

Effective Extensible Programming: Unleashing Julia on GPUs

Tim Besard, Christophe Foket and Bjorn De Sutter, *Member, IEEE*

Abstract—GPUs and other accelerators are popular devices for accelerating compute-intensive, parallelizable applications. However, programming these devices is a difficult task. Writing efficient device code is challenging, and is typically done in a low-level programming language. High-level languages are rarely supported, or do not integrate with the rest of the high-level language ecosystem. To overcome this, we propose compiler infrastructure to efficiently add support for new hardware or environments to an existing programming language. We evaluate our approach by adding support for NVIDIA GPUs to the Julia programming language. By integrating with the existing compiler, we significantly lower the cost to implement and maintain the new compiler, and facilitate reuse of existing application code. Moreover, use of the high-level Julia programming language enables new and dynamic approaches for GPU programming. This greatly improves programmer productivity, while maintaining application performance similar to that of the official NVIDIA CUDA toolkit.

Index Terms—Graphics processors, very high-level languages, code generation, retargetable compilers



1 INTRODUCTION

To satisfy ever higher computational demands, hardware vendors and software developers look at accelerators, specialized processors that are optimized for specific, typically parallel workloads, and perform much better at them than general-purpose processors [1], [2], [3], [4], [5]. Multiple hardware vendors are working on such accelerators and release many new devices every year. These rapid developments make it difficult for developers to keep up and gain sufficient experience programming the devices. This is exacerbated by the fact that many vendors only provide low-level toolchains, such as CUDA or OpenCL, which offer full control to reach peak performance at the cost of developer productivity [6].

To improve developer productivity, programmers commonly use high-level programming languages. However, these languages often rely on techniques and functionality that are hard to implement or even incompatible with execution on typical accelerators, such as interpretation, tracing Just-in-Time (JIT) compilation, or reliance on a managed runtime library. To remedy this, implementations of high-level languages for accelerators generally target a derived version of the language, such as a restricted subset or an embedded Domain Specific Language (DSL), in which incompatible features have been redefined or adjusted.

Modern extensible languages offer the means to realize such programming language derivatives [7]. For example, Lisp-like languages feature powerful macros for processing syntax, Python’s decorators make it possible to change the behavior of functions and methods, and the Julia programming language supports introspection of each of its Intermediate Representations (IRs). However, these facilities do not encourage reuse of existing compiler functionality. Most derived languages use a custom compiler, which simplifies

the implementation but hinders long-term maintainability when the host language gains features or changes semantics. It also forces users to learn and deal with the inevitable divergence between individual language implementations.

This paper presents a vision in which the high-level language compiler exposes interfaces to alter the compilation process (Section 2). Implementations of the language for other platforms can use these interfaces together with other extensible programming patterns to ensure that source code is compiled to compatible and efficient accelerator machine code. To demonstrate the power of this approach, we have added such interfaces to the reference compiler of the Julia language (Section 4), and used it to add support for NVIDIA GPUs (Section 5). We show that the resulting toolchain makes it possible to write generic and high-level GPU code, while performing similar to low-level CUDA C (Section 6). All code implementing this framework is available as open-source software on GitHub, and can be easily installed using the Julia package manager. Our contributions are as follows:

- We introduce interfaces for altering the compilation process, and implement them in the Julia compiler.
- We present an implementation of the Julia language for NVIDIA GPUs, using the introduced interfaces.
- We analyze the performance of benchmarks from the Rodinia suite ported to Julia. We show that kernels programmed in Julia perform similar to CUDA C code compiled with NVIDIA’s reference compiler.
- We demonstrate high-level programming with this toolchain, and show that Julia GPU code can be highly generic and flexible, without sacrificing performance.

2 VISION

Our proposed solution to the difficulty in integrating high-level languages and accelerators is a set of interfaces to the high-level language’s general purpose compiler, that provide fine-grained access to the different IRs and to the processes that generate and optimize those IRs. With these

• T. Besard, C. Foket and B. De Sutter are with the Department of Electronics and Information Systems, Ghent University, Belgium.
{tim.besard, christophe.foket}@ugent.be
Corresponding author: bjorn.desutter@ugent.be

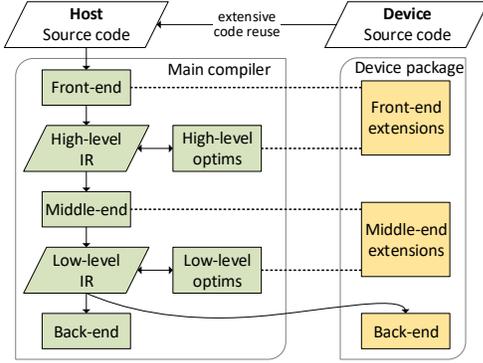


Figure 1. Abstract overview of the proposed toolchain.

interfaces, developers can influence the existing language implementation and, e.g., improve compatibility with new hardware or run-time environments without the need for a custom compiler or an embedded language subset.

Figure 1 shows an overview of the proposed toolchain. 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 their code 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, whereas current implementations of high-level languages for accelerators often reimplement large parts of the compiler. For example, Numba is a JIT compiler for Python, building on the CPython reference language implementation. As Figure 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 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 [8], and needs to be updated when changes are made to the language implementation. The semantics of code also differ slightly depending on whether it is interpreted by CPython or compiled with Numba [8], further impeding compatibility with existing Python code.

Our proposed compiler interfaces allow to share functionality between an existing language implementation and external derivatives, avoiding needless reimplementations of functionality by reconfiguring the existing compiler to generate code compatible with the platform at hand. This not only facilitates external language implementations. It also improves compatibility with existing code as it avoids inevitable differences between individual compiler implementations.

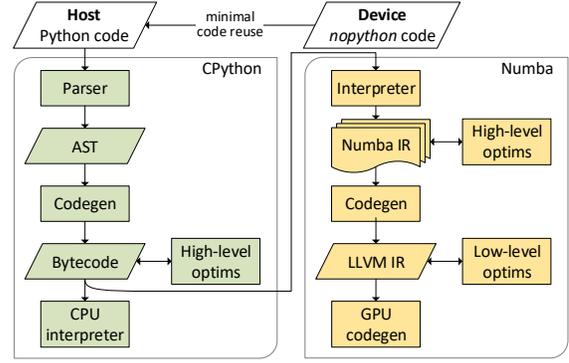


Figure 2. CPython and Numba compilation processes for host and device.

In some cases, even more reuse of existing infrastructure than suggested in Figure 1 is possible. When the main compiler’s back-end also targets accelerators, there is no need to reimplement a back-end in the device package. Instead, that existing back-end compiler can then be used for host and device code. Even if this is not the case, 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 1 are generally applicable. Their actual instantiation, however, will be specific to the host language and accelerator at hand. We expect further research into such interfaces to generalize the design and improve reusability across languages and accelerators. For now, we will design the interfaces around a single language and accelerator platform.

For this work, we chose to target Graphics Processing Units (GPUs), massively parallel accelerators that require specialized code, yet are usable for many kinds of applications. Specifically, we focus on Compute Unified Device Architecture (CUDA) GPUs, because of their mature toolchain and hardware availability. We target this hardware from the Julia programming language, a high-level technical computing language built on the Low-Level Virtual Machine (LLVM) compiler framework. As Section 3.2 will explain, Julia is a good fit for accelerator programming, while offering flexible tools to extend the language, e.g., for the purpose of targeting new hardware. Given its use of LLVM, and LLVM’s capabilities to target both CPUs and CUDA GPUs, we do not need to reimplement a device back-end ourselves.

3 BACKGROUND

3.1 GPU Accelerators

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 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.

Vendor-supported development environments for 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 [6]. Furthermore, due to a lack of abstractions, these implementations are often hardware-specific, or perform significantly worse on different hardware [9]. Libraries like CUB [10] or Thrust [11] 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.

Alternatively, developers can use optimized host libraries that are called from the host processor and not directly from the device. Hardware vendors provide libraries for popular interfaces like BLAS [12] and LAPACK [13]. There also exist third-party libraries like ArrayFire [14] and ViennaCL [15] that abstract over devices and platforms. These libraries typically export a C Application Programming Interface (API), which eases their use outside of the vendor-supplied development environment. For example, the CUDA BLAS library cuBLAS [16] can be used from Python [17], Julia [18], Octave [19], 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 offers fine-grained control over compilation and execution, while the use of a high-level language and its abstraction capabilities improve programmer productivity. However, existing high-level languages implementations for accelerators do not integrate well with the rest of the language. Embedded DSL such as PyGPU or Copperhead [20], [21] come with a learning curve, and programmers have to adapt their code. Continuum Analytics’ Numba [8] reimplements support for a subset of the Python language that is appropriately called `nopython` because it does not support many of the high-level features of Python because these features do not map well onto GPUs, while duplicating compiler functionality from the CPython reference implementation as shown in Figure 2. Our proposed interfaces serve to avoid this duplication, and integrate with the existing language implementation for the purpose of improved code compatibility and more effective compiler implementation.

3.2 Julia Programming Language

Julia is a high-level, high-performance dynamic programming language for technical computing [22]. It features

Listing 1

Single-dispatch polymorphism and branches that leads to unstable functions, returning differently-typed objects based on run-time values.

```

1 function intersect(a::Rect, b) # returns Rect or Line
2   if isa(b, Rect) return c::Rect
3   else if isa(b, Line) return c::Line
4   end
5 end
6 function intersect(a::Line, b) # returns Rect or Line
7   return c
8 end

```

Listing 2

Functionality of Listing 1 expressed through multiple dispatch.

```

1 function intersect(a::Rect, b::Rect) # returns Rect
2   return c::Rect
3 end
4 function intersect(a::Rect, b::Line) # returns Line
5   return c::Line
6 end

```

a type system with parametric polymorphism, multiple dispatch, metaprogramming capabilities, and other high-level features [23]. 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 [24], [25].

Julia’s competitive performance originates from clever language design that avoids the typical compilation and execution uncertainties associated with dynamic languages [26]. 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. 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 run-time 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 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 “this” type 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.

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, specula-

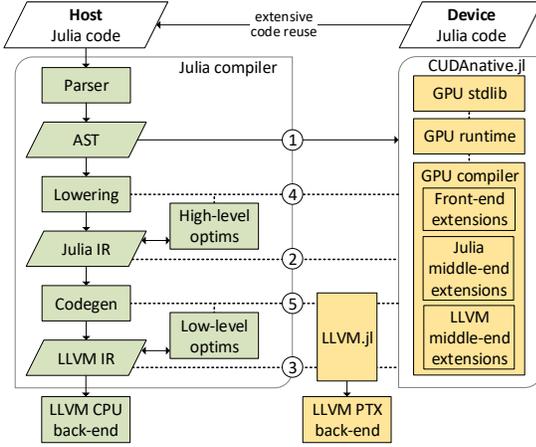


Figure 3. Overview of the compilation process for Julia GPU code with CUDA native.jl by means of compiler extension interfaces. Dashed arrows indicate generic interactions; solid arrows represent the flow of code.

tive 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 [27]. The Julia compiler targets this framework by emitting LLVM IR as the low-level IR from Figure 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 machine code. The left part of Figure 3 shows this existing Julia compilation tool flow. In the remainder of this paper, 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.

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 [24], [25]. The managed runtime library is only required for dynamic code that might trigger compilation, and certain language features such as garbage collection and stack unwinding.

Coincidentally, this design also makes the language well-suited for accelerator programming. Such hardware often features a different architecture and Instruction Set Architecture (ISA), 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 boils down to self-contained, static IR, a prerequisite for many accelerator programming models.

In addition, Julia features powerful metaprogramming and reflection capabilities, as shown in Table 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*

Table 1
Existing metaprogramming interfaces in Julia to access compiler IRs.

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

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. As we will discuss in the next section, however, their support does not yet suffice for targeting accelerators like GPUs.

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. It suffices for the main compiler because (i) metaprogramming and reflection do not require interfaces at the LLVM IR level, (ii) *llvmcall* is currently only used 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 IR builder interfaces.

However, as we will discuss in the next section, these string-based interfaces to the lower-level IRs do not suffice for targeting accelerators from within device packages.

4 EFFECTIVE EXTENSIBLE PROGRAMMING

As discussed in Section 2, we propose to integrate high-level programming languages with accelerator hardware by means of extension interfaces to the existing compiler that was, and will continue to be, developed by and large to target general-purpose hardware. The existing interfaces to manipulate the different IRs as discussed in the previous section provide a good starting point, but they do not yet suffice.

First, although they make it possible to improve compatibility with accelerators by manipulating general purpose IR or generating compatible IR from scratch, they fall short in reusing and repurposing the main compiler’s IR-generating components. Section 4.1 proposes our solution to make the compiler generate accelerator-optimized IR in the first place.

Secondly, the string-based interfaces to the lower-level IRs do not let the device package reuse main compiler functionality to generate and inject low-level IR code. As targeting accelerators such as GPUs requires more than injecting predetermined code snippets, this lack of reuse is problematic. Section 4.2 presents a solution to this issue.

Table 2
Additional interfaces for controlling code generation processes.

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

4.1 Front-end IR Interfaces

As a starting point to generate accelerator code, the language should offer access to the different representations of source code in the compiler, such as syntax trees, IRs, and machine code. This makes it possible to implement functionality that cannot be expressed in the source language by manually emitting intermediate 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. In the case of the Julia programming language, there already exist several metaprogramming interfaces that provide access to those intermediate forms of code, as shown in Table 1.

For external language implementations, having access to the code generated for each IR level is insufficient. In addition, access to the IR code should be augmented with access to the processes that generate that code. When compiling code for an environment that, e.g., does not support the Julia runtime library, the compiler needs to avoid calls to it. A typical case is that of exceptions, which rely on the runtime for stack unwinding and error reporting. In the main Julia compiler, these 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 altering the code generation process, the compiler 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 of the main compiler and duplicating functionality from it.

To overcome this problem and improve the reusability of the compiler, we added the four interfaces from Table 2 that offer additional control over code generation processes. More specifically, both the lowering of Abstract Syntax Trees (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 Section 5 uses these interfaces to replace or customize code generation functionality 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.

4.2 Back-end IR Interfaces

The codegen step in the main compiler translates (i.e., lowers) Julia IR constructs into LLVM IR. The C++ part of the codegen implementation directly invokes LLVM IR builder interfaces to do so; the 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; the C++ approach is not productive enough for the package developer (likely an expert in Julia and in his targeted accelerators, but not necessarily in C++ APIs). We hence strive for providing 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 Section 5 only requires about 1500 lines of code (LOC).

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) independent from 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 sometimes conservative upgrade policies by system administrators.

To facilitate interactions with LLVM, we have created the `LLVM.jl` package, which is available at <https://github.com/maleadt/LLVM.jl>. It provides a high-level wrapper to the LLVM C API, using Julia’s powerful Foreign Function Interface (FFI) to interact efficiently with the underlying libraries. The package can be used to inspect, modify or emit LLVM IR. It greatly improves the usability of the extension interfaces that operate at the LLVM IR level. In addition, the package enables reuse of back-end compiler functionality that are 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.

Listing 3 shows an example use of `LLVM.jl` for implementing the necessary lowering from Julia IR to LLVM IR. Specifically, this shows how to implement a custom function for loading values from a pointer using the `LLVM.jl` interfaces. In Julia IR, accessing, e.g., an element in an array, is modeled with 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 codegen processing step 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

Listing 3
Using LLVM.jl to implement functions that generate their own IR.

```

1 # loading a value from a pointer
2 @generated function load(p::Ptr{T}) where {T}
3     eltyp = LLVM.convert(LLVM.Type, T)
4
5 # create a LLVM module and function
6 mod = LLVM.Module("llvmcall")
7 param_typs = [LLVM.PointerType(eltyp)]
8 ft = LLVM.FunctionType(eltyp, param_typs)
9 f = LLVM.Function(mod, "load", ft)
10 # generate IR
11 LLVM.Builder() do builder
12     bb = LLVM.BasicBlock(f, "entry")
13     LLVM.position!(builder, bb)
14     ptr = LLVM.parameters(f)[1]
15     val = LLVM.load!(builder, ptr) # the actual load
16     LLVM.ret!(builder, val)
17 end
18 # inject the IR and call it
19 return :( llvmcall($f, $T, Tuple{Ptr{$T}}, p) )
20 end
21
22 @test load(pointer([42])) == 42

```

or more intrinsics in the main compiler, and writing the necessary lowering support in C++ using LLVM APIs. This is cumbersome, inflexible, and unproductive.

By contrast, the code in Listing 3 shows how to load a value from a pointer with Julia metaprogramming and the LLVM.jl package.¹ It is implemented using a generator function, declared with `@generated` on line 2, which builds the expressions that should be executed at run time. Generator functions are invoked during type-inference, for every combination of argument types the function is invoked with. In this case, this function generates LLVM IR and returns an `llvmcall` expression that injects the code into the compiler, effectively returning the IR that will have to be executed in the application. Note that this is much stronger than using macros: on line 3, 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 runtime element type of the pointer. The generated code hence depends on the inferred types, and can be customized and optimized for it at each invocation of `load` in the Julia IR. In the next section, we will discuss how this can be exploited to generate memory accesses optimized for the different types of memories in a GPU memory hierarchy.

Without the LLVM.jl interface, the `load` function body would have been full of string manipulations, which would have been a nightmare in terms of code readability. Moreover, it would have contained cases for every supported pointer type, and the optimization for, e.g., different types of memories, would be hard or impossible.

5 CUDA LANGUAGE IMPLEMENTATION

Eating our own dog food, we used the infrastructure from Section 4 to develop a GPU implementation of Julia that targets NVIDIA hardware via the CUDA toolkit. This implementation is an instantiation of the device package in Figure 1. It is distributed as a regular Julia package named

1. To avoid uninteresting clutter in our illustration of LLVM.jl, we show a simplified `load` function instead of the full `unsafe_load`.

`CUDANative.jl`, available at <https://github.com/JuliaGPU/CUDANative.jl>, and does not require any modifications to the underlying Julia compiler. It supports a subset of the Julia language, but that subset has proven extensive enough to implement real-life GPU applications and build high-level abstractions. The device package actually consists of three major components, as shown on the right of Figure 3: a standard library of GPU-specific functionality, a compiler to generate GPU machine code from Julia sources, and a runtime 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.

5.1 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.

Whereas many languages would implement these 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 isn’t expressive enough, intrinsics can be avoided by generating lower-level code directly using the metaprogramming interfaces from Table 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 `CUDANative.jl` relies heavily on this type of programming, with help from the LLVM API wrapper from Section 4.2 to facilitate interactions with the LLVM IR.

Julia’s expressiveness and metaprogramming functionality make it possible for most of the Julia 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 compatibility or performance of existing language features, as the next section illustrates.

5.1.1 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 Parallel Thread Execution (PTX) 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

Listing 4
Optimized GPU pointers in `CUDAnative.jl`, building on Listing 3.

```

1 # custom pointer with address-space information
2 struct DevPtr{T,AS}
3   ptr::Ptr{T}
4 end
5 # loading an indexed value from a pointer
6 @generated function unsafe_load(p::DevPtr{T,AS},
7                               i::Int=1)
8     where {T,AS}
9     eltyp = LLVM.convert(LLVM.Type, T)
10    # create a LLVM module and function
11    ...
12    # generate IR
13    LLVM.Builder() do builder
14        ptr = LLVM.gep!(builder, LLVM.parameters(f)[1],
15                       [parameters(f)[2]])
16        devptr_typ = LLVM.PointerType(eltyp, AS)
17        devptr = LLVM.addrspacecast!(builder, ptr, devptr_typ)
18        val = LLVM.load!(builder, devptr)
19    end
20    # inject the IR and call it
21    ...
22 end

```

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 [28] fall short when memory allocation sites are invisible. Pointers passed to a kernel as arguments, which happens often 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 `Ptr` objects: 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 4 shows how `CUDAnative.jl` provides a custom `DevPtr` type representing a pointer with address-space information. By implementing the expected method interface, which includes the `unsafe_load` method defined on line 6, `DevPtr` objects can be used in place of `Ptr` objects. This then yields specialized memory operations that perform better.

The implementation of `unsafe_load` in Listing 4 uses the metaprogramming techniques explained in Section 4.2. A generator function builds specialized LLVM IR and injects it back in the compiler, with the relevant address-space-specific load on lines 16 and 17. 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.

5.1.2 NVIDIA Device Library

Another important source of low-level GPU operations in `CUDAnative.jl` 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 manipula-

tion 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`.

5.2 GPU Compiler

Together with the main Julia compiler, the `CUDAnative.jl` infrastructure of Figure 3 instantiates the design from Figure 1, with the unaltered Julia IR and the unaltered LLVM IR 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.

The main interface for calling functions on a GPU resembles a call to an ordinary Julia function: `@cuda (config...) function(args...)`, where the `config` tuple indicates the launch configuration similar to the triple angle bracket syntax in CUDA C. 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 1, the new interfaces from Table 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 GPU extensions in the `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 4.2. This use of an external GPU back-end compiler rather than one embedded in the device package diverges from the design in Figure 1, as was already hinted in Section 2. For its CPU back end, the Julia compiler already relies on CPU LLVM 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 [28], voids the need for including a third-party GPU compiler or a reimplementa-

thereof in the device package. Without putting any burden on system administrators or users to install additional packages or tools, we can simply reuse the LLVM PTX back end.

Before generating machine code, LLVM optimization passes extensively optimize the LLVM IR. As part of that process, Julia and CUDAnative.jl lower specific constructs and optimize certain patterns. 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, while maintaining the aforementioned semantics. In the case of GPU code, this means that not the aggregate argument itself, but only a pointer to the argument will be stored in the designated state space (see Section 5.1.1). 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.

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. We implement this change at the LLVM IR level by generating a wrapper function that takes values as arguments, 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. Finally, the CUDAnative.jl runtime passes all immutable arguments by value instead of by reference. This yields a speedup of up to 20% on memory-intensive Rodinia benchmarks.

This optimization provides an excellent example of the code reuse enabled by our tool flow design and the added extension interfaces. Due to that reuse, the code to build the wrapper function and perform the necessary optimizations to inline the code requires less than 100 lines of Julia code.

5.3 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 5 uses the runtime API to initialize and upload memory, launch the kernel, and fetch results. The syntax for calling kernels (line 21) hides much of the underlying complexity: setting-up a parameter buffer, initializing the execution configuration, acquiring a reference to the compiled kernel code, etc.

Listing 5
Vector addition in CUDA C, using the CUDA run-time API.

```

1 #define cudaCall(err) // check return code for error
2 #define frand() (float)rand() / (float)(RAND_MAX)
3 __global__ void vadd(const float *a, const float *b,
4                    float *c) {
5     int i = blockIdx.x * blockDim.x + threadIdx.x;
6     c[i] = a[i] + b[i];
7 }
8
9 const int len = 100;
10
11 int main() {
12     float *a = new float[len], *b = new float[len];
13     for (int i = 0; i < len; i++) {
14         a[i] = frand(); b[i] = frand();
15     }
16     float *d_a, *d_b, *d_c;
17     cudaCall(cudaMalloc(&d_a, len * sizeof(float)));
18     cudaCall(cudaMemcpy(d_a, a, len * sizeof(float),
19                        cudaMemcpyHostToDevice));
20     cudaCall(cudaMalloc(&d_b, len * sizeof(float)));
21     cudaCall(cudaMemcpy(d_b, b, len * sizeof(float),
22                        cudaMemcpyHostToDevice));
23     cudaCall(cudaMalloc(&d_c, len * sizeof(float)));
24     vadd<<<1, len>>>(d_a, d_b, d_c);
25
26     float *c = new float[len];
27     cudaCall(cudaMemcpy(c, d_c, len * sizeof(float),
28                        cudaMemcpyDeviceToHost));
29     cudaCall(cudaFree(d_c));
30     cudaCall(cudaFree(d_b));
31     cudaCall(cudaFree(d_a));
32     return 0;
33 }
```

Listing 6
Vector addition in Julia using CUDAdrv.jl and CUDAnative.jl.

```

1 function vadd(a, b, c)
2     i = (blockIdx().x-1) * blockDim().x + threadIdx().x
3     c[i] = a[i] + b[i]
4     return
5 end
6
7 len = 100
8 a = rand(Float32, len)
9 b = rand(Float32, len)
10
11 d_a = CUDAdrv.Array(a)
12 d_b = CUDAdrv.Array(b)
13 d_c = similar(d_a)
14
15 @cuda (1,len) vadd(d_a, d_b, d_c)
16 c = Base.Array(d_c)
```

To improve the usability of the CUDA API from Julia, we have created *CUDAdrv.jl*, which is available at <https://github.com/JuliaGPU/CUDAdrv.jl>. This is a package 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 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 6 shows a Julia implementation of the vector addition from Listing 5, 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 the custom `Array` type, API error codes are automatically checked and converted to descriptive exceptions, GPU memory is automatically freed by the Julia garbage collector, etc.

Listing 7
Lowered code generated from the `@cuda` invocation in Listing 6.

```

1  # results of compile-time computations
2  ## at parse time
3  grid = (1,1,1)
4  block = (len,1,1)
5  shmem = 0
6  stream = CuDefaultStream()
7  func = vadd
8  args = (d_a, d_b, d_c)
9  ## during type inference
10 types = (CuDeviceArray{Float32,2,AS.Global},
           CuDeviceArray{Float32,2,AS.Global},
           CuDeviceArray{Float32,2,AS.Global})
11 partial_key = hash(func, types)
12 # determine the run-time environment
13 age = method_age(func, $types)
14 ctx = CuCurrentContext()
15 key = hash(partial_key, age, ctx)
16 # cached compilation
17 kernel = get!(kernel_cache, key) do
18     dev = device(ctx)
19     cufunction(dev, func, types)
20 end
21
22 cudacall(kernel, types, args,
           grid, block, shmem, stream)

```

5.4 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.

Whereas calling 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` function call is first evaluated. At that point, the invoked kernel and its functions are 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 6 is shown in Listing 7. Lines 3 to 11 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 15 in Listing 6 passes `CUDADrv.Arrays`, 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 22.

In addition to recompiling specialized and optimized kernels for changing run-time types, the `CUDANative.jl` runtime 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 is already supported in 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 17 of Listing 7. Upon a cache miss, the kernel is compiled and added to the cache. Finally, control is handed over to `CUDADrv.jl` on line 22 where `cudacall` converts the arguments and launches the kernel.

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, 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 6.2.2.

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 [29]. The environment centers 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 support by means of 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.

6 EVALUATION

To demonstrate the full capabilities of our framework, we present a two-fold evaluation: First, we experimentally evaluate the performance that can be obtained when programming GPUs in Julia, using mostly standard benchmarks.

Secondly, we illustrate the high-level programming capabilities of the infrastructure, for which we have implemented kernels using high-level Julia constructs. We demonstrate how these constructs can be applied to GPU programming without sacrificing performance.

6.1 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 v2s CPUs, and 64GB of DDR3 ECC memory.

6.2 Performance

To assess the performance of GPU kernels written in Julia, we have ported CUDA C benchmarks from the Rodinia suite that is commonly used to evaluate CUDA kernel compilers to `CUDAnative.jl` [30]. Our ports are available at <https://github.com/JuliaParallel/rodinia/>. 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: Pointers to arrays are represented by objects (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), taking also 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.

6.2.1 Methodology

Table 3 presents some relevant features of the benchmarks, as well as the experimental results. All execution times are reported in milliseconds, and show the mean value with error margins determined by propagating the standard deviation across operations [31]. 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` library to extend the profile with CPU timings. Benchmark inputs are the defaults parameters from Rodinia 3.1.

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

6.2.2 JIT Compilation Times

The latter remark is particularly the case for the sub-second execution times (column e) spent on Julia JIT compilation. The large fractions of the total runtime spent on JIT compilation is more a side-effect of the benchmarks' short-running kernels than of the excessive compilation times. As 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, their sub-second JIT compilation times will be amortized (almost) completely in real-world applications. Even within our range of benchmarks running anywhere in between 1ms and 6827 ms, this amortization can already be observed in the decreasing numbers in column (e/c). 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 precompiled, such as the code in all used Julia packages², or interpreted according to heuristics in the Julia compiler.

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 Rodinia kernels from Table 3, 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.

6.2.3 Kernel Performance

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

2. With the official package from <https://github.com/JuliaLang/PackageCompiler.jl>

benchmark	kernels	CUDA C								JULIA								JULIA/CUDA C	
		total		GPU kernel				total		GPU kernels				JIT compilation		total - JIT		GPU kernels	total - JIT
		LOC	(a) run time	LOC	PTX	(b) run time	(b/a)	LOC	(c) run time	LOC	PTX	(d) run time	(d/c)	(e) run time	(e/c)	(f = c-e) run time	(f/c)	(d/b)	(f/a)
lud	3	263	1.1 ±6.2%	126	1588	0.42 ±0.7%	39%	202	478.9 ±1.0%	110	2586	0.48 ±0.4%	0.10%	477.5 ±1.0%	100%	1.4 ±5.4%	0.29%	1.14	1.26
particlefilter	4	611	60.1 ±0.5%	152	2063	38.60 ±0.2%	64%	409	522.5 ±1.8%	123	1304	34.30 ±1.3%	6.56%	480.5 ±1.8%	92%	42.0 ±2.3%	8.04%	0.89	0.70
backprop	2	631	40.6 ±4.7%	56	230	0.19 ±0.3%	0%	317	105.8 ±3.7%	54	257	0.17 ±0.8%	0.16%	62.3 ±1.8%	59%	43.5 ±6.4%	41.14%	0.88	1.07
nw	2	340	50.9 ±15.2%	118	1341	1.89 ±0.2%	4%	255	214.7 ±3.0%	110	633	1.94 ±0.2%	0.90%	169.1 ±1.2%	79%	45.6 ±9.8%	21.26%	1.03	0.90
leukocyte	3	1665	181.8 ±1.1%	384	1474	86.09 ±0.1%	47%	856	725.2 ±2.9%	275	1344	67.92 ±0.1%	9.37%	434.0 ±1.7%	60%	291.2 ±4.6%	40.16%	0.79	1.60
pathfinder	1	166	196.9 ±2.3%	49	163	0.25 ±0.5%	0%	140	237.2 ±2.8%	52	152	0.25 ±0.8%	0.10%	38.7 ±2.1%	16%	198.6 ±3.0%	83.70%	1.01	1.01
hotspot	1	265	137.4 ±1.3%	91	237	0.11 ±0.5%	0%	228	184.7 ±1.7%	87	247	0.11 ±0.3%	0.06%	65.4 ±1.7%	35%	119.3 ±1.7%	64.61%	1.03	0.87
nn	1	270	253.6 ±1.6%	11	53	0.03 ±0.7%	0%	148	505.7 ±3.7%	11	61	0.03 ±1.6%	0.01%	100.1 ±1.8%	20%	405.6 ±4.2%	80.21%	1.00	1.60
bfs	2	184	1278.1 ±0.9%	33	143	6.49 ±0.2%	1%	135	1688.4 ±2.2%	28	161	6.03 ±0.1%	0.36%	40.4 ±1.5%	2%	1648.0 ±2.3%	97.61%	0.93	1.29
streamcluster	1	952	6457.2 ±5.1%	28	202	510.61 ±0.5%	8%	647	6827.6 ±3.4%	30	162	501.95 ±0.0%	7.35%	31.2 ±1.8%	0%	6796.4 ±3.4%	99.54%	0.98	1.05
average			±3.9%			±0.4%	16%		±2.6%			±0.6%		±1.6%		±4.3%		0.96	1.10

Table 3
Rodinia benchmark features and performance of CUDAnative.jl vs CUDA C.

speedup of 1% as achieved by `gpucc` on a wider range of Rodinia benchmarks [28]. `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 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, whereas C language compilers only use 32 bits to represent `int` values. The large speedups as seen with `particlefilter`, `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.

6.2.4 Application Performance

For the sake of completeness, the last column reports the wall clock execution times excluding JIT compilation time. 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 CUDA 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 CUDA C. 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 string objects during parsing. This could be solved by using the high-performance `TextParse.jl` package, 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 [24], [25].

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 CUDAnative.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

Table 4
GPU and CPU execution times for an empty kernel.

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

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.

The comparable host-GPU interaction performance between CUDA C and Julia 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 5.4 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 4 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. With 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.

6.3 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

Listing 8
Using CuArrays.jl to perform a fused broadcast on the GPU.

```
1 X = CuArray(rand(42))
2 f(x) = 3x^2 + 5x + 2
3 Y = f.(2 .* X.^2 .+ 6 .* X.^3 .- sqrt.(X))
```

Listing 9
Equivalent CUDA C kernel code for the fused broadcast in Listing 8. Host code has been omitted for brevity, and would be similar to Listing 5.

```
1 __device__ double f(double x) {
2     return 3*pow(x, 2) + 5*x + 2;
3 }
4 __global__ void kernel(double *X, double *Y, int N) {
5     int i = blockIdx.x * blockDim.x + threadIdx.x;
6     if (i < N) Y[i] = f(2*X[i]+6*pow(X[i],3)-sqrt(X[i]));
7 }
```

express operations on multidimensional data compactly [32]. 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 in Table 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%.

With this approach of providing high-level wrappers for APIs, developer effort is significantly lowered, but familiarity with GPUs and their programming model is still a requirement. The next section will discuss a higher-level approach, with abstractions that obviate GPU experience.

6.4 High-level GPU Programming

To demonstrate the high-level programming potential of this infrastructure, we use *CuArrays.jl* [33]. This package defines an array type for data that lives on the GPU, but exposes host-level operations that are implemented using the infrastructure from this paper to execute on the device. For example, constructing a *CuArray* object will allocate data on the GPU using *CUDAdrv.jl*, adding two such host objects together will queue an addition kernel on the GPU using *CUDAnative.jl*, etc. Certain other functionality is implemented using optimized host libraries like *cuBLAS* or *cuDNN*, but that is not relevant to the work in this paper.

The example from Listing 8 shows how to load the *CuArrays.jl* package, generate input data and upload it to the GPU on line 1, defining an auxiliary function for the sake of this example on line 2, and finally a series of element-wise operations including a call to the newly defined function on line 3. These operations, prefixed by a dot to indicate the element-wise application, are syntactically fused together into a single broadcast operation [32]: $Y = \text{broadcast}(x \rightarrow f(2x^2 + 6x^3 - \text{sqrt}(x)), X)$ where the first argument is a lambda containing the fused operations from line 3. The implementation of *broadcast* in *CuArrays.jl* then compiles this function using *CUDAnative.jl*, inlining calls to both the lambda and underlying function *f*.

Listing 10
Reducing an array of custom objects on the GPU.

```
1 # illustrational type that implements addition
2 struct Point{T}
3     x::T
4     y::T
5 end
6 +(a::Point{T}, b::Point{T}) where {T} =
7     Point(a.x+b.x, a.y+b.y)
8 data = [Point(rand(Int64)%100, rand(Int64)%100)
9         for _ in 1:42]
9 X = CuArray(data)
10 Y = reduce(+, #neutral element=# Point(0,0), X)
```

Conceptually, broadcasting a function over GPU arrays like *CuArray* is straightforward: Each thread processes one element, the grid is constructed relatively naively, there are no cross-thread dependencies, etc. However, the actual implementation relies on several advanced properties of the Julia language and compiler. Julia specializes functions on the types of its arguments, including the shape of each container. This makes it possible to write generic code, nonetheless compiled to statically typed assembly without type checks. Furthermore, every function in Julia has its own type. This permits use of higher-order arguments, even user-defined ones as in Listing 8, that still result in specialized code without, e.g., indirect function calls or calls to the runtime. In fact, the PTX code generated from Listing 8 is identical to that generated from the equivalent CUDA C in Listing 9, with the exception of slightly different inlining decisions made by the various compilers. The amount of source code, however, is dramatically reduced: Kernels can be expressed much more naturally, and API interactions (not shown in Listing 9 for the sake of brevity) disappear for most use cases.

Besides relatively simple operations such as *broadcast*, *CuArrays.jl* implements other data processing algorithms optimized for GPUs. This includes a parallel *reduce* implementation based on shuffle instructions [34]. These allow to exchange data between threads within the same thread block without using shared memory or need to synchronize execution. The implementation of shuffle in *CUDAnative.jl* exposes a fully generic interface that specializes on the argument types, whereas even the official CUDA C intrinsics are limited to certain primitive types. As a result, reducing a *CuArray* offers the same flexibility as described for *broadcast*. Listing 10 demonstrates reducing an array of custom objects using the *+* operator (more complex operators are supported but would make the example more confusing). Again, the invocation on line 10 compiles to a single kernel specialized on each of the arguments to *reduce*. This specialization includes generating sequences of 32-bit shuffle instructions to move the 128-bit *Point{Int64}* objects between threads, courtesy of a generated function producing LLVM IR with LLVM.jl as explained in Section 4. The final abstraction completely hides this complexity and demonstrates how metaprogramming can be used to override function behavior selectively and use LLVM.jl to tap into the full potential of the underlying compiler.

The abstractions from this section are idiomatic Julia code, made possible by the *CUDAnative.jl* JIT compiler. *CuArrays.jl* demonstrates the power of such abstractions, by combining the convenience of a host library containing

predefined abstractions, with the flexibility and performance of manual device programming. This greatly improves programmer productivity, while offering the necessary expressiveness to target, e.g., GPUs or other parallel accelerators. More importantly, these abstractions also obviate knowledge about the GPU or its programming model: Algorithms can be expressed as generic operations on multidimensional data, instead of GPU-specific kernels that operate on scalar values.

Finally, generic programming enables composability with existing, GPU agnostic applications and libraries. For example, the *Flux.jl* package provides a flexible machine learning platform capable of implementing a variety of standard machine learning models [35]. Its machine learning core is implemented with operations on generic array types. By initializing the model with a GPU array like `CuArray`, it will automatically be evaluated on the GPU, without needing to adapt any *Flux.jl* code to encode GPU support. Moreover, the models are not restricted to the array operations as implemented in the Julia standard libraries, in `CuArray.jl`, or in *Flux.jl*. With multiple dispatch, it is always possible to implement custom replacement operations. This might be useful, e.g., when GPUs are released in the future with new hardware features. These replacement operations can be implemented using `broadcast` as discussed above, or even as raw kernels. In both cases, they will seamlessly integrate in the existing code base. This approach therefore combines the ease-of-use of high-level array-based interfaces, with the flexibility of low-level kernel programming.

7 RELATED WORK

In recent times, many developments have added support for GPUs and similar accelerators to general purpose, high-level languages without depending on a lower-level, device specific language such as CUDA or OpenCL. One popular approach is to host a DSL in the general-purpose language, with properties that allow it to compile more easily for, e.g., GPUs. For example, Accelerate defines an embedded array language in Haskell [36], while Copperhead works with a functional, data-parallel subset of Python [21]. Parakeet uses a similar Python subset, with less emphasis on the functional aspect [37], whereas PyGPU specializes its DSL for image processing algorithms [20]. Other research defines entirely new languages, such as Lime [38], Chestnut [39] or HIPA^{cc} [40]. In each of these cases, the user needs to gain explicit knowledge about this language, lowering his productivity and impeding reuse of existing code.

We propose compiler infrastructure to write code for accelerators directly in the high-level source language, tightly integrated with the main compiler and language ecosystem. Rootbeer targets similar programmability with Java, but requires manual build-time actions to post-process and compile kernel source code [41]. Jacc features automatic run-time compilation and extraction of implicit parallelism, but requires the programmer to construct manually an execution task-graph using a relatively heavy-weight API [42].

By extending the main compiler, we greatly reduce the effort required to support new targets. This type of extensible programming has been extensively researched in the past [43], and has seen a recent revival [7], 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. The Rust language has experimental support for NVIDIA GPUs that does reuse low-level LLVM infrastructure, but lacks integration with the higher levels of the compiler and focuses on statically compiling device code with little run-time interactions or optimizations [44]. NumbaPro, being a Python compiler, does target a much higher-level language and interface, with corresponding run-time interactions like JIT compilation based on the kernel type signature [8]. However, it uses a custom Python compiler which significantly complicates the implementation and is not fully compatible with the Python language specification.

8 CONCLUSION AND FUTURE WORK

Conclusion: We presented an approach for efficiently adding support for new hardware or other environments to an existing programming language. We proposed a set of interfaces to repurpose the existing compiler, while maximizing reuse of functionality. We implemented these interfaces in the compiler for the high-level Julia programming language, and used that infrastructure to add support for NVIDIA GPUs. We then used the Rodinia benchmark suite to show how Julia can be used to write GPU code that performs similar to CUDA C. By integrating with the existing compiler, code compatibility is improved and many existing Julia packages can be used on the GPU without extra effort.

Our work on `CUDANative.jl` makes it possible to apply high-level principles to GPU programming, for example dynamically typed kernels or interactive programming tools like Jupyter. Furthermore, `CUDANative.jl` and its GPU JIT compiler make it possible to create highly flexible, high-level abstractions. The `CuArrays.jl` package demonstrates this, with a interface that combines the convenience of host-level libraries with the flexibility of manual device programming.

Status: Our changes to the Julia compiler have been accepted, and have been part of it since version 0.6. The extension interfaces are already being used by other researchers and developers to add support for more platforms to the Julia compiler, from similar hardware like AMD GPUs, to WebAssembly for targeting web browsers. These new developments invariably make use of `LLVM.jl`, and often mimic the design of `CUDANative.jl` as a starting point.

Our packages are critical to large parts of the Julia GPU infrastructure for NVIDIA GPUs. Furthermore, they work out of the box with Jupyter, enabling interactive programming and effective use of GPU hardware.

Future Work: We plan to improve support for GPU hardware features, and create high-level abstractions that maintain the ability to express low-level behavior. This includes a unified approach to GPU memory types, idiomatic support for communication primitives, etc. We are also working on compiler improvements to enable even more powerful abstractions, for example contextual method dispatch based on run-time device properties. This can both enhance expressiveness of the abstractions, and improve performance of the generated code.

ACKNOWLEDGMENTS

This work is supported by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT Vlaanderen), and by Ghent University through the Concerted Research Action on distributed smart cameras.

REFERENCES

- [1] C. Kachris and D. Soudris, "A survey on reconfigurable accelerators for cloud computing," in *Int'l Conf. Field Programmable Logic and Applications*. IEEE, 2016, pp. 1–10.
- [2] J. Weerasinghe *et al.*, "Enabling FPGAs in hyperscale data centers," in *Proc. Int'l Conf. Ubiquitous Intelligence and Computing, Autonomic and Trusted Computing, Scalable Computing and Communications*. IEEE, 2015, pp. 1078–1086.
- [3] S. Sarkar *et al.*, "Hardware accelerators for biocomputing: A survey," in *Proc. Int'l Symp. Circuits and Systems*. IEEE, 2010, pp. 3789–3792.
- [4] G. Pratz and L. Xing, "GPU computing in medical physics: A review," *Medical physics*, vol. 38, no. 5, pp. 2685–2697, 2011.
- [5] S. Aluru and N. Jammula, "A review of hardware acceleration for computational genomics," *IEEE Design & Test*, vol. 31, no. 1, 2014.
- [6] X. Li *et al.*, "Comparing programmer productivity in OpenACC and CUDA: an empirical investigation," *Int'l J. Computer Science, Engineering and Applications*, vol. 6, no. 5, pp. 1–15, 2016.
- [7] D. Zingaro, "Modern extensible languages," *SQRL Report*, vol. 47, p. 16, 2007.
- [8] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A LLVM-based Python JIT compiler," in *Proc. 2nd Workshop LLVM Compiler Infrastructure in HPC*. ACM, 2015, p. 7.
- [9] P. Du *et al.*, "From CUDA to OpenCL: Towards a performance-portable solution for multi-platform gpu programming," *Parallel Computing*, vol. 38, no. 8, pp. 391–407, 2012.
- [10] D. Merrill, "CUB: A pattern of "collective" software design, abstraction, and reuse for kernel-level programming," 2015, GPU Technology Conf.
- [11] J. Hoberock and N. Bell, "Thrust: A parallel template library," 2010. [Online]. Available: <https://developer.nvidia.com/thrust>
- [12] J. J. Dongarra *et al.*, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Mathematical Software*, vol. 16, no. 1, 1990.
- [13] E. Anderson *et al.*, *LAPACK Users' guide*. SIAM, 1999.
- [14] J. Malcolm *et al.*, "Arrayfire: a GPU acceleration platform," in *Proc. SPIE*, vol. 8403, 2012, pp. 84030A–1.
- [15] K. Rupp, F. Rudolf, and J. Weinbub, "ViennaCL: A high level linear algebra library for GPUs and multi-core CPUs," in *Int'l Workshop GPUs and Scientific Applications*, 2010, pp. 51–56.
- [16] NVIDIA, "cuBLAS: Dense linear algebra on GPUs," 2008. [Online]. Available: <https://developer.nvidia.com/cublas>
- [17] Continuum Analytics, "Anaconda Accelerate: GPU-accelerated numerical libraries for Python," 2017. [Online]. Available: <https://docs.anaconda.com/accelerate/>
- [18] Julia developers, "CUBLAS.jl: Julia interface to cuBLAS," 2017. [Online]. Available: <https://github.com/JuliaGPU/CUBLAS.jl/>
- [19] N. Markovskiy. (2014, 6) Drop-in acceleration of GNU Octave. NVIDIA. [Online]. Available: <https://devblogs.nvidia.com/paralleforall/drop-in-acceleration-gnu-octave/>
- [20] C. Lejdfors and L. Ohlsson, "PyGPU: A high-level language for high-speed image processing," in *Int'l Conf. Applied Computing 2007*. IADIS, 2007, pp. 66–81.
- [21] B. Catanzaro *et al.*, "Copperhead: Compiling an embedded data parallel language," *ACM SIGPLAN Notices*, vol. 46, no. 8, 2011.
- [22] J. Bezanson *et al.*, "Julia: A fresh approach to numerical computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, 2017.
- [23] J. W. Bezanson, "Abstractions in technical computing," Ph.D. dissertation, Massachusetts Institute of Technology, 2015.
- [24] J. Bezanson *et al.*, "Julia: A fast dynamic language for technical computing," *arXiv preprint arXiv:1209.5145*, 2012.
- [25] T. Besard, B. De Sutter, A. Frías-Velázquez, and W. Philips, "Case study of multiple trace transform implementations," *Int'l J. High Performance Computing Applications*, vol. 29, no. 4, pp. 489–505, 2015.
- [26] J. Bezanson, "Why is Julia fast? Can it be faster?" 2015, JuliaCon India.
- [27] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int Symp. Code Generation and Optimization*, 2004, pp. 75–86.
- [28] J. Wu *et al.*, "gpuc: An open-source GPGPU compiler," in *Proc. Int'l Symp. Code Generation and Optimization*. ACM, 2016, pp. 105–116.
- [29] M. Ragan-Kelley *et al.*, "The Jupyter/IPython architecture: a unified view of computational research, from interactive exploration to communication and publication," in *AGU Fall Meeting Abstracts*, 2014.
- [30] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *Int'l Symp. Workload Characterization*, 2009.
- [31] M. Giordano, "Uncertainty propagation with functionally correlated quantities," *arXiv preprint arXiv:1610.08716*, 2016.
- [32] S. G. Johnson. (2017) More dots: Syntactic loop fusion in Julia. [Online]. Available: <https://julialang.org/blog/2017/01/moredots>
- [33] M. Innes, "CuArrays.jl: CUDA-accelerated arrays for Julia," 2017. [Online]. Available: <https://github.com/FluxML/CuArrays.jl>
- [34] J. Luitjens. (2015) Faster parallel reductions on Kepler.
- [35] M. Innes, "Flux: Elegant machine learning with Julia," *The Journal of Open Source Software*, vol. 3, no. 25, p. 602, 2018.
- [36] M. M. Chakravarty *et al.*, "Accelerating Haskell array codes with multicore GPUs," in *Proc. 6th Workshop Declarative Aspects of Multicore Programming*. ACM, 2011, pp. 3–14.
- [37] A. Rubinsteyn *et al.*, "Parakeet: A just-in-time parallel accelerator for Python," in *USENIX Conf. Hot Topics in Parallelism*, 2012.
- [38] C. Dubach *et al.*, "Compiling a high-level language for GPUs," *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 1–12, 2012.
- [39] A. Stromme *et al.*, "Chestnut: A GPU programming language for non-experts," in *Proc. Int'l Workshop on Programming Models and Applications for Multicores and Manycores*, 2012.
- [40] R. Membarth *et al.*, "HIPA^{cc}: A domain-specific language and compiler for image processing," *IEEE Trans. Parallel and Distributed Systems*, vol. 27, no. 1, pp. 210–224, 2016.
- [41] P. C. Pratt-Szeliga *et al.*, "Rootbeer: Seamlessly using GPUs from Java," in *Int Conf High Performance Computing and Communication*, 2012, pp. 375–380.
- [42] J. Clarkson, C. Kotselidis, G. Brown, and M. Luján, "Boosting Java performance using GPGPUs," *arXiv preprint arXiv:1508.06791*, 2015.
- [43] N. Solntseff and A. Yezerski, "A survey of extensible programming languages," *Annual review in automatic programming*, vol. 7, 1974.
- [44] E. Holk *et al.*, "GPU programming in Rust: Implementing high-level abstractions in a systems-level language," in *Parallel and Distributed Processing Symp. Workshops & PhD Forum*, 2013.



Tim Besard is a PhD student at Ghent University in the Computer Systems Lab. He obtained his MSc in Computer Engineering from University College Ghent in 2011. His research focuses on compilation techniques of high-level languages for GPUs.



Christophe Foket is a post-doctoral researcher at Ghent University in the Computer Systems Lab. He obtained his MSc and PhD degrees in Computer Science from Ghent University's Faculty of Engineering in 2009 and 2015. His research focuses on compilation techniques of high-level languages for GPUs.



Bjorn De Sutter is associate professor at Ghent University in the Computer Systems Lab. He obtained his MSc and PhD degrees in Computer Science from Ghent University's Faculty of Engineering in 1997 and 2002. His research focuses on the use of compiler techniques to aid programmers with non-functional aspects of their software, such as performance, code size, reliability, and software protection.