully generic and equally applies to other array types. With `CuArray`s, where memory allocations are not backed by a high-performance garbage collector, the improvements are about 5% for all matrix sizes as used in this evaluation.

5.5.2 Multiple Dispatch

Beyond optimizing the use of array abstractions, it is always possible to use multiple dispatch for providing fine-grained method overloads that optimize critical pieces of underlying functionality. One obvious example as discussed in Section 5.3 are method overloads that avoiding scalar iteration, e.g., in an underlying library that is used by the application. Although the main purpose of these overloads is to improve performance when working with heterogeneous computing devices, the implementations are often generic and can be used for all array types.

Method overloads can also be specific to an array type, and provide functionality that only optimizes execution for that type. For example, the ForwardDiff.jl library as used in Listing 21 performs partial derivative evaluations on the input vector in chunks [127]. Performing the evaluations on small smaller chunks uses less memory but requires more evaluations of the target function. In the case of GPU execution, larger chunks also require more registers, which might result in inefficient use of the GPU's parallel compute units. ForwardDiff.jl uses a heuristic to optimize the chunk size and minimize the amount of chunks given the size of the input vector. Listing 39 shows how to override that heuristic for GPU arrays by hard-coding an empirically-chosen chunk size that performs well given a specific application and GPU. Note that a production-quality version of this function would need to specialize on the performance characteristics of the GPU hardware that backs the input array.

Chapter 6
Automatic Differentiation of GPU Broadcast Kernels

Where the previous chapters have focused on using array abstractions for programmability and portability, we show in this chapter how they can be used for high-level algorithmic optimization. Specifically, we show how AD (Automatic Differentiation), a set of techniques to evaluate the derivative of a computer program, can be applied to differentiate broadcast operations. Our approach exploits the structure and semantics of the operation to derive efficiently on GPU hardware.

The work in this chapter was developed together with Jarrett Revels and Valentin Churavy of Massachusetts Institute of Technology. My contribution focuses on the GPU implementation, optimization, and performance evaluation. Our work builds on the CUDAnative.jl compiler from Chapter 3 and the CuArrays.jl library from Chapter 4, and on existing packages for AD in Julia by Jarrett Revels.

The scientific contribution of this chapter is an optimization that exploits the structure of the broadcast operation for the purpose of efficient automatic differentiation on GPUs. We demonstrate the use of this technique for ML, where gradient-based optimization is prevalent and GPUs are ubiquitous. Using a realistic model that includes dynamic control flow, we show that our approach enables efficient differentiation that would not be possible with the AD approaches as used by many popular ML frameworks. These contributions have been presented at a peer-reviewed conference workshop.[1]

[1] Jarrett Revels, Tim Besard, Valentin Churavy, Bjorn De Sutter, and Juan Pablo Vielma. "Dynamic Automatic Differentiation of GPU Broadcast Kernels". Presented at the Workshop on Systems for ML at the Conference on Neural Information Processing Systems (NeurIPS). 2018. arXiv: 1810.08297 [cs.MS].

6.1 Related Work

In recent years, the increased use of gradient-based optimization in ML has motivated an upsurge in the development of ML-specific modeling languages that incorporate AD as a fundamental feature. However, contemporary ML research routinely seeks to utilize new modeling and optimization techniques that push these frameworks' AD capabilities to – and past – their limit. Both practical and exploratory implementations of such techniques demand advanced features such as nested differentiation, differentiation through data-dependent control flow, domain-specific hardware specialization, distributed parallelism, checkpointing, and more [99, 43, 86, 94, 10, 116, 1].

In the pursuit of solutions capable of incorporating such features, it has become clear that modeling languages' *expressiveness* must necessarily be constrained for the sake of *differentiability*. Recent endeavors [153, 34, 60, 55, 152, 72, 144, 137] that explore this tradeoff have been guided by established methods from programming language theory, provoking the evolution of a new research area known as *differentiable programming*. This is quite a natural development, as the narrative of traditional AD research has always been richly intertwined with research into programming languages and mathematical programming.

The work in this chapter aims to extend the expressiveness of Julia as an ML modeling language by using mixed-mode AD to dynamically compute the derivatives of broadcast expressions. By taking advantage of the design of the broadcast operation as discussed in Chapter 4, we can efficiently derive operations that would otherwise be prohibited by the modeling language, such as data-dependent control flow. We will show that these features are relevant to real-life ML models.

6.2 Background: Automatic Differentiation

Automatic differentiation is a technique for evaluating the derivative of a function as specified by a computer program. It differs from symbolic differentiation, where mathematical rules are applied to differentiate a mathematical expression and not an arbitrary computer program. Conversion of a computer program to such a mathematical expression is often difficult, and incompatible with program constructs such as control flow. Instead, AD decomposes the computer program into a sequence of elementary arithmetic operations and functions for which the derivatives are known, and applies the chain rule to aggregate these results and compute derivatives of arbitrary expressions.

```
1 w1 = x1
2 w2 = x2
3 w3 = w1 * w2
4 w4 = sin(w1)
5 w5 = w3 + w4
6 y  = w5
```

Listing 40: Implementation of `y = x1 * x2 + sin(x1)` using only elementary arithmetic operations and functions.

For example,[2] given the mathematical expression $f(x_1, x_2) = x_1 \cdot x_2 + sin(x_1)$ we can write a simple computer program that evaluates `y = x1 * x2 + sin(x1)`. For the purpose of AD, we break down this program into a sequence of elementary arithmetic operations and functions in Listing 40 for which we can simply look up the derivatives.

To derive the program in Listing 40 with respect to its inputs x_1 and x_2, i.e., to compute the partial derivatives $\frac{\partial y}{\partial x_1}$ and $\frac{\partial y}{\partial x_2}$, we need to repeatedly apply the chain rule in order to propagate the derivatives of intermediate w values:

$$\frac{\partial y}{\partial t} = \frac{\partial y}{\partial w} \cdot \frac{\partial w}{\partial t}$$

For operations that depend on more than two variables, we can use the multivariable version of the chain rule:

$$\frac{\partial y}{\partial t} = \sum_i \frac{\partial y}{\partial w_i} \cdot \frac{\partial w_i}{\partial t}$$

6.2.1 Forward Mode

In forward-mode AD, we apply the aforementioned chain rule in a forwards fashion, and compute the derivative of an expression based on the derivatives of its inputs. For example, if we take the program from Listing 40 and derive each elementary expression with respect to some yet-to-determine variable t, we get the following expressions:

[2] This example was modeled after content by Berland [11] and Ruffwind [131].

```
1 dw1 = dx1
2 dw2 = dx2
3 dw3 = w2 * dw1 + w1 * dw2
4 dw4 = cos(w1) * dw1
5 dw5 = dw3 + dw4
6 dy  = dw5
```

Listing 41: Derived version of the program in Listing 40 with forward accumulation of intermediate derivatives. Execution of this program should follow execution of the original program from Listing 41.

$$
\begin{aligned}
\frac{\partial w_1}{\partial t} &= \frac{\partial x_1}{\partial t} \\
\frac{\partial w_2}{\partial t} &= \frac{\partial x_2}{\partial t} \\
\frac{\partial w_3}{\partial t} &= \frac{\partial w_3}{\partial w_1} \cdot \frac{\partial w_1}{\partial t} + \frac{\partial w_3}{\partial w_2} \cdot \frac{\partial w_2}{\partial t} = w_2 \cdot \frac{\partial w_1}{\partial t} + w_1 \cdot \frac{\partial w_2}{\partial t} \\
\frac{\partial w_4}{\partial t} &= \frac{\partial w_4}{\partial w_1} \cdot \frac{\partial w_1}{\partial t} = cos(w_1) \cdot \frac{\partial w_1}{\partial t} \\
\frac{\partial w_5}{\partial t} &= \frac{\partial w_5}{\partial w_3} \cdot \frac{\partial w_3}{\partial t} + \frac{\partial w_5}{\partial w_4} \cdot \frac{\partial w_4}{\partial t} = \frac{\partial w_3}{\partial t} + \frac{\partial w_4}{\partial t} \\
\frac{\partial y}{\partial t} &= \frac{\partial w_5}{\partial t}
\end{aligned}
\tag{6.1}
$$

As our original expression has multiple inputs, x_1 and x_2, we need to fix the independent variable with respect to which the differentiation is performed. We do so by picking a variable for t. For example, with $t = x_1$ the expressions from Equation 6.1 compute $\frac{\partial y}{\partial x_1}$, whereas with $t = x_2$ we get $\frac{\partial y}{\partial x_2}$.

In the corresponding program from Listing 41, choosing a variable for t affects the initial values for `dx1` and `dx2`. For example, with $t = x_1$ we get that `dx1` $= \frac{\partial x_1}{\partial t} = \frac{\partial x_1}{\partial x_1} = 1$ and `dx2` $= \frac{\partial x_2}{\partial t} = \frac{\partial x_1}{\partial x_2} = 0$. Put differently, if we want to compute the derivative with respect to x_1, we need to set `dx1` to 1 and `dx2` to 0.

These so-called *seed values* determine the behavior of the derived program from Listing 41. The fact that there are two seed values, one for each input variable, necessitates two evaluations in order to compute partial derivatives with respect to both input variables. This is a common requirement, e.g., to compute the gradient of a function.

```
1 dw5 = dy
2 dw4 = dw5
3 dw3 = dw5
4 dw2 = dw3 * w1
5 dw1 = dw3 * w2 + dw4 * cos(w1)
6 dx1 = dw1
7 dx2 = dw2
```

Listing 42: Derived version of the program in Listing 40 with reverse accumulation of intermediate derivatives. Execution of this program should follow execution of the original program from Listing 41.

6.2.2 Reverse Mode

To avoid the cost of multiple evaluations of the derived program, one for each combination of seed values and corresponding with the number of inputs, reverse mode AD computes the derivative of a program from the outputs to its inputs. It builds on the fact that the chain rule is symmetric, and can be reversed in order to compute the derivative of an operation based on the derivatives of its outputs:

$$\frac{\partial t}{\partial y} = \frac{\partial t}{\partial w} \cdot \frac{\partial w}{\partial y} = \sum_i \frac{\partial t}{\partial w_i} \cdot \frac{\partial w_i}{\partial y}$$

If we apply this to the program from Listing 40, we get the following expressions:

$$\begin{aligned}
\frac{\partial t}{\partial w_5} &= \frac{\partial t}{\partial y} \\
\frac{\partial t}{\partial w_4} &= \frac{\partial t}{\partial w_5} \cdot \frac{\partial w_5}{\partial w_4} = \frac{\partial t}{\partial w_5} \\
\frac{\partial t}{\partial w_3} &= \frac{\partial t}{\partial w_5} \cdot \frac{\partial w_5}{\partial w_3} = \frac{\partial t}{\partial w_5} \\
\frac{\partial t}{\partial w_2} &= \frac{\partial t}{\partial w_3} \cdot \frac{\partial w_3}{\partial w_2} = \frac{\partial t}{\partial w_3} \cdot w_1 \\
\frac{\partial t}{\partial w_1} &= \frac{\partial t}{\partial w_3} \cdot \frac{\partial w_3}{\partial w_1} + \frac{\partial t}{\partial w_4} \cdot \frac{\partial w_4}{\partial w_1} = \frac{\partial t}{\partial w_3} \cdot w_2 + \frac{\partial t}{\partial w_4} \cdot cos(w_1) \\
\frac{\partial t}{\partial x_1} &= \frac{\partial t}{\partial w_1} \\
\frac{\partial t}{\partial x_2} &= \frac{\partial t}{\partial w_2}
\end{aligned} \tag{6.2}$$

Where forward mode fixed the independent variable, with reverse-mode AD we need to fix the dependent variable to be differentiated. In the case of the example from Listing 40 there is only a single dependent variable, `y`, so we only have a single value to seed, `dy`, giving it a value of 1 as per `dy` $= \frac{\partial t}{\partial y} = \frac{\partial y}{\partial y} = 1$. Evaluation of the derived program from Listing 42 now directly yields two values, `dx1` and `dx2`, respectively computing the partial derivatives $\frac{\partial t}{\partial x_1} = \frac{\partial y}{\partial x_1}$ and $\frac{\partial t}{\partial x_2} = \frac{\partial y}{\partial x_2}$.

6.2.3 Forward vs. Reverse Mode

To calculate the derivatives of the outputs of a function with respect to its inputs, forward mode requires a number of evaluations of the derived program that scales with the number of inputs, while the performance of reverse mode depends on the number of outputs. However, closer inspection of the derived programs in Listing 42 reveals that, given constant seed values, common compiler optimizations such as constant propagation, common subexpression elimination or partial evaluation should render both programs and hence both approaches identical.

In practice, these optimizations don't suffice. For one, values that result from, e.g., multiplication by a seed value of zero cannot be dropped due to floating-point semantics. Called functions might have side effects, or the compiler might have trouble proving they are side-effect free, again inhibiting essential optimizations. Constant propagation and other optimizations are also typically intraprocedural and would not forward constant seeds to functions that are not inlined. In the case of the Julia language, specialization can be used to avoid this issue. Indeed, the ChainRules.jl package with primitives for automatic differentiation expresses seed values using the `Zero` and `One` types, forcing the compiler to specialize code while making it possible to use multiple dispatch for optimized implementations of problematic operations such as multiplication by zero.

Relying on compiler optimizations to equalize forward and reverse mode does however not generalize to expressions that work with vectors, as is common with deep-learning applications where inputs represent weights and parameters of the network. The derivative of these functions is a matrix, known as the Jacobian matrix, with each element a partial derivative with respect to certain elements of the input and output vectors. This has the effect that seed values are no longer constant, and cannot be used to specialize the derived program.

6.2.4 Mixed Mode

It is clear that forward and reverse-mode differentiation have different trade-offs with respect to the characteristics of the function under derivation, such as the input and output arity. However, "optimal" differentiation cannot be achieved via pure forward- or reverse-mode approaches, but rather demands a *mixed-mode* approach [111]. Achieving optimality, in this case, is often defined as minimizing the number of multiply-adds required to differentiate a given program via selecting the optimal mode for each region of code, and is known as the OJA (Optimal Jacobian Accumulation) problem. This problem has been shown to be NP-complete [110].

Despite this general theoretical intractability, mixed-mode AD offers a host of other advantages that can still be leveraged in practice by heuristically exploiting the local structure of the target language's primitive operations. The `broadcast` operation inherits such a structure: its kernels are scalar without cross-element dependencies, resulting in a highly-sparse Jacobian with zero-valued cross-element partial derivatives that makes it ideally suited for forward-mode differentiation [126]. Furthermore, this structure also applies to fused broadcast kernels, and enables differentiation of fully-fused broadcast subgraphs without requiring the construction of a backwards pass [126].

However, use of forward-mode differentiation can have certain disadvantages, even in the context of the broadcast operation. Why, then, is forward mode the better choice for this use case? The answer to this question can be summarized in three points:

1. If the input arity N is greater than the output arity M, then reverse mode is algorithmically superior to forward mode. However, broadcast operations generally have low arity (often < 10), and in practice, forward mode often outperforms reverse mode for low-arity functions regardless of the N/M ratio. There are two reasons for this. First, reverse-mode implementations often incur relatively high constant costs that are not amortized in the low-arity regime. Second, forward mode's additional chain rule applications can be offset for low-arity functions by leveraging stack allocation schemes that make better use of cache bandwidth and allow for the exploitation of instruction-level parallelism [127].

2. In the case that the target function contains data-dependent control flow, reverse-mode implementations must dynamically allo-

cate the data-dependent regions of the computation graph.[3] For low-arity functions, the overhead of dynamic trace allocation can easily dwarf the cost of the target function's primal evaluation. For broadcasted operations, this high overhead would be incurred for every elementwise invocation, rendering the reverse-mode approach in this case wholly unsuitable for the GPU where excessive dynamic allocation is infeasible.

3. Following from the previous point, using forward mode for broadcast differentiation allows data-dependent control flow to occur within broadcasted scalar operations, thus avoiding several disadvantages inherent to vectorized control flow primitives currently employed by reverse-mode frameworks (e.g., TensorFlow's `where` [1]). The first disadvantage is programmability; vectorized control flow primitives are often more cumbersome to use than their naive scalar counterparts. The second disadvantage is that many vectorized control flow primitives require computing untaken branches. While these primitives do have the benefit of clearly avoiding warp divergence on the GPU, the experiment described in Section 6.3 demonstrates that this benefit does not necessarily offset the cost of computing untaken branches on newer GPU architectures – especially if the difference in cost between branches is substantial – since newer architectures support executing different instructions across a warp without forcing serialized execution.

We have created the MixedModeBroadcastAD.jl[4] package to demonstrate mixed-mode AD that exploits the structure of broadcast to perform efficient forward-mode differentiation. Further details on this implementation can be found in Revels et al. [126]. As a consequence of using forward-mode AD, we can execute complex models that include data-dependent control flow on the GPU.

[3] This requirement is not implementation-specific, but rather a hard theoretical limit; capturing intermediate values which depend on run time data will always require run time allocation in the general case, though certain optimizations may alleviate this burden in special cases. This requirement applies even to reverse-mode tools that claim to be "tapeless" by statically generating backwards pass code [103, 72], or performing equivalent transformation via language-level constructs such as delimited continuations or closures [150]. As Pearlmutter and Siskind [117] remark, it is "impossible" to "eliminate the tape from reverse-mode AD" because "the tape stores intermediate values computed during the forward phase that are needed during the reverse phase."

[4] Available at `https://github.com/jrevels/MixedModeBroadcastAD.jl`

While the MixedModeBroadcastAD.jl package is meant as a proof-of-concept, the technique at large is part of popular Julia packages such as ReverseDiff.jl [125], Flux.jl [74] and Zygote.jl [72].

6.3 Evaluation

To compare our forward-mode broadcast differentiation technique with existing reverse-mode approaches, we describe an experiment based on the cell update calculation that occurs during the execution of an HM-LSTM (Hierarchical Multiscale LSTM) neural network [44]. We use three different AD implementations to calculate gradients: TensorFlow-based reverse mode, Julia-based reverse mode, and Julia-based forward mode. Each of these implementations are available in the MixedMode-BroadcastAD.jl repository. Finally, we analyze GPU performance measurements obtained from benchmarking these implementations.

6.3.1 HM-LSTM Cell Update

The HM-LSTM cell update calculation is a real-world example of a broadcast operation that is amenable to differentiation. For a given time step t and layer ℓ, the update calculation for the cell $\mathbf{c}_t^\ell$ is:

$$\mathbf{c}_t^\ell = \begin{cases} \sigma.(\mathbf{f}_t^\ell) \mathbin{.\times} \mathbf{c}_{t-1}^\ell \mathbin{.+} \sigma.(\mathbf{i}_t^\ell) \mathbin{.\times} \tanh.(\mathbf{g}_t^\ell) & \text{if } z_{t-1}^\ell = 0, z_t^{\ell-1} = 1 \text{ (UPDATE)} \\ \mathbf{c}_{t-1}^\ell & \text{if } z_{t-1}^\ell = 0, z_t^{\ell-1} = 0 \text{ (COPY)} \\ \sigma.(\mathbf{i}_t^\ell) \mathbin{.\times} \tanh.(\mathbf{g}_t^\ell) & \text{if } z_t^{\ell-1} = 1 \text{ (FLUSH)} \end{cases} \tag{6.3}$$

where $\mathbf{f}$ and $\mathbf{i}$ are memory gates, $\mathbf{g}$ is a cell proposal vector, and z is a boundary state.

We chose this operation as our experimental test case because it is self-contained, hinges on data-dependent control flow, has a substantial computational cost difference between branches, and is relevant to a machine learning audience.

The benchmarks described in the following sections are primarily concerned with the calculation of $\frac{\partial \mathbf{c}_t^\ell}{\partial \mathbf{c}_{t-1}^\ell}$, $\frac{\partial \mathbf{c}_t^\ell}{\partial \mathbf{f}_t^\ell}$, $\frac{\partial \mathbf{c}_t^\ell}{\partial \mathbf{i}_t^\ell}$, and $\frac{\partial \mathbf{c}_t^\ell}{\partial \mathbf{g}_t^\ell}$.

6.3.2 Reverse-Mode TensorFlow

The first implementation tested in our experiment was a TensorFlow-based implementation derived from Finkelstein [56]. This implementation, shown in Listing 43, makes use of TensorFlow's vectorized control

```
def hmlstm_update_c(z, zb, c, f, i, g):
    i = tf.sigmoid(i)
    g = tf.tanh(g)
    f = tf.sigmoid(f)
    return tf.where(
        tf.equal(z, tf.constant(1., dtype=tf.float32)),
        tf.multiply(i, g),
        tf.where(
            tf.equal(zb, tf.constant(0., dtype=tf.float32)),
            tf.identity(c),
            tf.add(tf.multiply(f, c), tf.multiply(i, g))
        )
    )
```

Listing 43: Implementation of the HM-LSTM cell-update calculation from Equation 6.3 in Python with TensorFlow.

flow primitive `where`, which eagerly computes both branches of the conditional statement before returning the branch specified by the given predicate. This primitive sidesteps actual branching and thus avoids two potential pitfalls discussed in Section 6.2: dynamic trace allocation and warp divergence. Avoiding these two perceived pitfalls comes at the cost of restricting the programming model and limiting opportunities for optimizations such as broadcast fusion.

The TensorFlow XLA (Accelerated Linear Algebra) compiler breaks this computation up into six separate kernels, as can be derived from the computational graph produced by TensorFlow's HLO (High Level Optimizer). Each kernel represents a partially fused region of the forward and reverse passes, including broadcasted `select` operations that were generated from the initial code's `where` invocations. One of these kernels is shown in Figure 6.1.

6.3.3 Reverse-Mode Julia

The second implementation tested in our experiment was a reverse-mode implementation in the Julia language [30]. This implementation was directly derived from the HLO graph of the TensorFlow implementation described in the previous section. The intent was to exactly mirror TensorFlow's operations at the abstraction level of its HLO representation, and enable a direct comparison between reverse-mode TensorFlow and reverse-mode Julia before comparing against a forward-mode Julia implementation. To accomplish this, the HLO graph operations were manually transcribed to native Julia code shown in Listing 44.

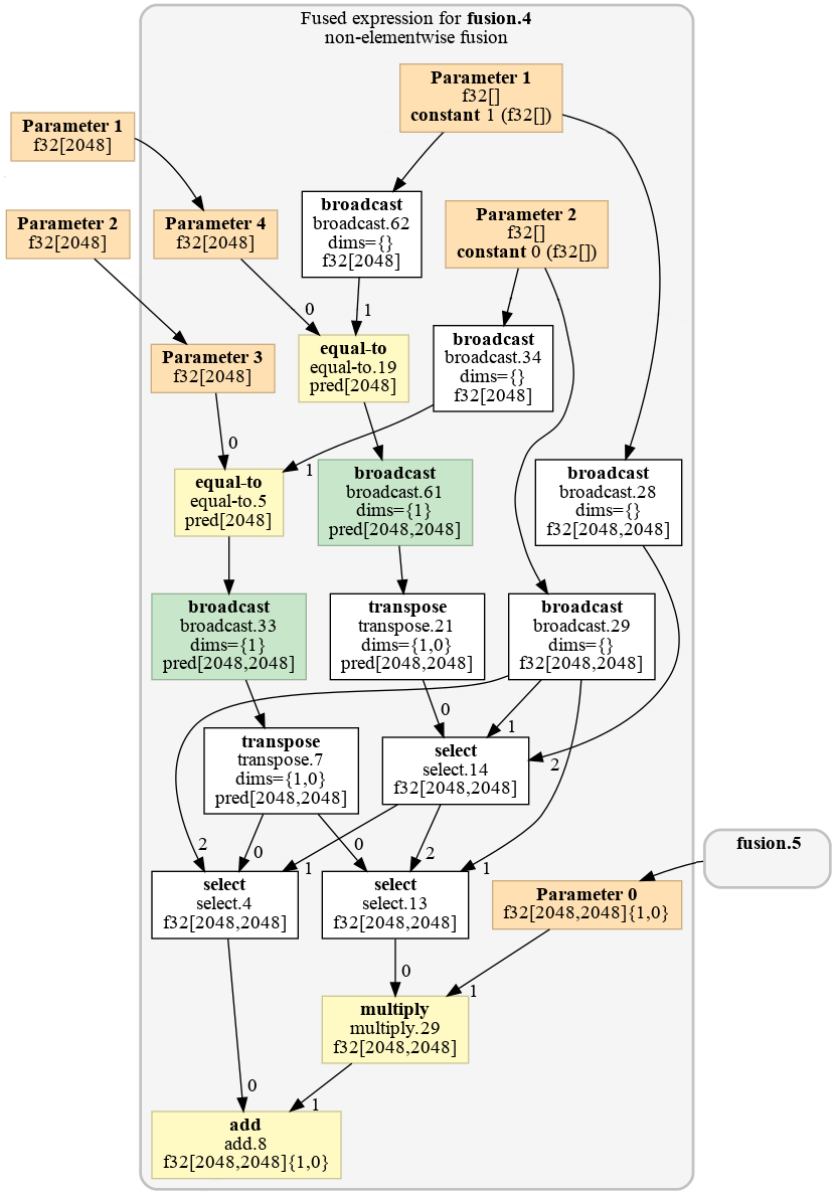

Figure 6.1: Partial HLO graph as emitted by the TensorFlow XLA compiler for the HM-LSTM cell-update implementation from Listing 43.

```
function hmlstm_update_c!(inputs, derivs, buffers)
    z, zb, c, f, i, g = inputs∇
    c, ∇f, ∇i, ∇g = derivs
    P0, P1, P2, P3, P4, P5 = c, z, zb, f, g, i

    fusion2 = fusion_2_or_5!(tanh, buffers[1], P5)
    tanh1   = broadcast!(tanh, buffers[2], P4)
    fusion5 = fusion_2_or_5!(tanh, buffers[3], P3)
    fusion  = fusion∇!(g, fusion2, tanh1, P1, P2)
    fusion1 = fusion1∇!(i, fusion2, tanh1, P1, P2)
    fusion3 = fusion3∇!(f, fusion5, P0, P1, P2)
    fusion4 = fusion4∇!(c, fusion5, P1, P2)
    return nothing
end

function fusion4∇!(c, fusion5, P1, P2)
    P4 = P1
    P3 = P2
    P2 = 0.0f0
    P1 = 1.0f0
    P0 = fusion5
    return broadcast∇!(c, P0, P1, P2, P3, P4) do p0, p1, p2, p3, p4
        equalto5 = p3 == p2
        equalto19 = p4 == p1
        select14 = ifelse(equalto19, p2, p1)
        select4 = ifelse(equalto5, select14, p2)
        select13 = ifelse(equalto5, p2, select14)
        multiply29 = select13 * p0
        return select4 + multiply29
    end
end
```

Listing 44: Partial implementation of the HM-LSTM cell-update calculation from Equation 6.3 in reverse-mode Julia that exactly mimics the execution by TensorFlow in Listing 43. Specifically, this listing shows the kernel from Figure 6.1.

```
function hmlstm_update_c(z, zb, c, f, i, g)
    if z == 1.0f0 # FLUSH
        return sigm(i) * tanh(g)
    elseif zb == 0.0f0 # COPY
        return c
    else # UPDATE
        return sigm(f) * c + sigm(i) * tanh(g)
    end
end
```

Listing 45: Implementation of the HM-LSTM cell-update calculation from Equation 6.3 in scalar Julia code, to be used with the broadcast abstraction.

6.3.4 Forward-Mode Julia

The third implementation tested in our experiment was a native Julia implementation of forward-mode broadcast differentiation. This implementation differs substantially from the previous reverse-mode implementations, most importantly the manner in which the primal calculation was expressed. While the reverse-mode implementations expressed control flow via vectorized primitives (e.g., `ifelse` in Listing 45), the forward-mode approach allows the fusion of control flow into the broadcasted kernel without incurring the reverse mode-specific performance penalties discussed in Section 6.2. Listing 45 shows the scalar computation of the cell update, fused into a single kernel by broadcasting over the model inputs:

$$\texttt{update}(c, f, i, g, z_1, z_2) = \begin{cases} \sigma(f) \times c + \sigma(i) \times \tanh(g) \\ \quad \text{if } z_1 = 0, z_2 = 1 \text{ (UPDATE)} \\ c \quad \text{if } z_1 = 0, z_2 = 0 \text{ (COPY)} \\ \sigma(i) \times \tanh(g) \\ \quad \text{if } z_2 = 1 \text{ (FLUSH)} \end{cases} \tag{6.4}$$

$$\mathbf{c}_t^\ell = \texttt{update.}(\mathbf{c}_{t-1}^\ell, \mathbf{f}_t^\ell, \mathbf{i}_t^\ell, \mathbf{g}_t^\ell, z_{t-1}^\ell, z_t^{\ell-1}) \tag{6.5}$$

An implementation of the $\mathbf{D}$ operator as described in Revels et al. [126] was then applied to Equation 6.4 to calculate the required gradients. The $\mathbf{D}$ operator itself was implemented using the multidimensional dual number type provided by the ForwardDiff.jl package, which represents a dual number as a pure Julia `struct` with two fields; one for the primal scalar, and one for a stack-allocated vector of perturbation coefficients [127].

6.4 Performance

Python code was executed using Python 3.6.3 with TensorFlow 1.5.0 and its XLA JIT compiler (which includes a copy of LLVM close to version 6.0). Julia code was executed using a development version of Julia 0.7, built on LLVM 6.0. All required versions of all Julia packages used are publicly available: CUDAnative.jl version 0.8.1, CUDAdrv.jl 0.8.3, LLVM.jl 0.9.8, and ForwardDiff 0.7.5. We used the CUDA toolkit at version 9.1.85, in combination with NVIDIA driver 390.30 and Linux 4.13 from Ubuntu 17.10.

The HM-LSTM implementations were tested on several generations of NVIDIA GPUs: a Tesla V100 (Volta), a Tesla P100 (Pascal), and a GTX Titan (Kepler), in combination with 2 hexa-core Intel Xeon E5-2603 v4 CPUs and 64 GiB of DDR4 memory.

We measure the performance of individual implementations using the NVIDIA profiling tools from the CUDA toolkit. We made sure that each implementation behaves identically from a CUDA API point of view, i.e., launching the same amount of kernels, performing identical allocations, etc. As such, we only report kernel timings, excluding, e.g., memory transfers and other CUDA API interactions,

For the sake of accurate comparison, our Julia-based benchmarks followed TensorFlow's configuration where possible, e.g., using page-locked memory allocated asynchronously using the driver API, performing an identical amount of memory transfers, launch kernels identically (using at most 64 threads and a corresponding number of blocks), etc.

6.4.1 Reverse Mode

Figure 6.2 shows the execution times to compute the aforementioned derivatives for each AD implementation across three generations of GPUs from NVIDIA, with $\mathbf{c}_{t-1}^{\ell}$, $\mathbf{f}_t^{\ell}$, $\mathbf{i}_t^{\ell}$, and $\mathbf{g}_t^{\ell}$ taking $n \times n$ random 32-bit floating-point matrix values and z_{t-1}^{ℓ} and $z_t^{\ell-1}$ taking n-element random 32-bit floating-point vector values where $n \in \{512, 1024, 2048\}$.

By comparing the performance of reverse-mode AD implemented in TensorFlow and Julia, with no semantic differences between both implementations, we aim to assess the overhead of using Julia for GPU programming. We have demonstrated in Chapter 3 that low-level kernel code performs competitively, yet Figure 6.2 shows a performance penalty on older hardware. This can be explained by the current implementation of the broadcast abstraction: part of the ability to work with heterogeneous containers is only materialized at run time, passing along iterators to default values for when extruding the dimensions of

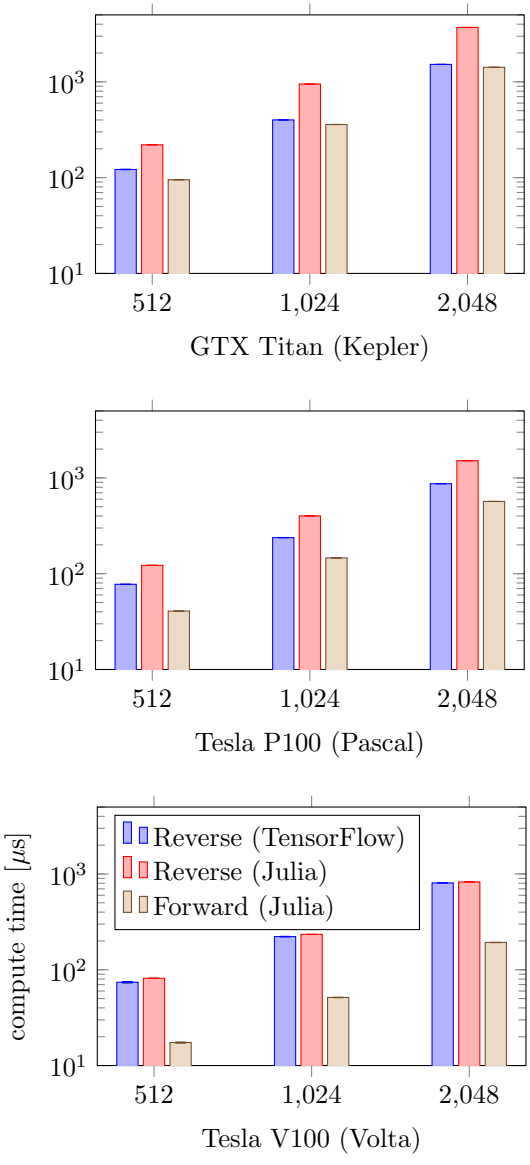

Figure 6.2: Total kernel compute times for the HM-LSTM cell-update calculation from Equation 6.3 across different AD implementations.

a container. As explained in Section 4.3.1, broadcasting over, e.g., a vector and a matrix results in a two-dimensional iteration space where the one-dimensional vector needs to be extruded into the second dimension. This is accomplished at run time, conditionally branching on the current indices and falling back to default values for any singleton dimension. These branches are relatively cheap on traditional multiprocessors, but perform badly on older generations of GPU hardware, further exacerbated by the thread divergence that they introduce. Meanwhile, XLA specializes the broadcasts with respect to the shape and size, effectively avoiding this run-time uncertainty. This kind of specialization is possible in Julia too, but has not been implemented in CuArrays.jl yet because of the neglegible impact on recent GPU hardware.

6.4.2 Forward Mode

As can be seen in Figure 6.2, the forward-mode Julia implementation features a speedup of 4.28x, 2.66x, and 2.60x over the reverse-mode Julia implementation on the Volta, Pascal, and Kepler architectures, respectively. Compared to the reverse-mode TensorFlow implementation, these speedups are 4.18x, 1.53x and 1.07x, respectively.

As mentioned in Section 6.2, a substantial advantage of the forward-mode approach is that it avoids the computation of untaken branches by allowing data-dependent control flow to be fused within the broadcasted operation itself. However, this kind of fine-grained branching has traditionally been considered unfavorable for GPUs, which typically require threads within a so-called "warp" (a group of typically 32 threads) to execute in lockstep. If threads within a warp branch to different instructions, the hardware must execute both branches on all threads within the warp and mask out the results of untaken branches on each thread. This is known as *warp divergence*, and can decrease performance significantly [32].

Fortunately, recent hardware improvements found on NVIDIA's Volta architecture can drastically mitigate the negative impact of warp divergence in many cases. This architecture enables independent thread scheduling by maintaining a program counter and call stack for every thread separately, thus allowing threads to execute different instructions without requiring serialized execution [114]. The effects of this architectural improvement can be seen in Figure 6.3, which shows the ratio of the forward-mode Julia implementation's execution time between uniformly distributed random control inputs and warp-uniform control inputs, thus controlling warp divergence as encountered during execution. Executing on a Tesla V100 GPU, the overhead of the thread-divergent

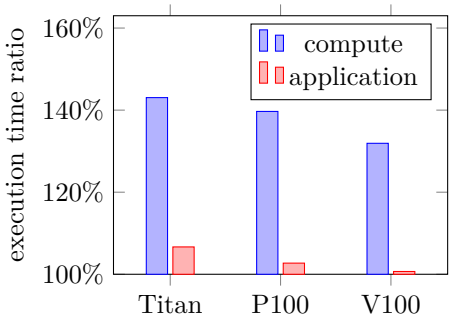

Figure 6.3: Total kernel compute and application execution times for the Julia forward-mode implementation, with random control inputs vs. warp-uniform control inputs.

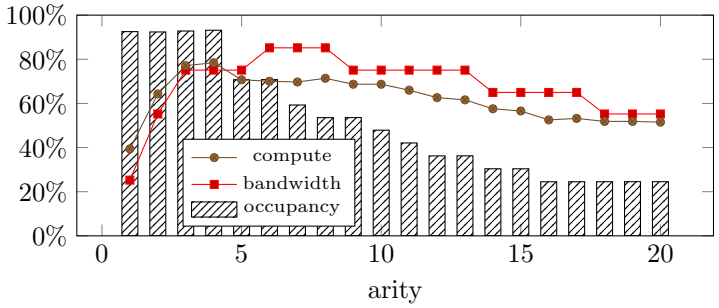

Figure 6.4: Effects of increasing operation arity on GPU compute utilization, memory bandwidth utilization, and kernel occupancy of a Tesla V100.

implementation in terms of kernel execution time drops from 40% to 30% on Kepler and Pascal. When looking at total application execution time, this cost is even lower (below 1%), as kernels also execute faster on more recent hardware.

6.4.3 Broadcast Arity

In addition to the main experiment, a different experiment was performed to explore how various indicators of GPU utilization scale as the arity of a forward mode-differentiated broadcast operation increases.

As previously stated, the ForwardDiff package's multidimensional dual number implementation utilizes a stack-allocated vector to store perturbation coefficients. By definition, the length of every input, output, and intermediate dual number's perturbation vector is equal to the input arity of the target operation. Thus, increasing the input arity of

the target operation increases register pressure. On CPUs, increased register usage can quickly result in excessive stack pressure, such that temporary values must be spilled into memory. On GPUs, however, many more registers are available; for example, the Tesla V100 contains 65,536 32-bit registers on each of its 84 SM (Shared Multiprocessor) [114]. This advantage is offset by the large number of threads executing concurrently on the GPU, since each thread reserves a number of registers for exclusive use. The balance between active thread count and register usage is captured by the *occupancy* metric, precisely defined as the ratio of active warps on an SM to the maximum number of active warps supported by the SM. With increased register usage, fewer warps can be allocated on each SM, and occupancy drops.

To assess the impact of a broadcasted operation's input arity on the performance of its forward-mode differentiation by dual numbers, we designed an artificial benchmark and measured its achieved occupancy and effective hardware utilization: the computation of $\mathbf{D}(f).(\mathbf{x}_1 \dots \mathbf{x}_N)$ where

$$f(x_1 \dots x_N) = \prod_i \tanh(g(x_i))$$

$$g(x) = \begin{cases} x & \text{if } x > \frac{1}{2} \\ -x & \text{otherwise} \end{cases}$$

and each $\mathbf{x}_j$ is a 1024×1024 random matrix with 32-bit floating-point elements. This benchmark provides a balanced workload for which it is easy to increase the arity N and measure the subsequent effect on hardware utilization. Figure 6.4 shows how occupancy drops steadily from 5 arguments on, at which point the amount of registers exceeds 32 and insufficient warps can be launched to satisfy the maximum number of concurrent warps per SM. Hardware utilization is initially limited by the low complexity of the kernel. It does not drop as strongly as the occupancy, since higher arity also increases the workload of the kernel, but at 18 arguments both compute and bandwidth utilization drop below 60% and the kernel can be considered latency-bound due to low occupancy.

Chapter 7
Status and Future Work

7.1 Code

The work presented in this dissertation is available as free and open-source software, published on GitHub.com, and can easily be installed using the Julia package manager. No special version of Julia is required, and all the contributions to Julia packages have been integrated with the respective development branches.

The interfaces from Chapter 2 have been part of Julia since version 0.6. Together with the LLVM.jl package as introduced in Section 2.5, they are used by the CUDAnative.jl compiler from Chapter 3, but also empower other packages that extend or alter the Julia compiler. This includes XLA.jl for executing Julia on Google TPUs [59], ExportWebAssembly.jl for exporting Julia code to WebAssembly [138], and MCAnalyzer.jl to instrument generated machine code for analysis by IACA (Intel Architecture Code Analyzer) or LLVM's MCA (Machine Code Analyzer) [45].

The Julia GPU stack for CUDA devices consists of several packages that have been created and/or are maintained by the author of this dissertation: CUDAapi.jl, providing re-usable components for CUDA API development; CUDAdrv.jl, wrapping the CUDA driver API; CUDAnative.jl, making it possible to compile and execute Julia for CUDA GPUs; and CuArrays.jl, building GPU array abstractions on top of CUDAnative.jl and the vendor libraries shipped with the CUDA toolkit. These packages do not consist of so-called "research code": They are documented, well-tested, and designed to work on a variety of user systems. This includes support for the major operating systems (Linux, macOS, and Windows), different versions of CUDA, 32 as well as 64-bit systems, etc. Every change is tested by several CI (Continuous Integration) services to keep track of this compatibility.

Several Julia applications and libraries rely on our GPU toolchain. Examples include Oceananigans.jl [46], a non-hydrostatic ocean model, Flux.jl [74], a popular machine-learning library, and Yao.jl [96], a framework for designing quantum algorithms. Work is under way to support use of CuArrays.jl in combination with Knet.jl [157], another machine-learning library, and DifferentialEquations.jl for solving differential equations on the GPU [120]. This list is not exhaustive: As demonstrated in Chapter 5, many applications and libraries already support and are actively using our GPU toolchain without an explicit dependency by virtue of the design of Julia's array abstractions.

7.2 Future Work

Although the work presented in this dissertation is already used as a foundation for a production-quality GPU programming environment, many improvements are still possible at each level of the stack. For example, the interfaces from Chapter 2 could be generalized: They are designed and implemented to cope with the restrictions of the GPU execution environment, while a more structured approach to represent troublesome operations would enable more powerful integration as well as further lower the implementation burden of retargeting Julia to other platforms. We are actively researching this approach for the purpose of native garbage collection and exception handling on the GPU.

At the language level, multiple dispatch is a powerful feature to override arbitrary functionality for specific argument types. It has enabled extensive reuse of the Julia standard library, by reimplementing specific methods that would result in GPU-incompatible code. Although this approach works well in the context of array operations, where we can dispatch on a GPU array type, for scalar kernels there is no GPU-specific type to dispatch on. By extending dispatch to incorporate a context, i.e., dispatching on the fact that code is to be executed on a GPU, we can override functions where multiple dispatch would fall short. Preliminary results based on Cassette.jl are promising, but require improvements to the Julia compiler in order to generate GPU-compatible code [149, 124].

GPU kernel programming in Julia with the compiler from Chapter 3 benefits from the productivity improvements that come with a high-level language: dynamically typed code, rich array types instead of simple pointers, etc. However, there is a large untapped potential of modeling GPU hardware features with high-level abstractions. For example, the explicit GPU memory hierarchy (global, shared, local, constant, texture and surface memory) could be modeled using an array type hierarchy. Complex operations like WMMA (Warp Matrix Multiply and Accumulate) could be represented by high-level expressions on static arrays. Initial research has shown that abstractions like these greatly improve programmer productivity, especially in contrast to the low-level CUDA C development environment that GPU programmers are used to [51].

Whereas abstractions at the kernel programming level would make it possible for expert programmers to use the GPU more efficiently, we also envision improved array abstractions that would benefit a much larger community of programmers. These abstractions, as introduced in Chapter 4, would focus on larger-scale operations such as efficient use of the GPU's multiple execution streams, or automatic distribution of operations across multiple GPUs. Initial results include the composability with DistributedArrays.jl from Chapter 5 for the purpose of programming clusters of GPUs. We are currently investigating other array applications and libraries that would benefit from GPU acceleration in order to determine bottlenecks and identify those operations that are crucial to GPU performance and programmability.

Beyond these improvements for the purpose of explicit GPU programming, we also envision Julia's IR to be well suited for high-level compiler analyses such as automatic parallelization, blocking and tiling transformations, etc. These optimizations would benefit from the rich type information that is still available at that stage of compilation, as demonstrated by some of the proof-of-concept compiler passes that are part of XLA.jl [59].

Chapter 8
Conclusion

In this dissertation I have presented techniques and abstractions for programming GPU hardware accelerators in a high-level programming language. To enable this, my work contributes several important improvements to different levels of the software development stack.

At the level of the programming language I have researched and developed interfaces to the high-level language's compiler that make it possible to alter the compilation process. Using those interfaces, an existing language can be retargeted to other platforms or execution environments without the need to reimplement significant parts of the compiler. This back end can coexist as an external package next to the main language implementation, without requiring any permanent changes to its compiler. I have implemented and contributed these interfaces for the Julia high-level programming language.

Building on the above interfaces I have developed the CUDAnative.jl back end for programming CUDA GPUs in Julia. The resulting GPU development environment offers unprecedented high-level programming capabilities for developing GPU kernels. At the same time, the generated code performs as if written in low-level CUDA C.

To further improve the GPU programming experience, I have worked on the CuArrays.jl package with array abstractions built on top of CUDAnative.jl. These abstractions present a data parallel programming model that obviates any GPU programming experience, and are well-suited to express scientific algorithms and engineering applications in a clear and concise manner. The availability of an GPU JIT compiler makes these abstractions very versatile, enabling both the design of novel higher-level abstractions, and the ability to optimize code at the lower-level kernel programming interface of CUDAnative.jl.

I have also demonstrated how the design of these array abstractions makes it possible to write portable applications that can be used across hardware platforms. Again by virtue of a JIT compiler to generate specialized code, it is possible to extend this portability to radically different execution environments as well as make it possible to reuse existing GPU-agnostic applications and libraries.

The design of certain array abstractions can also be exploited for the purpose of algorithmic optimizations. I have illustrated how the `broadcast` abstraction can be used to efficiently compute the derivative of array expressions in a GPU-friendly manner. I have worked on a GPU implementation of this technique based on CUDAnative.jl and CuArrays.jl, leading to better performance than the approaches taken by contemporary machine learning frameworks.

The high-level programming capabilities as presented in this dissertation improve the productivity of GPU programming at different levels of abstraction, but have not radically changed the core programming interface. I envision future research to further exploit the high-level language's features in order to improve productivity, develop novel abstractions for the GPUs esoteric hardware features, or do away with the kernel-based programming interface altogether and rely on Julia's high-level IR to empower replacement abstractions.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. "TensorFlow: A System for Large-Scale Machine Learning". In: *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2016, pp. 265–283.

[2] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L Steele Jr, Sam Tobin-Hochstadt, Joao Dias, Carl Eastlund, et al. *The Fortress Language Specification*. 2005.

[3] Srinivas Aluru and Nagakishore Jammula. "A Review of Hardware Acceleration for Computational Genomics". In: *IEEE Design & Test* 31.1 (2014).

[4] Edward Anderson, Zhaojun Bai, Christian Bischof, L Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, et al. *LAPACK Users' guide*. SIAM, 1999.

[5] Todd A Anderson, Hai Liu, Lindsey Kuper, Ehsan Totoni, Jan Vitek, and Tatiana Shpeisman. "Parallelizing Julia with a Non-Invasive DSL". In: *LIPIcs-Leibniz International Proceedings in Informatics*. Vol. 74. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2017.

[6] John Bachan, Dan Bonachea, Paul H. Hargrove, Steve Hofmeyr, Mathias Jacquelin, Amir Kamil, Brian van Straalen, and Scott B. Baden. "The UPC++ PGAS Library for Exascale Computing". In: *Proceedings of the PGAS Applications Workshop (PAW)*. ACM, 2017, 7:1–7:4.

[7] Satish Balay, William D Gropp, Lois Curfman McInnes, and Barry F Smith. "Efficient Management of Parallelism in Object-oriented Numerical Software libraries". In: *Modern Software Tools for Scientific Computing*. Springer, 1997, pp. 163–202.

[8] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. "Legion: Expressing Locality and Independence with Logical Regions". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, 2012, pp. 1–11.

[9] Matt Bauman. *Extensible Broadcast Fusion*. 2018. URL: https://julialang.org/blog/2018/05/extensible-broadcast-fusion.

[10] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. "Automatic Differentiation In Machine Learning: A Survey." In: *Journal of Machine Learning Research (JMLR)* 18.153 (2017), pp. 1–153.

[11] Håvard Berland. "Automatic Differentiation". 2006. URL: http://www.robots.ox.ac.uk/~tvg/publications/talks/autodiff.pdf.

[12] Tim Besard. "Effectively using GPUs with Julia". Presented at the third Belgian Julia user meetup (Ghent, Belgium, Dec. 21, 2018).

[13] Tim Besard. "High-Level GPU Programming with CUDA.jl". Presented at the first Belgian Julia user meetup (Ghent, Belgium, Sept. 29, 2015).

[14] Tim Besard. "High-Level Language Design for Extensible Accelerator Programming". Presented at the Workshop on High-Level Programming for Heterogeneous and Hierarchical Parallel Systems (HLPGPU) at the Conference on High Performance and Embedded Architecture and Compilation (HiPEAC) (Valencia, Spain, Jan. 23, 2019).

[15] Tim Besard. "High-Level Language Design for Extensible Accelerator Programming". Presented at the Workshop on Embedded Multicore Programming at the HiPEAC Computing Systems Week (CSW) (Edinburgh, United Kingdom, Apr. 17, 2019).

[16] Tim Besard. "Interfacing with LLVM using LLVM.jl". Presented at JuliaCon 2017 (Berkeley, CA, United States, June 22, 2017).

[17] Tim Besard. "Introduction to Julia". Presented at the Belgian TensorFlow user meetup (Ghent, Belgium, Feb. 6, 2019).

[18] Tim Besard. "Julia on the GPU". Presented at the second Belgian Julia user meetup (Brussels, Belgium, Apr. 13, 2016).

[19] Tim Besard. "Just Compile It: High-level Programming on the GPU with Julia". Presented at the European LLVM Developers Meeting (EuroLLVM). 2019.

[20] Tim Besard. "Programming NVIDIA GPUs in Julia with CUDAnative.jl". Presented at JuliaCon 2017 (Berkeley, CA, United States, June 21, 2017).

[21] Tim Besard, Valentin Churavy, and Simon Danisch. "GPU Programming with Julia". Presented at JuliaCon 2017 (Berkeley, CA, United States, June 20, 2017).

[22] Tim Besard, Valentin Churavy, Alan Edelman, and Bjorn De Sutter. "Rapid Software Prototyping for Heterogeneous and Distributed Platforms". In: *Advances in Engineering Software (AES)* (2019). DOI: 10.1016/j.advengsoft.2019.02.002.

[23] Tim Besard, Bjorn De Sutter, Andrés Frías-Velázquez, and Wilfried Philips. "Case Study of Multiple Trace Transform Implementations". In: *International Journal of High Performance Computing Applications (IJHPCA)* 29.4 (2015), pp. 489–505. DOI: 10.1177/1094342015584091.

[24] Tim Besard, Christophe Foket, and Bjorn De Sutter. "Effective Extensible Programming: Unleashing Julia on GPUs". In: *Transactions on Parallel and Distributed Systems (TPDS)* (2018). ISSN: 1045-9219. DOI: 10.1109/TPDS.2018.2872064. arXiv: 1712.03112 [cs.PL].

[25] Tim Besard and Mike Innes. "Julia: A Fresh Approach to GPU Computing". Presented at the GPU Technology Conference (GTC) (San Jose, CA, United States, Mar. 28, 2018).

[26] Jeff Bezanson. "Abstractions in Technical Computing". PhD thesis. Massachusetts Institute of Technology, 2015.

[27] Jeff Bezanson. "Why is Julia Fast? Can it be Faster?" Presented at JuliaCon India. 2015.

[28] Jeff Bezanson, Jiahao Chen, Stefan Karpinski, Viral Shah, and Alan Edelman. "Array Operators using Multiple Dispatch: A Design Methodology for Array Implementations in Dynamic Languages". In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ACM, 2014.

[29] Jeff Bezanson, Benjamin Chung, Jiahao Chen, Stefan Karpinski, Viral B. Shah, Jan Vitek, and Lionel Zoubritzky. "Julia: Dynamism and Performance Reconciled by Design". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), p. 120.

[30] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. "Julia: A Fresh Approach to Numerical Computing". In: *SIAM Review* 59.1 (2017), pp. 65–98.

[31] Jeff Bezanson, Stefan Karpinski, Viral B Shah, and Alan Edelman. "Julia: A Fast Dynamic Language for Technical Computing". 2012. arXiv: 1209.5145 [cs.PL].

[32] Piotr Bialas and Adam Strzelecki. "Benchmarking the Cost of Thread Divergence in CUDA". In: *International Conference on Parallel Processing and Applied Mathematics (PPAM)*. Springer. 2015, pp. 570–579.

[33] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. "Cilk: An Efficient Multithreaded Runtime System". In: *Journal of Parallel and Distributed Computing* 37.1 (1996), pp. 55–69.

[34] Olivier Breuleux and Bart van Merriënboer. "Automatic Differentiation in Myia". In: (2017).

[35] Xing Cai, Hans Petter Langtangen, and Halvard Moe. "On the Performance of the Python Programming Language for Serial and Parallel Scientific Computations". In: *Scientific Programming* 13.1 (2005), pp. 31–56.

[36] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. "Kokkos: Enabling Manycore Performance Portability through Polymorphic Memory Access Patterns". In: *Journal of Parallel and Distributed Computing* (2014), pp. 3202–3216.

[37] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. "Copperhead: Compiling an Embedded Data Parallel Language". In: *ACM SIGPLAN Notices* 46.8 (2011).

[38] Manuel MT Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. "Accelerating Haskell Array Codes with Multicore GPUs". In: *Proceedings of the 6th Workshop on Declarative Aspects of Multicore Programming (DAMP)*. ACM. 2011, pp. 3–14.

[39] Bradford L Chamberlain, David Callahan, and Hans P Zima. "Parallel Programmability and the Chapel Language". In: *International Journal of High Performance Computing Applications (IJHPCA)* 21.3 (2007), pp. 291–312.

[40] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing". In: *ACM SIGPLAN Notices*. Vol. 40. 10. ACM. 2005, pp. 519–538.

[41] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. "Rodinia: A Benchmark Suite for Heterogeneous Computing". In: *International Symposium on Workload Characterization (IISWC)*. 2009.

[42] Jiahao Chen and Jarrett Revels. "Robust Benchmarking in Noisy Environments". 2016. arXiv: 1608.04295 [cs.PF]. URL: https://github.com/JuliaCI/BenchmarkTools.jl.

[43] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. "TVM: End-to-End Compilation Stack for Deep Learning". Presented at the Conference on Systems and Machine Learning (SysML). 2018.

[44] Junyoung Chung, Sungjin Ahn, and Yoshua Bengio. "Hierarchical Multiscale Recurrent Neural Networks". 2016. arXiv: 1609.01704 [cs.LG].

[45] Valentin Churavy. *MCAnalyzer.jl: A Set of Tools for Machine Code Analyzing of Julia Code*. 2019. URL: https://github.com/vchuravy/MCAnalyzer.jl.

[46] Climate Modeling Alliance. *Oceananigans.jl: A Fast and Friendly Mon-hydrostatic Ocean Model in Julia*. URL: https://github.com/climate-machine/Oceananigans.jl/.

[47] Continuum Analytics. "Anaconda Accelerate: GPU-Accelerated Numerical Libraries for Python". 2017. URL: https://docs.anaconda.com/accelerate/.

[48] Dave Cunningham, Rajesh Bordawekar, and Vijay Saraswat. "GPU Programming in a High Level Language: Compiling X10 to CUDA". In: *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*. ACM. 2011, p. 8.

[49] Leonardo Dagum and Ramesh Menon. "OpenMP: An Industry-Standard API for Shared-Memory Programming". In: *IEEE Computing in Science & Engineering* (1998), pp. 46–55.

[50] Alain Darte. "On the Complexity of Loop Fusion". In: *Parallel Computing* 26.9 (2000), pp. 1175–1193. DOI: 10.1016/s0167-8191(00)00034-x.

[51] Yasser Deceukelier. "An Extensible API for Productive GPU Programming in Julia". MA thesis. Ghent University, 2016.

[52] Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S Duff. "A Set of Level 3 Basic Linear Algebra Subprograms". In: *ACM Transactions on Mathematical Software* 16.1 (1990).

[53] 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". In: *Parallel Computing* 38.8 (2012), pp. 391–407.

[54] Daniel L. Dvorak. "NASA Study on Flight Software Complexity". In: *AIAA Infotech@Aerospace Conference*. 2009, p. 1882.

[55] Conal Elliott. "The Simple Essence of Automatic Differentiation". In: *Proceedings of the ACM on Programming Languages* 2.70 (2018), pp. 1–29. ISSN: 2475-1421. DOI: 10.1145/3236765.

[56] Paige Finkelstein. *A Tensorflow Implementation of a Hierarchical and Multiscale RNN*. 2017. URL: https://github.com/bolducp/hierarchical-rnn.

[57] Keno Fischer. *The Julia C++ Interface*. 2018. URL: https://github.com/Keno/Cxx.jl.

[58] Keno Fischer and Jameson Nash. "Getting to Machine Learning from a General Purpose Compiler". Presented at the C4ML Workshop at the Conference on Code Generation and Optimization (CGO). 2019.

[59] Keno Fischer and Elliot Saba. "Automatic Full Compilation of Julia Programs and ML Models to Cloud TPUs". Presented at the Workshop on Systems for ML at the Conference on Neural Information Processing Systems (NIPS). 2018. arXiv: 1810.09868 [cs.PL]. URL: https://github.com/JuliaTPU/XLA.jl.

[60] Roy Frostig, Matthew James Johnson, and Chris Leary. "Compiling Machine Learning Programs via High-Level Tracing". Presented at the Conference on Systems and Machine Learning (SysML). 2018.

[61] Juan Fumero, Michel Steuwer, Lukas Stadler, and Christophe Dubach. "Just-in-time GPU Compilation for Interpreted Languages with Partial Evaluation". In: *ACM SIGPLAN Notices*. Vol. 52. 7. ACM. 2017, pp. 60–73.

[62] Mosè Giordano. "Uncertainty Propagation with Functionally Correlated Quantities". 2016. arXiv: 1610.08716 [physics.data-an].

[63] Bart Goossens, Jonas De Vylder, and Wilfried Philips. "Quasar—a new heterogeneous programming framework for image and video processing algorithms on CPU and GPU". In: *2014 IEEE International Conference on Image Processing (ICIP)*. IEEE. 2014, pp. 2183–2185.

[64] Robert Groth. "Is the Software Industry's Productivity Declining?" In: *IEEE Software* 21.6 (2004), pp. 92–94.

[65] Takahiro Harada. "A Framework to Transform In-Core GPU Algorithms to Out-of-Core Algorithms". In: *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*. ACM. 2016, pp. 179–180.

[66] Mark Harris, Shubhabrata Sengupta, and John D Owens. "Parallel Prefix Sum with CUDA". In: *GPU Gems* 3.39 (2007), pp. 851–876.

[67] Robert L Henderson. "Job Scheduling under the Portable Batch System". In: *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*. Springer. 1995, pp. 279–294.

[68] Troels Henriksen, Niels GW Serup, Martin Elsman, Fritz Henglein, and Cosmin E Oancea. "Futhark: Purely Functional GPU-Programming with Nested Parallelism and In-Place Array Updates". In: *ACM SIGPLAN Notices* 52.6 (2017), pp. 556–571.

[69] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. "An Overview of the Trilinos Project". In: *ACM Transactions on Mathematical Software* (2005), pp. 397–423.

[70] Jared Hoberock and Nathan Bell. "Thrust: A Parallel Template Library". 2010. URL: https://developer.nvidia.com/thrust.

[71] Leslie Hogben. *Handbook of Linear Algebra*. Discrete Mathematics and Its Applications. Chapman and Hall/CRC, 2006.

[72] Mike Innes. "Don't Unroll Adjoint: Differentiating SSA-Form Programs". Presented at the Workshop on Systems for ML at the Conference on Neural Information Processing Systems (NIPS). 2018. arXiv: 1810.07951 [cs.PL]. URL: https://github.com/FluxML/Zygote.jl.

[73] Mike Innes, Stefan Karpinski, Viral Shah, David Barber, Pontus Stenetorp, Tim Besard, James Bradbury, Valentin Churavy, Simon Danisch, Alan Edelman, Jon Malmaud, Jarrett Revels, and Deniz Yuret. "On Machine Learning and Programming Languages". Presented at the Conference on Systems and Machine Learning (SysML). 2018.

[74] Mike Innes, Elliot Saba, Keno Fischer, Dhairya Gandhi, Marco Concetto Rudilosso, Neethu Mariya Joy, Tejan Karmali, Avik Pal Singh, and Viral Shah. "Fashionable Modelling with Flux". Presented at the Workshop on Systems for ML at the Conference on Neural Information Processing Systems (NIPS). 2018. arXiv: 1811.01457 [cs.PL]. URL: https://github.com/FluxML/Flux.jl.

[75] Philip M Johnson, Hongbing Kou, Michael Paulding, Qin Zhang, Aaron Kagawa, and Takuya Yamashita. "Improving Software Development Management through Software Project Telemetry". In: *IEEE software* 22.4 (2005), pp. 76–85.

[76] Steven G. Johnson. *More Dots: Syntactic Loop Fusion in Julia*. 2017. URL: https://julialang.org/blog/2017/01/moredots.

[77] Julia developers. *CLArrays.jl: OpenCL-Backed GPU arrays for Julia*. 2018. URL: https://github.com/JuliaGPU/CLArrays.jl.

[78] Julia developers. *CuArrays.jl: CUDA-Accelerated Arrays for Julia*. 2019. URL: https://github.com/JuliaGPU/CuArrays.jl.

[79] Julia developers. *CUBLAS.jl: Julia Interface to cuBLAS*. 2017. URL: https://github.com/JuliaGPU/CUBLAS.jl.

[80] Julia developers. *DistributedArrays.jl: Distributed Arrays in Julia*. 2019. URL: https://github.com/JuliaParallel/DistributedArrays.jl.

[81] Julia developers. *PackageCompiler.jl: Compile your Julia Package*. 2019. URL: https://github.com/JuliaLang/PackageCompiler.jl.

[82] Christoforos Kachris and Dimitrios Soudris. "A Survey on Reconfigurable Accelerators for Cloud Computing". In: *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2016, pp. 1–10.

[83] Ken Kennedy and Kathryn S. McKinley. "Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution". In: *Languages and Compilers for Parallel Computing*. Berlin, Heidelberg: Springer, 1994, pp. 301–320. ISBN: 978-3-540-48308-3.

[84] Jiri Kraus. *An Introduction to CUDA-Aware MPI*. 2013. URL: https://devblogs.nvidia.com/introduction-cuda-aware-mpi/.

[85] Mayuram S Krishnan, Charlie H Kriebel, Sunder Kekre, and Tridas Mukhopadhyay. "An Empirical Analysis of Productivity and Quality in Software Products". In: *Management Science* 46.6 (2000), pp. 745–759.

[86] Griffin Lacey, Graham W. Taylor, and Shawki Areibi. "Deep Learning on FPGAs: Past, Present, and Future". 2016. arXiv: 1602.04283 [cs.DC].

[87] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. "Numba: A LLVM-based Python JIT Compiler". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. ACM. 2015, 7:1–7:6.

[88] P Lancaster and HK Farahat. "Norms on Direct Sums and Tensor Products". In: *Mathematics of Computation* 26.118 (1972), pp. 401–414.

[89] Raphael Landaverde, Tiansheng Zhang, Ayse K Coskun, and Martin Herbordt. "An Investigation of Unified Memory Access Performance in CUDA". In: *Proceedings of the High Performance Extreme Computing Conference (HPEC)*. IEEE. 2014, pp. 1–6.

[90] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. 2004, pp. 75–86.

[91] Chris Lattner and Jacques Pienaar. "MLIR Primer: A Compiler Infrastructure for the End of Moore's Law". Presented at the C4ML Workshop at the Conference on Code Generation and Optimization (CGO). 2019.

[92] Roland Leißa, Klaas Boesche, Sebastian Hack, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller, and Bertil Schmidt. "AnyDSL: A Partial Evaluation Framework for Programming High-Performance Libraries". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), p. 119.

[93] Calle Lejdfors and Lennart Ohlsson. "PyGPU: A High-Level Language for High-Speed Image Processing". In: *Proceedings of the International Conference on Applied Computing (ACIT)*. IADIS. 2007, pp. 66–81.

[94] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. "Differentiable Programming for Image Processing and Deep Learning in Halide". In: *ACM Transactions on Graphics (TOG)* 37.4 (2018), p. 139.

[95] Xuechao Li, Po-Chou Shih, Jeffrey Overbey, Cheryl Seals, and Alvin Lim. "Comparing Programmer Productivity in OpenACC and CUDA: An Empirical Investigation". In: *International Journal of Computer Science, Engineering and Applications (IJCSEA)* 6.5 (2016), pp. 1–15.

[96] Jin-Guo Liu, Yi-Hong Zhang, Yuan Wan, and Lei Wang. "Variational Quantum Eigensolver with Fewer Qubits". 2019. arXiv: 1902.02663 [quant-ph].

[97] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K Panda. "High Performance RDMA-Based MPI iMplementation over InfiniBand". In: *International Journal of Parallel Programming* 32.3 (2004), pp. 167–198.

[98] Justin Luitjens. *Faster Parallel Reductions on Kepler*. 2015.

[99] Dougal Maclaurin, David Duvenaud, and Ryan Adams. "Gradient-based Hyperparameter Optimization through Reversible Learning". In: *International Conference on Machine Learning*. 2015, pp. 2113–2122.

[100] Anton Malakhov. "Composable Multi-Threading for Python Libraries". In: *Proceedings of the Python in Science Conferences (SciPy)*. 2016.

[101] James Malcolm, Pavan Yalamanchili, Chris McClanahan, Vishwanath Venugopalakrishnan, Krunal Patel, and John Melonakos. "ArrayFire: a GPU Acceleration Platform". In: *Proceedings of the SPIE*. Vol. 8403. International Society for Optics and Photonics. 2012, 84030A–1.

[102] Nikolay Markovskiy. "Drop-In Acceleration of GNU Octave". 2014. URL: https://devblogs.nvidia.com/parallelforall/drop-in-acceleration-gnu-octave/.

[103] Bart van Merriënboer, Alexander B. Wiltschko, and Dan Moldovan. "Tangent: Automatic Differentiation Using Source Code Transformation in Python". 2017. arXiv: 1711.02712 [cs.MS].

[104] Duane Merrill. "CUB: A pattern of "collective" Software Design, Abstraction, and Reuse for Kernel-Level Programming". Presented at the GPU Technology Conference (GTC). 2015. URL: https://nvlabs.github.io/cub/.

[105] RV Mises and Hilda Pollaczek-Geiringer. "Praktische Verfahren der Gleichungsauflösung." In: *Journal of Applied Mathematics and Mechanics (ZAMM)* 9.1 (1929), pp. 152–164.

[106] MPI Forum. *MPI: A Message-Passing Interface Standard*. Tech. rep. 1994.

[107] Jameson Nash. *Inference Convergence Algorithm in Julia*. 2016. URL: https://juliacomputing.com/blog/2016/04/04/inference-convergence.html.

[108] Jameson Nash. *Inference Convergence Algorithm in Julia - Revisited*. 2017. URL: https://juliacomputing.com/blog/2017/05/15/inference-converage2.html.

[109] Jameson Nash. *Static and Ahead of Time Compiled Julia*. 2016. URL: https://juliacomputing.com/blog/2016/02/09/static-julia.html.

[110] Uwe Naumann. "Optimal Jacobian Accumulation is NP-Complete". In: *Mathematical Programming* 112.2 (2008), pp. 427–441.

[111] Uwe Naumann. "Optimized Jacobian Accumulation Techniques". In: *Problems in Modern Applied Mathematics*. WSES Press, 2000.

[112] Jaroslaw Nieplocha, Robert J Harrison, and Richard J Littlefield. "Global Arrays: A Portable Shared-Memory Programming Model for Distributed Memory Computers". In: *Proceedings of the Conference on Supercomputing*. IEEE Computer Society Press. 1994, pp. 340–349.

[113] NVIDIA. "cuBLAS: Dense Linear Algebra on GPUs". 2008. URL: https://developer.nvidia.com/cublas.

[114] NVIDIA Corporation. "NVIDIA Tesla V100 GPU Architecture". 2017. URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[115] Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. "SPIRAL in Scala: Towards the Systematic Construction of Generators for Performance Libraries". In: *ACM SIGPLAN Notices*. Vol. 49. 3. ACM. 2013, pp. 125–134.

[116] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. "Automatic Differentiation in PyTorch". Presented at the Autodiff Workshop at the Conference on Neural Information Processing Systems (NIPS). 2017.

[117] Barak A. Pearlmutter and Jeffrey Mark Siskind. "Reverse-mode AD in a Functional Framework: Lambda the Ultimate Backpropagator". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30.2 (2008), 7:1–7:36. ISSN: 0164-0925. DOI: 10.1145/1330017.1330018. URL: http://doi.acm.org/10.1145/1330017.1330018.

[118] Fernando Pérez and Brian E Granger. "IPython: A System for Interactive Scientific Computing". In: *Computing in Science & Engineering* 9.3 (2007).

[119] Guillem Pratx and Lei Xing. "GPU Computing in Medical Physics: A Review". In: *Medical Physics* 38.5 (2011), pp. 2685–2697.

[120] Christopher Rackauckas and Qing Nie. "Differentialequations.jl: A Performant and Feature-rich Ecosystem for Solving Differential Equations in Julia". In: *Journal of Open Research Software (JORS)* 5.1 (2017).

[121] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. "Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines". In: *ACM Transactions on Graphics* 31.4 (2012), 32:1–32:12.

[122] Min Ragan-Kelley, Fernando Pérez, Brian E Granger, Thomas Kluyver, Paul Ivanov, Jonathan Frederic, and Matthias Bussonnier. "The Jupyter/IPython Architecture: A Unified View of Computational Research, from Interactive Exploration to Communication and Publication." In: *AGU Fall Meeting Abstracts*. 2014.

[123] Jeffrey Regier, Kiran Pamnany, Keno Fischer, Andreas Noack, Maximilian Lam, Jarrett Revels, Steve Howard, Ryan Giordano, David Schlegel, Jon McAuliffe, and Prabhat Thomas Rollin. "Cataloging the Visible Universe through Bayesian Inference at Petascale". 2018. arXiv: 1801.10277 [cs.DC].

[124] Jarrett Revels. *Cassette.jl: A Tool For Dynamically Extending the Julia Compiler*. 2019. URL: https://github.com/jrevels/Cassette.jl.

[125] Jarrett Revels. *ReverseDiff: Reverse-Mode AD in Julia*. 2018. URL: https://github.com/JuliaDiff/ReverseDiff.jl.

[126] Jarrett Revels, Tim Besard, Valentin Churavy, Bjorn De Sutter, and Juan Pablo Vielma. "Dynamic Automatic Differentiation of GPU Broadcast Kernels". Presented at the Workshop on Systems for ML at the Conference on Neural Information Processing Systems (NeurIPS). 2018. arXiv: 1810.08297 [cs.MS].

[127] Jarrett Revels, Miles Lubin, and T. Papamarkou. "Forward-Mode Automatic Differentiation in Julia". 2016. arXiv: 1607.07892 [cs.MS]. URL: https://github.com/JuliaDiff/ForwardDiff.jl.

[128] Tiark Rompf and Martin Odersky. "Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs". In: *ACM SIGPLAN Notices*. Vol. 46. 2. ACM. 2010, pp. 127–136.

[129] Noam Ross. *Vectorization in R: Why?* 2014. URL: http://www.noamross.net/blog/2014/4/16/vectorization-in-r--why.html.

[130] Alex Rubinsteyn et al. "Parakeet: A Just-in-Time Parallel Accelerator for Python". In: *Proceedings on the USENIX Conference on Hot Topics in Parallelism (HotPar)*. 2012.

[131] Phil Ruffwind. *Reverse-mode Automatic Differentiation: A Tutorial*. 2016. URL: https://rufflewind.com/2016-12-30/reverse-mode-automatic-differentiation.

[132] Karl Rupp, Florian Rudolf, and Josef Weinbub. "ViennaCL: A High Level Linear Algebra Library for GPUs and Multi-Core CPUs". In: *International Workshop on GPUs and Scientific Applications (GPUScA)*. 2010, pp. 51–56.

[133] Nikolay Sakharnykh. *Beyond GPU Memory Limits with Unified Memory on Pascal*. 2016. URL: https://devblogs.nvidia.com/beyond-gpu-memory-limits-unified-memory-pascal/.

[134] David P. Sanders and Luis Benet. *IntervalArithmetic.jl: Compile your Julia Package*. 2019. URL: https://github.com/JuliaIntervals/IntervalArithmetic.jl.

[135] Souradip Sarkar, Turbo Majumder, Ananth Kalyanaraman, and Partha Pratim Pande. "Hardware Accelerators for Biocomputing: A Survey". In: *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2010, pp. 3789–3792.

[136] Jeffrey Sarnoff. *DoubleFloats.jl: Extended Precision Float and Complex Types*. 2019. URL: https://github.com/JuliaMath/DoubleFloats.jl.

[137] Amir Shaikhha, Andrew Fitzgibbon, Dimitrios Vytiniotis, Simon L. Peyton Jones, and Christoph Koch. "Efficient Differentiable Programming in a Functional Array-Processing Language". 2018. arXiv: 1806.02136 [cs.MS].

[138] Tim Short. *ExportWebAssembly.jl: Export Julia functions to WebAssembly and JavaScript*. 2018. URL: https://github.com/tshort/ExportWebAssembly.jl.

[139] N Solntseff and A Yezerski. "A Survey of Extensible Programming Languages". In: *Annual Review in Automatic Programming* 7 (1974).

[140] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. "Lift: a Functional Data-Parallel IR for High-Performance GPU Code Generation". In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2017, pp. 74–85.

[141] Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. "Delite: A Compiler Architecture for Performance-oriented Embedded Domain-specific Languages". In: *ACM Transactions on Embedded Computing Systems (TECS)* 13.4s (2014), p. 134.

[142] Walid Taha and Tim Sheard. "MetaML and Multi-stage Programming with Explicit Annotations". In: *Theoretical Computer Science* 248.1-2 (2000), pp. 211–242.

[143] Thomas Tan, Qi Li, Barry Boehm, Ye Yang, Mei He, and Ramin Moazeni. "Productivity tRends in Incremental and Iterative Software Development". In: *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE. 2009, pp. 1–10.

[144] The PyTorch Team. *Torch Script Documentation*. 2018. URL: https://pytorch.org/docs/master/jit.html.

[145] TIOBE Company. *TIOBE Programming Community Index*. Feb. 2018. URL: https://www.tiobe.com/tiobe-index/.

[146] Qiang Tu et al. "Evolution in Open Source Software: A Case Study". In: *Proceedings of the International Conference on Software Maintenance (ICSME)*. IEEE. 2000, pp. 131–142.

[147] Sujatha R Upadhyaya. "Parallel Approaches to Machine Learning: A Comprehensive Survey". In: *Journal of Parallel and Distributed Computing* 73.3 (2013), pp. 284–292.

[148] Stefan Van Der Walt, Chris Colbert, and Gael Varoquaux. "The NumPy Array: A Structure for Efficient Numerical Computation". In: *Computing in Science & Engineering* 13.2 (2011), p. 22.

[149] Henri Verroken. "Contextual Language Abstractions for Low-Level GPGPU". MA thesis. Ghent University, 2018.

[150] Fei Wang, Xilun Wu, Gregory Essertel, James Decker, and Tiark Rompf. "Demystifying Differentiable Programming: Shift/Reset the Penultimate Backpropagator". 2018. arXiv: 1803.10228 [cs.LG].

[151] Jagath Weerasinghe, Francois Abel, Christoph Hagleitner, and Andreas Herkersdorf. "Enabling FPGAs in Hyperscale Data Centers". In: *Proceedings of the International Conference on Ubiquitous Intelligence and Computing, Autonomic and Trusted Computing, Scalable Computing and Communications*. IEEE. 2015, pp. 1078–1086.

[152] Richard Wei. "First-Class Automatic Differentiation in Swift: A Manifesto". 2018. URL: https://gist.github.com/rxwei/30ba75ce092ab3b0dce4bde1fc2c9f1d.

[153] Richard Wei, Lane Schwartz, and Vikram Adve. "A Modern Compiler Framework for Neural Network DSLs". Presented at the Workshop on ML Systems at the Conference on Neural Information Processing Systems (NIPS). 2017.

[154] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. "OpenACC: First Experiences with Real-world Applications". In: *Proceedings of the European Conference on Parallel Processing (Euro-Par)*. Springer-Verlag, 2012, pp. 859–870.

[155] Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune, Rob Springer, Xuetian Weng, and Robert Hundt. "gpucc: An Open-Source GPGPU Compiler". In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. ACM. 2016, pp. 105–116.

[156] Andy B Yoo, Morris A Jette, and Mark Grondona. "Slurm: Simple Linux Utility for Resource Management". In: *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*. Springer. 2003, pp. 44–60.

[157] Deniz Yuret. “Knet: Beginning Deep Learning with 100 Lines of Julia”. In: *Machine Learning Systems Workshop at NIPS*. Vol. 2016. 2016, p. 5.

[158] Francesco Zappa Nardelli, Julia Belyakova, Artem Pelenitsyn, Benjamin Chung, Jeff Bezanson, and Jan Vitek. “Julia Subtyping: A Rational Reconstruction”. In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), p. 113.

[159] Daniel Zingaro. “Modern Extensible Languages”. In: *SQRL Report* 47 (2007), p. 16.