Transparent Distributed Programming in Julia

by

Valentin Churavy
B.Sc, Universität Osnabrück (2014)

Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Science

at the

Massachusetts Institute of Technology
Cambridge, Massachusetts

June 2019

Signature redacted

Signature of Author: ____________________
Department of Electrical Engineering and Computer Science
May 23, 2019

Signature redacted

Certified by: ____________________
Alan Edelman
Professor of Applied Mathematics
Computer Science and AI Laboratories
Applied Computing Group Leader

Signature redacted

Accepted by: ____________________
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Abstract

Scientific and engineering problems grow ever larger and more challenging; solving them requires taking advantage of domain expertise and modern compute capabilities. This encourages efficient usage of GPUs and using large scale cluster environments efficiently. Domain experts should not need to acquire the deep knowledge required to develop applications that scale, but rather should be able to express data science and engineering problems in terms of vectorized operations and linear algebra, that is in language inherent to the field.

The approach introduced here, gives performance engineers access to low-level capabilities of the hardware, allowing them to collaborate with domain experts in the same language. This removes the need to rewrite scientific code in a low-level language, speeding up the iteration cycle and allowing for rapid prototyping.

We investigate composable, layered abstractions for scientific computing. They separate the user intent, the what, from the how of the implementation and the where of the execution. The focus is on the distributed aspects, how array abstractions for distributed and accelerated computing can compose with each other and how we can provide access to low-level capabilities in a transparent fashion.

Building and debugging these abstractions is challenging. This work introduces *Cthulhu*, a unique debugging tool for abstractions, that takes into consideration the dynamic execution model and the static compilation process of Julia.

Acknowledgments

I am immensely grateful to Tim Besard for his continued collaboration and the many valuable discussions we had over the years, without him GPU accelerated programming in Julia would hardly be possible. The Julia community has been an important part of my last few years and I can't thank all of the many people that make it such a wonderful community to be part of. I am thankful to Peter Ahrens and Jarret Revels for their valuable discussions, comments, and a fabulous work environment.

I would like to thank Jameson Nash, Matt Bauman, Keno Fischer, Jeff Bezanson, and Andreas Noack for their work on these topics, their mentorship, and for welcoming me and my ideas. These are the giants, whose shoulders I stand upon.

I am grateful for my thesis advisor, Alan Edelman, who always provided me with the freedom to explore my ideas and who reminds that what matters most is to make someone else's life just a little bit easier.

Many thanks to my collaborators Ali Ramadhan, Greg Wagner, Lucas Wilcox, Jeremy Kozdon, Chris Rackauckas and Yingbo Ma for being open to experimentation and their input on these subjects. I thank Simon Danish for his work on *GPUArrays* and Julia Samaroo for asking me questions about code generation and for working on *AMDGPUnative*.

Thank you, to all my family and friends, who have cheered me on and who have provided, Coffee, Chocolate and Conversation. Special thanks to both Stefan Polenz and Nili Persits for their valuable comments on this thesis.

This research is supported in part by NSF DMS-1312831, NSF OAC-1835443, Darpa XDATA, and an ARAMCO MITEI grant.

Listings

List of Tables

List of Figures

1 Introduction

Scientists, engineers and researchers work on ever growing problems, that require more and more computation. They also face challenging applications and need to quickly adapt to ever changing requirements. We want domain experts to quickly be able to prototype, iterate and evaluate new ideas on data-sets ranging from small to large. Programs should be portable, run on accelerators, work on large scale cluster environments, and be worked on by teams of domain experts and performance engineers.

This thesis presents my work towards achieving this goal, I will discuss a distributed and heterogeneous programming model in the Julia [12, 11, 14] programming language, originally presented as joint work in Besard et al. [8].

I will focus on the distributed aspects, expand on it by introducing a unified kernel language for heterogeneous programming, and a powerful debugging tool for understanding complex programs using abstractions in Julia.

In Sections 1.1 and 1.2 I introduce the foundations necessary to build distributed infrastructure in Julia. In Sections 1.3 and 1.4 I briefly discuss Julia's compiler and type-inference mechanism. I find it important to built an appreciation of both in developers and users of the language so that they are able to build and debug abstractions and advanced programs in Julia.

I will then introduce the set of abstractions from Besard et al. [8], that form a programming model well suited for scientific programming in Section 2. During that work we observed performance limitations of distributed programming in Julia, and in Section 3 I will discuss these and potential pathways of addressing them.

The array focused programming model, introduced in Section 2, is good for rapid prototyping, but advanced users have expressed a need for more low-level control and to express computational kernels directly. In Section 4 I introduce a small package, that implements a limited kernel language for heterogeneous programming.

These contributions are extensions to the programming language Julia and are made possible by extensively taking advantage of advanced features; some of which make it non-trivial to debug and understand applications using them. In Section 5 I introduce a tool I developed to help debug advanced applications.

Listing 1: Simple asynchronous code example

```
1 @sync for i in 1:10
2   @async begin
3     println("Hello from task $i")
4   end
5 end
```

Listing 2: Expanded code

```
tasks = Task[]
for i in 1:10
  thunk = () -> println("Hello from task $i")
  task = Task(thunk)
  push!(tasks, task)
  schedule(task)
end
Base.sync_end(tasks)
```

1.1 Asynchronous programming in Julia

Asynchronous programming is not the focus of this thesis, but it is used through-out the distributed programming stack in Julia, see Listing 3 as an example of how both interact.

Julia supports asynchronous programming in the form of `Tasks`, tasks are concurrently executed threads of control that communicate through `Channels` and memory. They are an implementation of communicating sequential processes (CSP) [15] and are scheduled cooperatively on a single-threaded runtime. They yield control at explicit yield points – e.g. calls to `yield` – or implicit yield points, like blocking on a condition, lock, or waiting for IO.

Listing 1 is an example of tasks in action. The `@async` macro in Line 2 creates a task object, that takes an anonymous function to be executed. The anonymous function consist of the expression given to the `@async` macro. The task is then scheduled in the runtime for execution. Listing 2 contains the expanded code generated from the macros.

The `@sync` macro synchronizes on all task created in it's lexical scope, by waiting on all of them to complete.

1.2 Distributed Programming in Julia

There are two prevalent programming models for distributed programming in Julia. An asynchronous master-worker model based on futures and remote-procedure calls (RPCs) and Message Passing Interface (MPI[38]) as the more traditional model.

Much ink has been spilled on the benefits or lack thereof of MPI and I will focus on an analysis of the master-worker model as implemented in Julia standard-library `Distributed.jl`.

There has been work done in Julia to implement a Partitioned Global Address Space (PGAS) model. The `GASP.jl` [40, 45] used for the Celeste project is a good example, but most users and most of the infrastructure discussed here is built on top of the RPC interface.

The primary user-facing interface consists of two data-structures `Future` and `RemoteChannel`, and several variants of `remotecall`. A future represents a value being computed on a remote process, which can be fetched and waited upon, but the future can only be set to a single value. Remote channels extend this concept by allowing several values to be produced and fetched. They function similar to channels used in communication between tasks.

A `remotecall` invokes a function or lambda on a remote process, it returns a `Future` as the handle for the response. There are two variants `remotecall_fetch` and `remotecall_wait`, that aim to minimize the number of messages sent. Both variants will block until a response with the data has been received.

`remotecall_fetch(...)` is equivalent to `fetch(remotecall(...))`, it is automatically fetching the result of the remote call, and `remotecall_wait` automatically waits on the remote call to finish.

Under the hood remote calls are implemented through different message types that are serialized and sent across a network channel. This is similar to active messages [21]. The messages have unique tags and after deserialization the corresponding message handler is invoked.

We will see how this minimal interface can be used to implement high-level abstractions in Section 2.5. A common pattern is to combine asynchronous programming with the blocking variants of remotecall. This reduces the amount of network traffic while allowing for several remote procedures to be started across the master-worker system. As an example Listing 3 invokes a thunk on each worker process that generates some random data and sends back the sum of it to the master process. This intertwined style makes it also harder to debug and profile distributed code in Julia and was one of the primary motivations for the creation of more advanced tools in Section 5.1.

Listing 3: Combining asynchronous and distributed programming

```julia
results = Float64[]
@sync for p in workers()
  @async begin
    r = remotecall_fetch(p) do
      data = rand(64)
      sum(data)
    end
    push!(results, r)
  end
end
sum(results)
```

1.3 Multiple level of representation

All computer programs eventually have to be translated into action to be performed by a computer. This can either happen through interpretation or through compilation and later execution. The Julia language is, despite its dynamic nature, a primarily compiled language. The Julia compiler uses multiple levels of representation to optimize code and in the end to translate it into machine code.

By using multiple representations each layer can focus on one task at a time and the code successively gets transformed from high-level code written by humans to low-level code suitable to execution on a machine. The process is called lowering and is used in many programming languages and compilers.

One powerful characteristic of the Julia language is that it allows access to these various intermediate representations through reflection methods. Table 1 contains a list of various representations and the corresponding reflection method, starting from the highest, Julia source code, and ending with the lowest, machine code. The reflection methods **`@code_*`** operate on the function level and take a callsite like **`@code_lowered f(1.0)`**.

Table 1: Hierarchy of representations, and the reflection macros used to obtain each

	Julia source code
`@code_lowered`	Lowered code
`@code_typed optimize=false`	Type-inferred code, without optimizations
`@code_typed optimize=true`	Type-inferred code, with optimizations – primarily inlining
`@code_llvm optimize=false`	LLVM IR – as produced by code generation
`@code_llvm optimize=true`	LLVM IR – post-optimizations
`@code_native`	Machine code

Examples of the output of various reflection methods will be given throughout this work and Section 5.1 I will introduce a tool to easily explore these various levels.

1.4 A brief introduction to Julia's type-inference

One of the major benefits of the Julia programming language is that despite it being an utterly dynamic programming language, it offers performance comparable to static languages such as C and C++. This is achieved using, *type-inference* [14, 50], aggressive devirtualization, and compilation through LLVM [30]. Especially type-inference is important to obtain good performance. This in turn rewards programmers with advanced understanding of it and an appreciation for its capabilities and limitations. In particular since most knowledge gained by doing performance optimizations in other programming languages still applies, but understanding the action of type-inference on a users code and the performance consequences that go along is not trivial for interesting computation.

The type-inference algorithm can be summarized as such: Given a function signature (set of concrete types), using abstract interpretation to propagate the types in control-flow order through the function. During propagation we can encounter three kinds of functions, built-ins, intrinsics or generic functions. To compute the return type of a built-in or intrinsic function we can use `t-funcs` (type functions), that take a signature and return the output type of the intrinsic or built-in, whereas computing the return type of a generic function, requires us to apply type-inference recursively. This recursion is terminated by either encountering a function that consists consists solely of intrinsics or built-ins, or by triggering the recursion detection and returning an upper type-bound. That information can now be used while inferring the caller.

An example is presented in Listing 4; given the function call `f(1, 2)`, we have the signature `(f, Int64, Int64)` where `Int64` is a concrete type. Knowing the signature of the function we are calling and with the `t-funcs` in Listing 5, we can use its lowered representation in Listing 6 to obtain the type-inferred result in Listing 7. After type-inference we now know that the result of this call signature will be a `Int64` and we can reuse this result when inferring other functions that call `f`.

Note that this process as described is unbounded, mostly due to the presence of recursion in the user program. Therefore type-inference is limited when detecting a recursive cycle and the return type of that function is inferred to the current best upper bound or `Any` and the function is evaluated with dynamic semantics.

Listing 4: Inferring a simple function

```
function f(x, y)
  return x + 2*y
end
```

Listing 5: Simplified t-funcs

```
tfunc(::typeof(+), ::Type{T}, ::Type{T}) where T = T
tfunc(::typeof(*), ::Type{T}, ::Type{T}) where T = T
```

Listing 6: Lowered representation

```
%1 = 2 * y
%2 = x + %1
return %2
```

Listing 7: After type-inference

```
%1 = (2 * y)::Int64
%2 = (x + %1)::Int64
return %2::Int64
```

Programs that perform computation in the type-domain are challenging to analyze since they can cause unbounded behavior. In Listing 8 the input value of **`fib`** is lifted into the type-domain by wrapping it in **`Val`**. As we can see in Listing 9 the return value of **`fib`** is constant. We can introspect the compiler a bit further and see the lattice element **`Const`** that is used for constant propagation in Listing 10.

Since recursion is a common pattern in code, the compiler has relaxed heuristics in the case of self-recursion, but one can easily run into scenarios where an additional indirection will defeat the inference pass and the compiler falls back onto dynamic semantics.

In the following I will present three cases; first a computation in the type-domain with increasing values (Listing 11), next by introducing an intermediate function call the self-recursion detection will be stimified (Listing 15) and lastly an example of bounded structural complexity (Listing 16).

The first example in Listing 11 is deceptively simple, we define a function **`limit`**, that is self recursive and increments `N` until reaching a limit at k ($k = 1000$ is arbitrarily chosen). In Listing 8 we have seen that such computation in the type-domain can be constant folded. Albeit, if we inspect the code with **`code_typed`** in Listing 12 we can see that dynamic semantics are being executed. The reason is that the Julia compiler tries harder to infer type-domain computations that are potentially convergent and punts on computations that

Listing 8: Calculating the fibonacci sequence in the type-domain

```julia
fib(::Val{0}) = 0
fib(::Val{1}) = 1
fib(::Val{N}) where N = fib(Val(N-1)) + fib(Val(N-2))
```

Listing 9: After type-inference and optimizations

```
julia> @code_typed fib(Val(20))
CodeInfo(
1       return 6765
) => Int64
```

Listing 10: After type-inference, without optimizations

```
julia> @code_typed optimize=false fib(Val(20))
CodeInfo(
1  %1 = ($(Expr(:static_parameter, 1)) - 1)::Const(19, false)
   %2 = (Main.Val)(%1)::Const(Val{19}(), true)
   %3 = (Main.fib)(%2)::Const(4181, false)
   %4 = ($(Expr(:static_parameter, 1)) - 2)::Const(18, false)
   %5 = (Main.Val)(%4)::Const(Val{18}(), true)
   %6 = (Main.fib)(%5)::Const(2584, false)
   %7 = (%3 + %6)::Const(6765, false)
        return %7
) => Int64
```

are potentially divergent. This includes differentiating between subtraction and additions, as we can see in Listing 13 and Listing 14.

The second example in Listing 15 demonstrates the limitation that functions have to be self-recursive to get the benefit of relaxed heuristics. It is in fact similar to the earlier example of calculating the Fibonacci sequence in Listing 8. The only difference is that instead of being self-recursive it is a recursive cycle of two functions, for each call to `fib` we call the function `notfib`, which in return calls `fib`, forming a cycle. The Julia compiler will not perform the same level of optimizations as in Listing 9, since we are no longer the triggering the self-recursive heuristic.

An additional limitation is structural complexity of types, in order to prevent geometric blowup, the Julia compiler limits the inference of functions whose signatures are not decreasing in complexity, even in the case of self-recursion. Take Listing 16 as a fun example. In it we define the natural numbers as tuples! So zero is `()`, one is `((),)`, and two is `(((),),)`, and so forth. Line 2 defines the successor function and the next three lines define the empty tuple to be the zero element. Line 6 finally defines the addition of two natural numbers, by unpacking `T2` and incrementing `T1`, we recurse until `T2` is zero.

Listing 11: **`limit`** function that operates in the type-domain

```

limit(::Val{1000}) = 1000
limit(::Val{N}) where N = limit(Val(N+1))
```

Listing 12: Heuristic assumes **`limit`** diverges

```
julia> @code_typed limit(Val(0))
CodeInfo(
1  %1 = invoke Main.limit($(QuoteNode(Val{2}()))::Val{2})::Int64
       return %1
) => Int64
```

Listing 13: **`limit`** function that operates in the type-domain, but uses decrement instead of increment

```
mit(::Val{1000}) = 1000
mit(::Val{N}) where N = limit(Val(N-1))
```

Listing 14: Heuristic assumes that **`limit`** converges

```
lia> @code_typed limit(Val(2000))
deInfo(
     return 1000
=> Int64
```

Listing 15: Not all problems can be solved by adding one layer of abstraction

```
fib(::Val{0}) = 0
fib(::Val{1}) = 1
fib(::Val{N}) where N = notfib(Val(N-1)) + notfib(Val(N-2))
```

Listing 16: The natural numbers defined as nested tuples

```

S(T::Tuple) = (T,)
+(::Tuple{}, T::Tuple{Any}) = T
+(T::Tuple{Any}, ::Tuple{}) = T
+(::Tuple{}, ::Tuple{}) = Tuple{}
+(T1::Tuple{Any}, T2::Tuple{Any}) = (T1,) + first(T2)
```

Julia will infer `((),) + (((() ,) ,) ,)`, e.g. $1 + 3$, to be constant, whereas $1 + 4$, e.g `((),) + ((((() ,) ,) ,) ,)` will not be inferred to be constant, due to the structural complexity of the recursive call not decreasing.

Optimizations During the type-inference process the Julia optimizer also performs more traditional compiler optimizations, such as constant propagation and function inlining.

2 A transparent programming model

In Besard et al. [8] we presented a programming model focused on array based programming, that lends itself well for prototyping of scientific and engineering software. We found that in particular such a programming model allows for the easy usage of distributed computing and composes well with heterogeneous/accelerated computing.

Julia is the perfect language to provide such an infrastructure in, since due to its dynamical nature and its goal to being close to mathematics, it is easy to pick up, and due to the use of type-inference (as discussed in Section 1.4) and a just-in-time (JIT) compiler built on-top of LLVM [30] it provides excellent performance.

A fundamental aspect of my work in Besard et al. [8], was to focus on a strong separation of concerns, using Julia's multiple-dispatch to layer implementations of array abstractions in a transparent fashion. This design facilitates code reuse and fosters collaborations, between domain experts and performance engineers, who can use these layered abstractions to provide optimized hardware-specific implementations. I will discuss these ideas in Section 2.5.1 and in the following I will provide the basis of the programming model and how two of the implementations, namely `CuArray.jl` and `DistributedArrays.jl`, work.

In order to achieve high-performance codes in a given domain, programmers have to encode a great deal of knowledge about the problems and about how to efficiently solve them. In the linear algebra domain this means exploiting knowledge about structured matrices – not just sparse versus dense.

If abstractions are not well layered, or not present, them implementing programs requires an in-depth understanding of the chosen programming language and all levels of execution. To achieve the computational performance necessary for large programs additionally requires often the need to code in a low-level language. This greatly increases the barrier to entry and makes it harder to write efficient and reusable code.

The core tenant of the programming model we are advocating for is the expression of data science and engineering problems in terms of vectorized operations and linear algebra, that is in language inherent to the field, not to the programming language. This natural and concise representation makes it easier to iterate over different prototype implementations, and from an implementation standpoint nicely separates the concerns of *what* from the *how*. Several examples of high-level programs can be found in Besard et al. [8].

The operations we will focus on are higher-order functional constructs such as `map`, `reduce`, `mapreduce`, and `broadcast`. They are commonly used in Julia and many other

Listing 17: Example use of the `reduce` abstraction.

```
1 a::Array{Int} = [1 2; 3 4]
2
3 reduce(+,             a)
4 reduce((x,y)->2x+y^2, a)
```

high-level languages such as R, Python and Matlab. In contrast to other languages, in Julia they are not required to achieve high-performance, and are not implemented in a low-level high-performance language. Rather, they are implemented in Julia itself [13], allowing for composability and layering of abstractions, while maintaining great performance.

At the same time Julia's compiler infrastructure enables higher-order abstractions that compose with arbitrary user code. The `reduce` abstraction is a prime example of such an abstraction. Listing 17 illustrates how the first argument to the `reduce` function can be any transformation function that reduces two scalar values. The compiler specializes the implementation of `reduce`, which only deals with the semantics of the abstraction, with the transformation function as specified by the user. This can be an operation or function from the standard library, as on Line 3, or a user-specified one as shown on Line 4. Furthermore, the underlying storage is implemented by a separate container type. In the example this is the standard `Array`, 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.

The expressiveness and performance of these array abstractions makes it possible to reuse them outside of prototyping code. Array abstractions on generically typed arrays are used, e.g., in the `ForwardDiff.jl` package. The code in the package can be composed with any concrete array implementation, which makes the package equally suited for use during prototyping and for reuse as is in optimized production code. In Sections 2.4 and 2.5, we will further focus on portability through the use of different array types.

2.1 The `map`, `reduce`, and `broadcast` abstractions

The `map`, `reduce`, and `broadcast` functions are higher-order abstractions that are essential to high-level array programming in Julia. They compose with user code that determines *what* is computed, while the methods that implement these abstractions determine *how* and *where* that computation will happen. These implementations can be specialized on

Listing 18: Example use of the `map` and `broadcast` abstractions.

```
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])
```

the type of the arguments, selecting an implementation that maximizes performance or otherwise preserves the array type, e.g., to prevent slowdown due to unnecessary memory transfers. This is especially important in the realm of high performance computing, where it is crucial to minimize the amount of allocation and to efficiently use the available memory bandwidth

At its core, `map` transforms collections by applying a function elementwise over its arguments, as shown in Listing 18. The arguments are collections that all have to be of the same shape.

The `broadcast` abstractions generalizes the behavior of `map` to containers of heterogeneous shapes by padding dimensions accordingly. This greatly improves use with objects of different shape. For example:

$$\texttt{broadcast}(f, \mathbf{A}, b, \mathbf{c}) = \texttt{broadcast}(f, \begin{bmatrix} A_{11} & \dots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{m1} & \dots & A_{mn} \end{bmatrix}, b, \begin{bmatrix} c_1 \\ \vdots \\ c_m \end{bmatrix})$$
$$= \begin{bmatrix} f(A_{11}, b, c_1) & \dots & f(A_{1n}, b, c_1) \\ \vdots & \ddots & \vdots \\ f(A_{m1}, b, c_m) & \dots & f(A_{mn}, b, c_m) \end{bmatrix}$$

The `reduce` abstraction reduces the dimensionality of a container by applying a binary function along certain dimensions of an collection. As an example computing the sum of an array by calling `reduce(+, array)`, reduces the `array::Array{T, N}` from a N-dimensional object to a zero-dimensional object – e.g. a scalar. By choosing a dimension, or set of dimensions, to reduce over, we generally go from N to $N - M$ dimensions, where M is the number of dimensions to be reduced over.

Table 2: Different forms of broadcast syntax and their execution

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

A common pattern is to call `reduce` after having performed a `map`. This computation can be performed with a single call to `mapreduce` instead, slightly improving performance by avoiding the intermediate array as returned by the inner `map`.

Although seemingly simple, these abstractions are very versatile and capable of expressing a wide range of computations. Furthermore, the abstractions expose a great deal of parallelism, and are therefore ideal candidates for parallel programming. This will be discussed in Section 2.3.

2.2 Dot Expressions

As a means of making `broadcast` easier to use, so-called *dot expressions* can be used in Julia to denote elementwise transformations [25]. The Julia parser lowers this syntactic sugar to invocations of the `broadcast` function, as illustrated with some examples in Table 2.

Elementwise assignments call the `broadcast!` function, which performs in-place assignment to avoid allocating an output container.

A `broadcast` expression is treated a single statement in the Julia parser, and instead of being lowered to a series of calls to `broadcast`, the expression is lowered to several calls to `broadcasted`, which constructs a `Broadcasted` from the expression. A `Broadcasted` represents a broadcast tree, e.g. the application of an elementwise function over a set of arguments. The arguments are either scalars, collections, or `Broadcasted` themselves. The constructed `Broadcasted` is then passed to the `materialize` or `materialize!` function in case of a in-place broadcast [6].

The `Broadcasted` is used to perform various transformation, among them broadcast fusion, which eliminates the need for temporary storage. Furthermore the `Broadcasted` is accessible to implementers at runtime, which enables fine-grained customization of 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 for splitting broadcast expressions into chunks that can be computed in parallel.

Despite `Broadcasted` being a runtime object, its construction can often be elided due to Julia's type-inference and optimizations.

2.3 Array Infrastructure Portability

The previous sections have only used `Array`, the basic array type in the Julia base library, but this conscious layering of abstractions allows user code to 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.

2.4 CuArrays.jl

The *CuArrays.jl* package [26] defines a `CuArray` type alongside optimized implementations of many common array operations for NVIDIA GPUs. Some of 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 2 are implemented on top of CUDAnative.jl [9], a package that compiles arbitrary Julia code to PTX machine code for NVIDIA GPUs. The performance of code generated by this package is on-par with the performance of CUDA C as compiled by the NVIDIA compiler [9]

CuArrays [8] and CUDAnative [9] are the foundations for the first-class GPU ecosystem in Julia and have seen wide adoption. Besard et al. [8] contains a full description of its capabilities and here I will focus on the distributed aspects of the work.

2.5 DistributedArrays.jl

The *DistributedArrays.jl* package builds upon Julia's distributed computing infrastructure to provide a Global Array-like interface [39]. A `DArray` is a 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 Remote Procedure Calls (RPCs) are issued automatically when accessing memory that is not local to the process. This makes it possible to

Listing 19: Low-level implementation of in-place `map` taken from DistributedArrays.jl.

```julia
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
```

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, since 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 2.5.1 will show how this pattern makes it possible to compose array types that, like `DArray`, wrap other arrays.

Listing 19 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 the computations on the data it owns. The example implements an in-place `map` through 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 19 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` that executes 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 the data is already locally available this copy is avoided. The call to `remotecall_wait` is a blocking RPC and is wrapped into an `@async` block, which starts a lightweight task.

Listing 20: High-level use of the **map!** abstraction with distributed arrays from DistributedArrays.jl

```
 1 # prepare a parallel computing environment
 2 using Distributed
 3 addprocs(2)
 4
 5 using DistributedArrays
 6
 7 a = distribute(rand(2,2))
 8 b = similar(a)
 9
10 map!(sin, b, a)
```

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 on all enclosed tasks to make sure the computation is finished when returning from the `map!` function.

The distributed computing abstractions as used in Listing 19 are defined in the Julia standard library. They are built on top of a **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 Secure Shell (SSH) 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 [49], the Portable Batch System [22], and others. For environments that rely on the Message Passing Interface (MPI), **MPIManager** from MPI.jl can be used to communicate with processes over an optimized communication fabric such as InfiniBand [32]. 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 19 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 20 shows how to use the `map!` abstraction from Listing 19 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.

2.5.1 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 2.5, the distributed chunks of a `DArray` are arrays, typically regular CPU-based `Arrays`, 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, the resulting `DArray{CuArray}` object 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.

Listing 19 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.

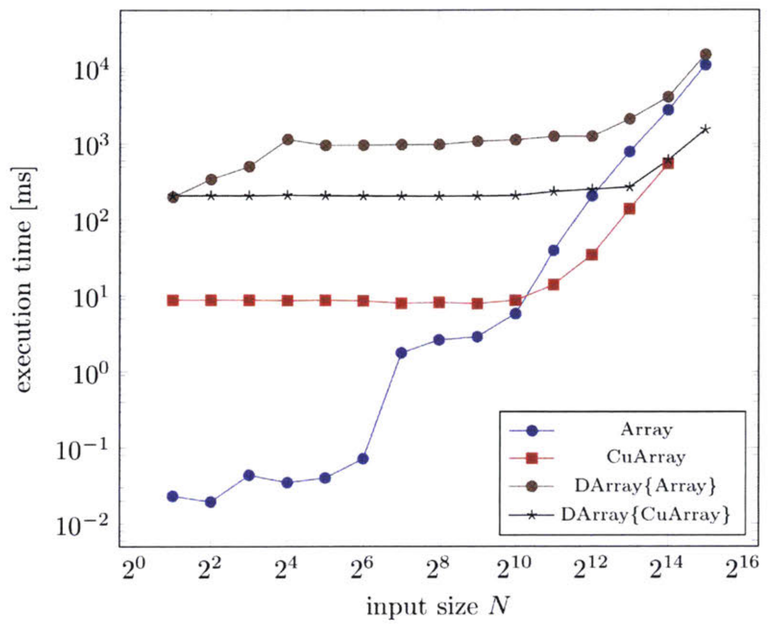

Figure 1: Time to execute the **domeigen** function from Besard et al. [8] and compute the dominant eigenvector and eigenvalue of a $N \times N$ matrix. We benchmark for 1000 iterations of the power method, approximating the reference eigenvalue with sufficient accuracy.

3 Towards understandable performance

In Besard et al. [8] we discussed three examples that highlight different aspects of writing high-level code. In Figure 1 as an example we measured a simple iterative method for calculating the dominant eigenvector and corresponding eigenvalue of a matrix.

The above results showed that distributed arrays display 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. However the communication does not explain all the overhead, some of it is caused by actual inefficiencies in the current implementation of `DArray`.

The first major inefficiency stems from the fact that communication and computation share the same thread. As explained in Section 1.1 Julia uses one event loop to schedule tasks, which enables forward progress when a task is blocked on IO. The scheduler is current cooperative tasks, relying on tasks to yield frequently. This has the downside that

Listing 21: Example of a busy worker causing a stall due to communication dependencies.

```
asyncmap([2, 3, 4]) do p
    if p == 2
        remotecall_fetch(p, data) do D
            InteractiveUtils.peakflops()
        end
    else
        remotecall_fetch(p, data) do D
            remotecall_fetch(2, D) do D
                DistributedArrays.localpart(D)
            end
        end
    end
end
```

a task not yielding back control to the event loop, because it is busy with a numerical computation, causes other tasks to be stalled. If one of those stalled task is responsible for communicating with other workers, those other workers will in turn stall. This in turns causes more workers to stall - effectively returning to single threaded computation (with extra overhead).

To investigate this further we examine the code snippet in Listing 21, using three worker processes labeled `1,2,3`. We start a long-running computation on worker `2` using the remote-procedure infrastructure introduced in Section 1.2, and then try to access data on process `2` from process `3, 4`.

Using a Julia implementation of vectorclocks [34] and the `ShiViz` and `TSViz` tools from Beschastnikh et al. [10], we can observe a serialization of the requests from workers `3` and `4`. This limits the forward progress the system can make and inhibits scaling.

There are long-term plans to address these issues, by allowing multiple tasks in a single worker process to execute in parallel, greatly reducing the frequency of stalling and hopefully allowing us to add a singular thread responsible for satisfying communication requests.

Another slowdown is due to the many data copies occurring as part of IPC. The vector-matrix product in the `domeigen` function 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 Remote Direct Memory Access (RDMA) 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, and would have to explore alternative approaches. For now, the communication overhead is significant. As a result, the matrix-vector product used in `domeigen` shows little speed-up with `DArray{Array}`. It is bound by memory bandwidth and the cost of communication is much higher than the computational cost of the operation. When executing `domeigen` 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` scale nicely.

Listing 22: Using dispatch to transparently execute a kernel

```
 1
 2 function kernelf(I, args...)
 3     # low-level kernel
 4 end
 5
 6 function f(args...)
 7     execute(kernelf, args...)
 8 end
 9
10 function execute(f, A::Array, args...)
11     for I in eachindex(A)
12         f(I, A, args...)
13     end
14     return nothing
15 end
16
17 function execute(f, A::CuArray, args...)
18     function kernel(f, A, args...)
19         i = (blockIdx().x-1) * blockDim().x + threadIdx().x
20         i > length(A) && return nothing
21         I = eachindex(A)[i]
22         f(I, A, args...)
23         return nothing
24     end
25
26     N = length(A)
27     threads = min(N, 256)
28     blocks = ceil(Int, N / threads)
29     @cuda threads=threads blocks=blocks kernel(f, A, args...)
30     return nothing
31 end
```

4 Unified kernel language for heterogeneous programming

The programming model and abstractions presented in Section 2 are sufficient for many scientific programming tasks. As an example the `DifferentialEquations.jl` [43] is running transparently on GPUs by using the aforementioned abstractions [42, 28].

Even if those abstractions are not enough, Julia's type dispatch allows for transparent execution by providing a small harness and separating the kernel from the execution schedule. Listing 22 is a particular example of this [1].

The `execute` function has two methods and uses method dispatch on the second argument to select between them. The first in Line 10 uses the `Array` type to execute the code on the CPU (note we could have made this a generic fallback, by using `AbstractArray`),

[1]See `https://github.com/JuliaLabs/ShallowWaterBench/blob/aa8b4add55172f0c1920d6f8cebc8eccc424f2b5/src/GPUMeshing/src/GPUMeshing.jl#L30-L61` for a real-life example

Listing 23: Type hierarchy of devices

```julia
abstract type Device end
struct CPU <: Device end

abstract type GPU <: Device end
struct CUDA <: GPU end
```

whereas the second method in Line 17 uses `CuArray` from the `CuArrays.jl` package [26] to execute code on the GPU. The inner function `kernel` in Line 18 uses methods from `CUDAnative.jl` [9] that are specific for CUDA GPUs to calculate the index. This wrapper kernel is then launched on the GPU with the `@cuda` macro.

Users now don't need to reason about where the execution happens, as long as in the opportune moment the `execute` function is being called. This can also happen transparently if the library writer provides an exposed function `f` (Line 6), while the kernel implementation can live in `kernelf` (Line 2).

While this pattern is powerful and often sufficient for even complicated libraries – the `GPUArrays.jl` [27] library uses a structure similar to this – it does not expose low-level GPU features such as shared memory or warp-level synchronization and communication.

As an example implementing discontinuous Galerkin methods on the GPU [37] requires access to shared memory for performant implementations. As part of the CliMA collaboration [2] I developed a package called `GPUifyLoops` that uses a set of macros to implement a unified kernel language, enabling the development of heterogeneous programs. The package was inspired by my collaborators previous experience with OCCA [35].

Host-device interactions Similar to the example in Listing 22 `GPUifyLoops` uses dispatch to choose the device on which a kernel will be launched. However instead of using a positional argument and the array type, GPUifyLoops has an explicit hierarchy of devices, which allows for future extension. The current type hierarchy is shown in Listing 23 and only supports single-threaded CPU and CUDA execution. In the future I plan to extend it with a multi-threaded CPU implementation, `MTCPU <: Device`, and support for AMD GPUs, `AMD <: GPU`. The latter depends on code-generation for AMD GPUs, which is currently in-progress[3].

Users can launch a kernel on a compute device by using the `@launch` macro, which takes

[2]`https://clima.caltech.edu/`

[3]`https://github.com/JuliaGPU/AMDGPUnative.jl`

Listing 24: Demonstration of the `@launch` macro

```
 1
 2 device(::CuArray) = CUDA()
 3 device(::Array) = CPU()
 4
 5 function kernelf(A)
 6     # low-level kernel
 7 end
 8
 9 function f(A)
10     @launch device(A) threads=length(A) kernelf(A)
11 end
12
13 import GPUifyLoops: launch_config
14 function launch_config(::typeof(kernelf), maxthreads, A; kwargs...)
15     N = length(A)
16     threads = min(N, maxthreads)
17     blocks = ceil(Int, N / threads)
18     return (threads=threads, blocks=blocks)
19 end
```

as a first argument the kind of device to launch on and as a second argument the function to execute and that function's arguments. Furthermore it takes the same keyword arguments (Line 10) as the `@cuda` macro from `CUDAnative.jl` allowing for fine-grained control of the launch configuration.

There is a fair amount of flexibility here that allows library writers to decide how to select which device to launch on. In Line 2 I specified one particular way of choosing the device through dispatch on the function arguments, but in `Oceananigans.jl` the device is part of the model definition. Furthermore users can choose to provide a launch configuration out-of-band by implementing the `launch_config` function. In Line 14 is a particular implementation choice that uses reflection on the compiled GPU kernel to obtain the maximum number of threads that are supported in one block and adaptively changes the number of launched blocks accordingly. The `kwargs...` can be used to consider callsite knowledge to change the launch configuration.

All other device interactions are mediated by the particular array implementation, for CUDA as an example that would be `CuArrays.jl` and `CUDAnative.jl`.

Kernel primitives `GPUifyLoops.jl` goal is to simplify writing of heterogeneous code and to reduce boilerplate in users programs. Its main functionality is provided through several macros.

Listing 25: A simple example of the loop macro

```
@loop for k in (1:Nz; blockIdx().z)
    # ...
end
```

Listing 26: Expanded CPU version

```
for k in 1:Nz
    # ...
end
```

Listing 27: Expanded GPU version

```
for k in blockIdx().z
    if (k in 1:Nz)
        # ...
    end
end
```

`@loop` split a Julia `for` loop iteration space into a CPU expression and a GPU expression.

`@shmem` creates a shared memory region on the GPU or a statically sized array on the CPU.

`@scratch` creates a register array on the GPU or a statically sized CPU array.

`@synchronize` synchronizes memory on the warp level on the GPU, no effect on the CPU.

The `@loop` macro takes a Julia for loop expression and translates it to either a GPU or CPU version. The primary difference to a normal for-loop is that the iterator – e.g. `for I in iterator` – is given by a block with two statements. The first statement is the expression for the whole iteration space, as one would execute it on the CPU and the second statement is how to calculate the current index on the GPU.

In Listing 25 we see a reduced example at work, the iteration space spans `1:Nz` and the GPU index is given by `blockIdx().z` a method provided by `CUDAnative.jl`. On the CPU it lowers to equivalent code given in Listing 26 and on the GPU it expands to Listing 27. It is important to note that on the GPU there is a bounds check against the CPU iteration space, otherwise an user error would easily cause executions on CPU and GPU to diverge.

The usage and purpose of both the `@shmem` and `@scratch` are slightly more subtle. Both are required for obtaining good performance on GPUs. `@shmem` is functionally equivalent to *static shared memory* and is used to load data accessed often or irregularly across a

Listing 28: Usage of the shmem, scratch and synchronize macros

```
 1
 2 s_D = @shmem Float64 (N, N)
 3 l   = @scratch Float64 (N, Ne) 1
 4
 5 @loop for e in (1:Ne; blockIdx().x)
 6     @loop for j in (1:N; threadIdx().y)
 7         @loop for i in (1:N; threadIdx().x)
 8             s_D[i, j] = D[i, j, e]
 9         end
10     end
11     @synchronize
12
13     @loop for j in (1:N; threadIdx().j)
14         for i in 1:N
15             l[j, e] += s_D[i, j, e]
16         end
17     end
18 end
```

thread block. On the CPU this turns into a form of cache-blocking. `@scratch` on the other hand is used to obtain an array of registers on the GPU allowing for temporary storage allocation across loop-iterations.

The example in Listing 28 shows these varying use-cases. Firstly in Line 2, we setup a shared memory region of size $N \times N$ that we will fill with data from a global array in Line 8. Next we setup a scratch memory region in Line 3, that we will use in Line 15 to perform a thread-local reduction. Before we can use the shared memory, we have to issue a `@synchronize` statement so that all threads in a thread block are converged to that point. The scratch memory has an additional subtlety that some of it's dimensions are implicit on the GPU, as an example in Line 3 the defined scratch memory has one explicit dimension on the GPU and the trailing dimensions are implicit.

Function replacement Lastly a common issue with Julia GPU support is that there are functions whose definitions are not GPU compatible, e.g. the `log2` function, which is implemented by calling into *libm*. Calling a host library is not possible from the GPU, and therefore users would experience surprising and hard to understand errors from time to time. To mitigate this, as part of the `@launch` macro, `GPUifyLoops` uses `Cassette.jl` [46] to recursively replace functions in the the call tree with GPU compatible functions provided by `CUDAnative.jl`, additionally this is also how the switching between GPU and CPU implementations is implemented.

Adoption The implementation is made available under the MIT license at `https://github.com/vchuravy/GPUifyLoops.jl`, and is being actively used in several simulations, among them a non-hydrostatic ocean model called `Oceananigans.jl` [4] and a discontinous Galerkin code as part of the `CliMA`[5] project. Both applications have been the driving motivation behind `GPUifyLoops` and it's features, and both have reported good performance that is equivalent to hand-written GPU kernels in Julia.

[4]`https://github.com/climate-machine/Oceananigans.jl`
[5]`https://github.com/climate-machine/CLIMA`

5 Static walks through dynamic programs

In Section 1.4 I introduced the Julia type-inference algorithm, which is crucial for the language to achieve good performance. Consequently, when type-inference goes awry, performance degradation will ensue.

In section 1.4 I gave several examples of heuristics triggering, causing code to no longer fully infer. Small changes to code can cause performance changes that can either accumulate slowly with minor changes or be sudden. Sudden slowdowns often means that a critical section of code – may it be a tight loop in a numerical simulation – transitioned from being statically inferred to being executed with dynamic semantics.

Julia has several tools for benchmarking and profiling, however these traditional profilers and debuggers focus purely on effect and not on the cause. The dynamic-static dichotomy in Julia may mean that after finding a region of slow code (via profiling) one has to work through the compile stack (via reflection), to determine the source of the performance slowdowns.

The problems one is looking for are sources for type-inference providing suboptimal results. The origins could be heuristic triggering or the code is dynamic and no static path could be established, or static knowledge is limited or simply not available.

This is further complicated – as discussed in Section 1.4 – by the fact that some heuristics are context sensitive, and are hard to reproduce in isolation.

I have found that code making heavy use of asynchronous programming, as introduced in Section 1.1, or code that relies heavily on abstractions – like Section 2 – become very hard to analyze. In Section 1.3 I introduced the multiple levels of representation used by the Julia compiler and in Section 5.1 I will introduce `Cthulhu.jl` - a tool I built for simplifying this debugging task and to take static walks through dynamic programs. It exposes the multiple representations in an interactive way and tries to faithfully reproduce the type-inference process described in Section 1.4.

5.1 `Cthulhu.jl` – a tool for interactive introspection

`Cthulhu.jl` came about after a particular long and frustrating debugging session with one of my lab-mates[6] In particular we were trying to understand a type-inference problem that occurred several function calls deep. Our method of debugging was to use `@code_typed` to

[6]I have to thank Peter Ahrens for the fiendish problem and inspiration – *Madness; utter madness*

reflect the result of type-inference on our top-level function, find the function call that was returning a ::Any, reconstruct the call signature manually and invoke @code_typed again. The code was a recursive code-generator and we had to traverse many function calls to get to the source of the issue.

Cthulhu.jl simplifies this process and makes the various representations easily accessible. Given a top-level function call f(1.0) as a starting point, the user can call the @descend macro on it to enter the calltree. The user is presented with the output of @code_typed on that function call and with a list of all non-inlined function calls inside the method. The user can then select a particular callsite and descend deeper into the calltree, while keeping the same inference context and parameters. The menu also offers the choice to switch to the output of @code_typed optimize=false, and to dump the outputs of @code_llvm and @code_native.

Cthulhu.jl can only use statically available information and cannot use dynamic runtime information, albeit in limited circumstances for a dynamic callsite, Cthulhu.jl offers the choice between all possible function calls. Furthermore Cthulhu.jl is capable of descending into task thunks – e.g. code that will be run asynchronously, and functions that will be called on the GPU.

To the best of my knowledge Cthulhu.jl is a unique tool, that has no homomorph in another programming language. While it is useful as a debugger, I would hesitate to describe it as such, since it does not debug the program under investigation directly, but rather debugs a part of the compilation process. The only similar tool in the Julia ecosystem is Traceur.jl[7], but it focuses on non-interactive use-cases and dynamic analysis.

Examples The following example in Listing 29[8] highlights some of of the use-cases and features of Cthulhu.jl. It demonstrates an inference failure on the function f in Line 8, where Julia inferred the return value to be Union{BitArray{1}, Array{Union{},1}}.

The function f broadcasts the function contains over an Interval and an array of Float64, how could the return type ever be an array of type Union{} – the bottom lattice element of type-inference.

An initial investigation with @descend f(i, xs) reveals very little and the output is quite verbose, the only notable thing is that there are two paths through the function and one of the branches introduces an array allocation with element type Union{}. The

[7]https://github.com/JunoLab/Traceur.jl
[8]https://github.com/JuliaLang/julia/issues/31890

Listing 29: Inference failure on broadcasting the **contains** function

```
struct Interval <: Number
    a::Float64
    b::Float64
end

Base.eltype(::Interval) = Float64
contains(i::Interval, x::Float64) = i.a <= x <= i.b
f(I::Interval, xs::Vector{Float64}) = contains.(I, xs)
```

computation of the element type is most likely elided, which means we have to turn off optimizations. To reduce the verbosity of the output in all examples line-information printing is turned off.

Listing 30: First analysis step with Cthulhu, inspecting unoptimized Julia IR

```
  %-1  = invoke f(::Interval,::Array{Float64,1})::Union{BitArray{1},
                                                      Array{Union{},1}}
CodeInfo(
1  %1 = Base.broadcasted(Main.contains, i, a)::Base.Broadcast.Broadcasted{...}
   %2 = Base.materialize(%1)::Union{BitArray{1}, Array{Union{},1}}
      return %2
)
Select a call to descend into or  to ascend. [q]uit.
Toggles: [o]ptimize, [w]arn, [d]ebuginfo.
Show: [L]LVM IR, [N]ative code
Advanced: dump [P]arams cache.

  %1  = invoke broadcasted(::typeof(contains),::Interval,::Array{Float64,1})
   %2  = invoke materialize(::Base.Broadcast.Broadcasted{...}
```

In Listing 30 we see the first step down the rabbit hole. After lowering and desugaring `f` turns into two function calls, first the construction of the `Broadcasted` and then, the materialization thereof. We choose to step into the call to `materialize`, and after an additional step into `copy` end up in Listing 31.

In the abridged version of the `code_typed` reflection of `copy(bc)` we can see, that the first calculation of `Union{}` as the eltype happens in Line 8. We can also see that Julia inferred this as a `Const`, so it is no wonder that we did not see this computation in the optimized code. In this particular instance we are computing the eltype of `contains` on `::Interval` and `Array{Float64,1}`.

Stepping into `combine_eltypes` we finally discover the source of our problems. The call to `eltypes` in Line 4 returns `Tuple{Float64, Float64}` and therefore the call to

Listing 31: Stepping into the `copy(Broadcasted)` function called from `materialize`

```
1    %-1  = invoke copy(::Base.Broadcast.Broadcasted{...}
2   CodeInfo(
3
4   # Abridged
5    %12 = Base.getproperty(bc, :f)::Core.Compiler.Const(contains, false)
6    %13 = Base.getproperty(bc, :args)::Tuple{Interval,Array{Float64,1}}
7    (ElType =
8      Base.Broadcast.combine_eltypes(%12, %13))::Core.Compiler.Const(Union{})
9    %15 = Base.isconcretetype::Core.Compiler.Const(isconcretetype, false)
10   %16 = (%15)(ElType)::Core.Compiler.Const(false, true)
11  # Rest of code abridged
12
13  Select a call to descend into or  to ascend. [q]uit.
14  Toggles: [o]ptimize, [w]arn, [d]ebuginfo.
15  Show: [L]LVM IR, [N]ative code
16  Advanced: dump [P]arams cache.
17
18    %12  = invoke getproperty(::Base.Broadcast.Broadcasted{...},::Symbol)
19    %13  = invoke getproperty(::Base.Broadcast.Broadcasted{...},::Symbol)
20    %14  = invoke combine_eltypes(
21               ::typeof(contains),
22               ::Tuple{Interval,Array{Float64,1}}
23            )::Core.Compiler.Const(Union{}, false)
24  # Abridged
```

`return_type`, in Line 6 yields a `Union{}`. Recalling our definition from Listing 29 we intended `Interval` to be used as a `Number` and not as a container that can be broadcasted over. For convenience we implemented the function `eltype`. The `combine_eltypes` function base Julia, using `eltype` as an implicit trait function of containers to establish the return type of broadcasting `contains`. This allows the direct allocation of an output array of the right type and size.

The user code is easily fixed by removing the `eltype` definition, but cases like this make a strong argument that Julia needs a powerful trait or interface system, so that implicit assumptions like this could be checked.

Adoption `Cthulhu.jl` is made available under the MIT license at `https://github.com/JuliaDebug/Cthulhu.jl`, and has adoption among advanced Julia users encountering issues that require inspection through heavily layered abstractions. I personally have found great use for it in debugging interactions between `Cassette.jl`[46] and user code in `GPUifyLoops.jl`.

Listing 32: Unoptimized Julia IR for `combine_eltypes`, showing the source of the inference issue. `eltype` is part of the implicit trait of containers and so the call to `return_type` uses the wrong argument types.

```
  %-1  = invoke combine_eltypes(::typeof(contains),
                                ::Tuple{Interval,Array{Float64,1}})

 %1 = Base._return_type::Core.Compiler.return_type
 %2 = Base.Broadcast.eltypes(args)::Tuple{Float64,Float64}
 %3 = (%1)(f, %2)::Core.Compiler.Const(Union{}, false)
      return %3

Select a call to descend into or  to ascend. [q]uit.
Toggles: [o]ptimize, [w]arn, [d]ebuginfo.
Show: [L]LVM IR, [N]ative code
Advanced: dump [P]arams cache.

 %2  = invoke eltypes(
   ::Tuple{Interval,Array{Float64,1}})::Tuple{Float64,Float64}
 %3  = return_type
   < #contains(::Float64,::Float64)::Core.Compiler.Const(Union{}, false) >
```

6 Conclusion

6.1 Related work

Section 2 is focused on array abstractions and linear algebra since that is the programming model most commonly used in the prototyping stage of engineering applications. Indeed MATLAB and NumPy 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. If they interact with accelerators like GPUs they use libraries, such as ArrayFire [48], which provide functions that can be called from the CPU but are executed on a GPU. 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 [12] and causes composability [33] and extensibility problems. Numba [29] 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.

We have shown that this is not a problem in Julia since the abstractions themselves are implemented in the same programming language as used by the main application or library developer. Furthermore we use higher-order array abstractions to separate the intent of the developer from the actual execution, and we do so in a composable and extensible manner [6].

The idea of separating the algorithm (what to compute) from the schedule (how and where to compute) is most prominent in Halide [44, 31]. Halide uses a domain-specific language (DSL) embedded in C++ to allow programmers to write pipelines (image algorithms) independently of the schedule and then specify a schedule and execution target. 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. Additionally, Distributed Halide [20] allows for the distributed execution of a Halide 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.

Heterogeneous programming has seen a furor of development in the realm of machine learning, mainly in the form of frameworks and DSLs that are capable of transparently using accelerators and scheduling operations in a distributed heterogeneous manner. Frameworks such as TensorFlow [1] and PyTorch [41] make it easy to take advantage of heterogeneous compute resources, but since they are effectively mini languages embedded in Python with their own compiler infrastructure and their own implementation of array abstractions, they fail to compose with the larger Python ecosystem and are hard to extend. In Innes et al. [24] my collaborators and I discussed the reason for this failure of composability in the context of machine learning itself. While these frameworks can be used for engineering workloads, they often require recasting the problem at hand in terms of machine learning and do not cater to the needs of engineers outside machine learning.

On the other side of the spectrum is the development of special purpose HPC languages such as Chapel [17], IBMs X10 [18], 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. There are some initial developments to adapt these HPC languages to heterogeneous computing, but it is not clear how that will play out and if they will manage to address the diverse set of challenges in heterogeneous computing, while providing an attractive and usable programming model.

In C and C++ there is a host of solutions for heterogeneous programming, like CUB [36], Thrust [7], OpenMP [19], OpenACC [47] and several others. There are also several approaches for distributed programming like MPI [38], Legion [5], and UPC++ [3]. Trilinos [23], PetSC [4], and Kokkos [16], are HPC libraries developed to facilitate the reuse of common numerical infrastructure and have found a fervent following in the HPC community. They are large and complicated libraries that achieve excellent performance in cluster environments, and they are well suited for performance engineers comfortable with C/C++, GPU programming, and distributed programming, but they are not as usable as higher-level programming languages and require a higher investment in time and effort to become proficient. They are thereby less suited for an initial exploration and prototyping phase.

The most closely related project to the work presented in Section 4 is OCCA [35], with the major difference being that GPUifyLoops, is more explicit about the calculation of the kernel indices – OCCA using annotations to denotate inner and outer loops. Both approaches are macro based in their host-language, with OCCA targeting C/C++.

6.2 Retrospective

The work on CuArrays.jl and DistributedArrays.jl has made it greatly easier to execute realistic array applications on respectively GPUs and distributed systems. We also show how these array packages compose, and make it possible to target distributed CPUs and GPUs alike.

I have introduced a unified kernel language for heterogeneous applications. This extends the work on array abstraction for scenarios that require fine control to obtain high-performance on GPUs.

Developers often approach me to help with performance issues they encounter, while using or developing abstractions in Julia, it seems that there is an innate complexity in developing well composing, well layered and powerful abstractions. To reduce some of the complexity involved I developed a tool, *Cthulhu*, to help myself and others to navigate some of these challenges working with such a flexible and powerful language. Cthulhu exposes the hidden parts of the compiler and the language.

References

[1] Martn Abadi et al. "TensorFlow: A System for Large-scale Machine Learning". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation.* OSDI'16. USENIX Association, 2016, pp. 265–283.

[2] Eric Allen et al. *The Fortress language specification.* 2005.

[3] John Bachan et al. "The UPC++ PGAS Library for Exascale Computing". In: *Proceedings of the Second Annual PGAS Applications Workshop.* PAW17. ACM, 2017, 7:1–7:4.

[4] 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.

[5] 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 '12. IEEE Computer Society, 2012, pp. 1–11.

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

[7] Nathan Bell and Jared Hoberock. "Thrust: A productivity-oriented library for CUDA". In: *GPU computing gems Jade edition.* Elsevier, 2011, pp. 359–371.

[8] Tim Besard, Valentin Churavy, Alan Edelman, and Bjorn De Sutter. "Rapid software prototyping for heterogeneous and distributed platforms". In: *Advances in engineering software* 132 (June 2019), pp. 29–46. ISSN: 0965-9978. DOI: 10.1016/j.advengsoft.2019.02.002.

[9] Tim Besard, Christophe Foket, and Bjorn De Sutter. "Effective Extensible Programming: Unleashing Julia on GPUs". In: *IEEE Transactions on Parallel and Distributed Systems* (2018).

[10] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D. Ernst. "Debugging distributed systems: Challenges and options for validation and debugging". In: *Communications of the ACM* (2016).

[11] Jeff Bezanson et al. "Julia: A Fresh Approach to Numerical Computing". In: *SIAM Review* 59.1 (2017), pp. 65–98.

[12] Jeff Bezanson, Stefan Karpinski, Viral B Shah, and Alan Edelman. "Julia: A fast dynamic language for technical computing". In: *arXiv preprint arXiv:1209.5145* (2012).

[13] Jeff Bezanson et al. "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.

[14] Jeff Bezanson et al. "Julia: Dynamism and Performance Reconciled by Design". In: *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications.* to appear. ACM, 2018.

[15] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. *A theory of communicating sequential processes.* 1984.

[16] 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.

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

[18] Philippe Charles et al. "X10: An object-oriented approach to non-uniform cluster computing". In: *ACM SIGPLAN Notices*. Vol. 40. 10. ACM. 2005, pp. 519–538.

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

[20] Tyler Denniston, Shoaib Kamil, and Saman Amarasinghe. "Distributed Halide". In: *SIGPLAN Not.* 51.8 (2016), 5:1–5:12.

[21] Thorsten von Eicken, David E Culler, Seth Copen Goldstein, and Klaus Erik Schauser. "Active messages: a mechanism for integrated communication and computation". In: *Proceedings of the 19th annual international symposium on Computer architecture*. Vol. 20. ACM, May 1992, pp. 256–266. ISBN: 9780897915090. DOI: `10.1145/146628.140382`.

[22] Robert L Henderson. "Job scheduling under the portable batch system". In: *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 1995, pp. 279–294.

[23] Michael A. Heroux et al. "An overview of the Trilinos project". In: *ACM Transactions on Mathematical Software* (2005), pp. 397–423.

[24] Mike Innes et al. "On Machine Learning and Programming Languages". SysML Conference. 2018.

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

[26] Julia developers. *CuArrays.jl: CUDA-accelerated arrays for Julia*. 2018. URL: `https://github.com/JuliaGPU/CuArrays.jl`.

[27] Julia developers. *GPUArrays.jl: GPU-accelerated arrays for Julia*. 2018. URL: `https://github.com/JuliaGPU/GPUArrays.jl`.

[28] JuliaDiffEq. *DifferentialEquations.jl v6.4.0: Full GPU ODE, Performance, ModelingToolkit*. `http://juliadiffeq.org/2019/05/09/GPU.html`. Accessed: 2019-5-19.

[29] 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.* LLVM '15. 2015, 7:1–7:6.

[30] 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: Feedback-directed and Runtime Optimization.* CGO '04. Palo Alto, California: IEEE Computer Society, 2004, pp. 75–. ISBN: 9780769521022.

[31] Tzu-Mao Li et al. "Differentiable programming for image processing and deep learning in Halide". In: *Proceedings of the ACM Transactions on Graphics (SIGGRAPH)* 37.4 (2018), 139:1–139:13.

[32] 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.

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

[34] F Mattern. "Virtual time and global states of distributed systems". In: (1988).

[35] David S Medina, Amik St-Cyr, and T Warburton. "OCCA: A unified approach to multi-threading languages". In: (Mar. 2014). arXiv: `1403.0968 [cs.DC]`.

[36] Duane Merrill and NVIDIA-Labs. *CUDA UnBound (CUB) Library.* 2015. URL: `https://nvlabs.github.io/cub/`.

[37] A Modave, A St-Cyr, and T Warburton. "GPU performance analysis of a nodal discontinuous Galerkin method for acoustic and elastic models". In: *Computers & geosciences* 91 (June 2016), pp. 64–76. ISSN: 0098-3004. DOI: `10.1016/j.cageo.2016.03.008`.

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

[39] 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 1994 ACM/IEEE conference on Supercomputing.* IEEE Computer Society Press. 1994, pp. 340–349.

[40] Kiran Pamnany et al. "Dtree: Dynamic Task Scheduling at Petascale". en. In: *High Performance Computing*. Lecture Notes in Computer Science. Springer, Cham, July 2015, pp. 122–138. ISBN: 9783319201184, 9783319201191. DOI: 10.1007/978-3-319-20119-1_10.

[41] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. *PyTorch*. 2017.

[42] Christopher Rackauckas. *Solving Systems of Stochastic PDEs and using GPUs in Julia*. Dec. 2017. URL: http://www.stochasticlifestyle.com/solving-systems-stochastic-pdes-using-gpus-julia/.

[43] Christopher Rackauckas and Qing Nie. "DifferentialEquations.jl A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia". In: 5.1 (2017). Exported from https://app.dimensions.ai on 2019/05/05. DOI: 10.5334/jors.151. URL: https://app.dimensions.ai/details/publication/pub.1085583166%20and%20http://openresearchsoftware.metajnl.com/articles/10.5334/jors.151/galley/245/download/.

[44] Jonathan Ragan-Kelley et al. "Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines". In: *ACM Transactions on Graphics* 31.4 (2012), 32:1–32:12.

[45] Jeffrey Regier et al. "Learning an Astronomical Catalog of the Visible Universe through Scalable Bayesian Inference". In: (Nov. 2016). arXiv: 1611.03404 [cs.DC].

[46] Jarrett Revels et al. *jrevels/Cassette.jl: v0.2.3*. Apr. 2019. DOI: 10.5281/zenodo.2625463. URL: https://doi.org/10.5281/zenodo.2625463.

[47] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. "OpenACC: First Experiences with Real-world Applications". In: *Proceedings of the 18th International Conference on Parallel Processing*. Euro-Par'12. Springer-Verlag, 2012, pp. 859–870.

[48] Pavan Yalamanchili et al. *ArrayFire - A high performance software library for parallel computing with an easy-to-use API*. 2015. URL: https://github.com/arrayfire/arrayfire.

[49] 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*. Springer. 2003, pp. 44–60.

[50] Francesco Zappa Nardelli et al. "Julia Subtyping: A Rational Reconstruction". In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018), 113:1–113:27. ISSN: 2475-1421. DOI: 10.1145/3276483.